ObfuscationGT: A Comprehensive Evaluation of Disassembler Accuracy and Coverage on Obfuscated Code

Caroline König[1,2], Patrick Felbauer[1,2,3], Michael Pucher[1,2,3], David Schmidt[1,2,3], Patrick Kochberger[4], Pascal Lun[1], and Sebastian Schrittwieser[1,3]

[1] University of Vienna, Faculty of Computer Science
[2] Christian Doppler Lab for Assurance and Transparency in Software Protection
[3] UniVie Doctoral School Computer Science
[4] Institute of IT Security Research, University of Applied Sciences St. Pölten

Abstract. Binary disassembly is essential for analyzing malware, finding vulnerabilities, and understanding closed source software. However, disassembly remains inherently lossy. Compilation removes high level constructs such as types, control flow, and variable names, and leaves only low level machine instructions. Static analysis often faces ambiguity, which complicates efforts to reconstruct a program's logic with certainty. Until now, it is unclear how code obfuscation affects the accuracy of disassembly. Many obfuscations aim at hindering semantic understanding rather than misleading disassemblers directly, yet even these introduce atypical constructs such as indirect jumps, inline assembly, or unconventional control transfers that can confuse heuristic-driven analyses. Disassemblers presumably tune their heuristics for compiler generated patterns. Thus, obfuscated code could break these assumptions.
In this paper, we evaluate how code obfuscation techniques impact state of the art disassemblers. We focus on function entry point identification, instruction recovery, and jump table reconstruction. Our results show that obfuscations that introduce complex indirections significantly reduce disassembly accuracy with differences across disassemblers. Additionally, we show that many source code obfuscations persist through compilation optimization and further decrease disassembler performance. Finally, we demonstrate that layering obfuscation techniques can confuse more disassemblers than applying a single transformation in isolation.

Keywords: Static analysis; Code obfuscation.

1 Introduction

Binary disassembly is central for analyzing compiled software. It supports vulnerability discovery, exploit research, and malware analysis. Despite its prominent role in code analysis, disassembly remains inherently lossy because compilation discards semantic information. In architectures such as x86, instructions have variable length, so even deciding where one instruction ends and the following one

begins can be ambiguous. Compilers also place code and data side by side without clear separation, making it impossible to know with certainty whether a sequence of bytes represents instructions or data. As a result, disassemblers operate with uncertainty and rely on approximations and heuristics. Code obfuscation complicates these even further. Methods such as virtualization, control-flow flattening, and Mixed Boolean-Arithmetic (MBA) transformations are widely deployed to protect intellectual property and hinder reverse engineering [27]. These techniques also appear in malware, where they are used to hide program logic and delay detection by automated tools [40]. While techniques such as these do not directly break the disassembly process, they can alter the code in ways that disassembler heuristics tuned to compiler-generated code may fail to anticipate.

Prior work has studied disassembly accuracy in general-purpose settings. Pang et al. [29] analyzed disassembler heuristics and showed how design choices trade-off between scalability and accuracy. Their follow-up work [30] introduced a ground truth benchmarking analysis to evaluate disassembly performance. However, these studies did not consider obfuscated code. Related research on obfuscation has examined how humans cope with obfuscated programs [8, 9, 20, 10, 5]. While such works highlight challenges in manual analysis, they neither evaluate disassembly accuracy nor cover a broad range of obfuscation techniques and layerings thereof. Hence, we lack systematic knowledge on how code obfuscations affect disassembly correctness and coverage, and where current disassemblers fail under obfuscating transformations. This gap has implications for both attackers and defenders. Security analysts require reliable disassembly even when malware authors apply obfuscation. Conversely, software developers who employ obfuscation as a defense need to understand which techniques degrade disassembly most effectively. Without systematic evaluation, both sides operate without a clear understanding of the limitations of disassemblers.

In this paper, we address this gap and answer *RQ1: What is the impact of different obfuscation techniques on disassembly accuracy?* To this end, we use Tigress [36, 12], a widely adopted C source-to-source obfuscator, to apply 61 distinct transformations, including virtualization, control-flow flattening, and opaque predicates. We combine Tigress with OracleGT [30], a framework that generates ground truth metadata of instructions, functions, and jump tables to measure disassembly accuracy. Additionally, we analyze instances where disassemblers produce incorrect output to derive insights into their strengths and weaknesses when processing obfuscated code. Because we employ a source-to-source obfuscator, it remains unclear whether the applied transformations persist in the final binary and thus effectively hinder disassemblers once compiler optimizations are introduced. To address this uncertainty, we compile binaries using five optimization levels and answer *RQ2: How do different optimization levels influence disassembler accuracy for source code obfuscation?* By applying combinations of obfuscation transformations, we investigate *RQ3: Does combining obfuscation transformations strengthen protection against binary disassemblers?*

Our results show that linear sweep disassemblers, e.g., Objdump, Angr, and Binary Ninja, maintain robust instruction and function recovery capabilities, whereas recursive descent disassemblers like BAP, IDA Pro, and Radare2 emphasize precision at the cost of coverage. Self-modifying code particularly impairs recursive descent approaches, and extending the analysis scope increases recall at the expense of precision. Further, our results highlight that higher optimization levels applied to obfuscated code can strengthen protection against disassembly. Finally, strategically layering obfuscation techniques increases the number of disassemblers that struggle to analyze the resulting binaries.
In summary, we make the following contributions:

- We perform the first systematic evaluation of disassembler accuracy and coverage on obfuscated binaries;
- We introduce a benchmarking framework that combines Tigress obfuscations with OracleGT ground truth generation, enabling reproducible and fine-grained measurements;
- We provide insights into how eight analyzed disassemblers behave on protected code, showing their strengths and weaknesses.

To enable reproducibility and future work, we make our source code and artifacts available: https://github.com/CDL-AsTra/ObfuscationGT/.

2 Related Work

Code Obfuscation Obfuscation alters code structures and data in ways that raise the effort required for reverse engineering. Collberg et al. [13] introduced a taxonomy of obfuscating transformations and defined potency (protection of the asset, i.e. an algorithm or data structures), resilience (protection of the transformation itself), stealth, and cost as core evaluation dimensions that continue to serve as a basis for measuring protection strength. Quantifying these dimensions in practice, however, is challenging, which has motivated broad surveys and the analysis of evaluation methodologies. Schrittwieser et al. [33] examined which obfuscating transformations withstand state-of-the-art code analysis by defining concrete attack scenarios. De Sutter et al. [15] expanded this work with a large systematic review of 571 papers and identified recurring methodological gaps, such as weak strength measurements and unrepresentative data sets. Further, there exist formal attempts to quantify obfuscation resilience. Banescu et al. [2] borrowed concepts from cryptography to formalize attacker models and guarantees, and later proposed a framework for selecting the software features that best predict the effort required for automated attacks [34]. From a complementary perspective, empirical studies have examined how specific transformations affect human comprehension and attack effort. Early work by Ceccato et al. [8] showed how obfuscations, such as opaque predicates, reduce human code comprehension. Later experiments broadened this view: Ceccato et al. [9] analyzed how professional reverse engineers cope with obfuscated code; Hänsch et al. [20] showed that obfuscation has a measurable negative effect on

code comprehension; Ceccato et al. [10] analyzed the behavior of professional hackers while performing real attack tasks; and Shalan et al. [5] demonstrated how control flow obfuscations reduce the accuracy of reverse-engineering tasks. Viticchi'e et al. [38] reported similar effects for data obfuscations. Our work addresses an earlier point in the analysis workflow. We quantify how obfuscation affects disassembly correctness and coverage. Since disassembly is the starting point for both automated and human analysts, inaccuracies at this stage directly limit the applicability of the approaches above. Linn and Debray [26] showed that junk byte insertion and branch functions can cause disassemblers to incorrectly disassemble over 93% of instructions in obfuscated binaries, though Kruegel et al. [24] later developed static analysis techniques to partially recover from such obfuscations. Closer to our measurement goal, Yadegari et al. [41] assessed virtualization-based obfuscation by computing a normalized graph edit distance (GED) between the deobfuscated and original control-flow graphs. GED captures how structurally close two graphs are, and thus how faithfully deobfuscation reconstructs the original program, but it does not indicate whether individual instructions, functions, or jump tables were correctly recovered in the first place. Because our focus is disassembly correctness and coverage rather than structural resemblance to the source, we measure against compiler-derived ground truth from OracleGT [30], which yields direct per-instruction, per-function, and per-jump-table metrics that a graph-similarity score cannot provide.

Disassembler Accuracy Disassembler behavior on binary code has been studied extensively in the past. Jiang et al. [23] evaluated ARM binaries and described common challenges such as inline data, dual instruction sets, and ambiguous branch targets. Pang et al. [29] analyzed the heuristics used by widely deployed disassemblers and showed how their design leads to trade-offs between correctness and coverage. Their later framework, OracleGT [30], generates ground truth data directly from the compiler for multiple architectures, allowing precise measurement of disassembler accuracy. Wijayadi et al. [39] proposed a trace-based method to derive ground truth from binaries, although the implementation is not publicly available for reuse. These works focus solely on unprotected binaries. We extend this line of work by evaluating disassembler performance on obfuscated code, where structural changes introduce challenges not present in the unobfuscated setting. We further note that our evaluation is scoped to the disassembly stage. Some source-level constructs, notably switch statements, may not surface as recovered jump tables at the assembly level yet be reconstructed downstream during control-flow structuring or decompilation, where compiler-induced patterns are recovered even at `-O0` [3]. Since disassembly is the input to these later stages, we deliberately measure what disassemblers recover directly and do not credit reconstruction performed by subsequent structuring passes.

3 Methodology

We introduce our methodology for measuring disassembly performance on obfuscated binaries. The analysis proceeds in two phases: first, we generate obfuscated code, and second, we measure the disassembly performance.

3.1 Dataset Generation

We based our dataset on GNU Coreutils [17], which provides code for 102 programs. Since Tigress, the obfuscation framework we used, requires a single C file as input, we merged each program's source files into a single file. We excluded shared library code from the merge and instead linked it to each binary, consistent with the Coreutils build process. This approach (1) simplifies the merging process by reducing the number of files to combine, making it more robust, and (2) reflects realistic obfuscation use cases, where developers protect sensitive functions rather than the entire code to minimize overhead. We performed the merging step with GCC (version 8.4.0), which reduced the number of programs to 93 (91.18%), as nine programs failed to compile from a single-file source.

We selected Tigress (version 4.0.11) for obfuscation as it provides two advantages: (1) it offers a diverse set of obfuscation techniques, e.g., *control flow*, *function*, and *anti analysis* transformations, and (2) it is a C-to-C obfuscation framework. This has the advantage that we can measure disassembler performance through OracleGT, a modified GCC (version 8.1.0) compiler developed by Pang et al. [30]. As updating OracleGT would require major modifications and the merging process was incompatible with GCC 8.1.0, we used different compiler versions for distinct stages of the pipeline. Specifically, GCC 8.4.0 was used only as a pre-processing step to merge source files and apply Tigress transformations, producing an obfuscated C file without generating binaries. All binaries used in our evaluation and their corresponding ground truth were compiled using the OracleGT-modified GCC 8.1.0 toolchain [30]. Since both the executable code and the ground truth are produced by the same compiler and compilation process, instruction-level alignment is guaranteed. Consequently, the use of GCC 8.4.0 does not affect the evaluated binaries or the correctness of the ground truth.

To select obfuscation transformations, five researchers independently identified transformations that could influence disassembler performance by challenging disassembly heuristics. After this, the researchers met to resolve disagreements and consolidate the selection. In the end, we selected 61 transformations belonging to seven obfuscation techniques: *add opaque*, *anti alias analysis*, *anti taint analysis*, *encode branches*, *flatten*, *self-modify*, and *virtualize*. We initially also considered *JIT* and *JitDynamic*, but excluded them due to known limitations of the JIT transformation, as several benchmark functions rely on `__builtin_` calls, which are not supported by this transformation.

Code Obfuscations Techniques The following selected obfuscation techniques and configurations were used to generate our dataset. We refer to each technique by its Tigress name and explain its effect and relevance for disassembly.

Anti Alias Analysis In Tigress, this transformation replaces direct function calls with indirect ones. For example, the call `x = foo(n)` becomes an indirect call through an array of function pointers. Thus, it introduces ambiguity in resolving call targets, which complicates static disassembly.

Anti Taint Analysis By introducing elaborate control flow structures that handle data copies, *anti taint analysis* obfuscates information flow. Instead of direct assignments, the obfuscation uses iterative increments or bitwise copies within a loop. More sophisticated variants use custom signal handlers for copying or inserting implicit flows [7].

Encode Branches By replacing direct jumps with indirect control flow that resolves at runtime, *encode branches* obfuscates branch targets. For this technique, we used the *branchFuns* and *goto2nopSled* strategies offered by Tigress. *BranchFuns* introduces a central dispatch function that overwrites its return address with a target address passed as an argument. *Goto2nopSled* inserts a sequence of NOP instructions before basic blocks and redirects jumps to land within this sled, which conceals the entry point of a block.

Self-modify Self-modifying code alters its instructions during execution. Disassemblers typically recover instructions by linearly or recursively decoding a fixed instruction sequence. When code mutates at runtime, the binary no longer reflects executable instructions, invalidating the assumption of static code immutability. We applied the *arithmetic* and *comparison* transformations. *Arithmetic* modifies arithmetic operations, and *comparison* alters branch conditions to prevent static inference, resulting in apparently nondeterministic control flow paths. Each transformation type combines with an *update style*, which defines where changes occur and how inline assembly manages register state. The *text* style rewrites instructions directly in the text segment. The *stack* style uses inline assembly that saves and restores registers. The *clobber* style relies on the compiler clobber list to track and preserve register state during modifications.

Opaque Predicates By introducing complex expressions that evaluate to a fixed value at runtime, *opaque predicates* increase the apparent branching factor and inject spurious control flow edges and call targets, potentially lowering the precision of path sensitive analysis and confusing predicate evaluation heuristics. The types of *opaque predicates* in Tigress describe how the control flow is altered. *Call* inserts a random conditional call. *Bug* hides buggy code behind a condition evaluating to false. *True* gates code behind a condition that is always true. *Junk* inserts random bytes into a branch that is never picked. *Question* duplicates code in conditionals that evaluate to either true or false. Tigress supports various opaque constructs, namely *array*, *env*, or *list*: *array* generates opaque expressions

using arrays, *env* derives expressions from entropy, and *list* builds expressions through linked list structures.

Flatten By centralizing the order in which basic blocks are executed through a dispatch mechanism, *flatten* obfuscates the control flow. Tigress implements this technique using either plain label jumps (*goto*), label arrays (*indirect*), function pointers (*call*), or switch case dispatching (*switch*), similar to the approach of László et al. [25]. We included *flatten* in our analysis because the dispatcher introduces control flow indirection, which could complicate control flow reconstruction and hinder disassembly.

Virtualize Virtualization replaces a function's native instructions with a random bytecode representation, embedding an interpreter that translates it to native code at runtime. We applied virtualization using eight dispatcher styles, namely *switch*, *direct*, *indirect*, *call*, *ifnest*, *linear*, *binary*, and *interpolation*. These styles differ in the way the interpreter dispatches bytecode instructions and encodes control flow transitions within the virtualized execution. Disassemblers rely on static patterns to identify instruction boundaries and recover code structure. By replacing instructions with opaque bytecode interpreted at runtime, virtualization removes these patterns and introduces custom control logic.

Code Obfuscation Layering Additionally, to applying each obfuscation individually, we evaluated pairwise combinations. Because the number of possible combinations grows exponentially, we limited the configurations by selecting at most two per technique. We selected them based on the average F-score of the individual techniques, favoring diverse variants over related subvariants. This procedure yielded 13 configurations, as only one variant of *anti alias analysis* appeared in the individual round, resulting in 156 samples per program.

3.2 OracleGT

Pang et al. [29, 30] introduced OracleGT to generate ground truth data to measure disassembler performance by tracing the compilation, assembly, and linking stages. They instrumented compilers and linkers to record metadata about instructions, functions, and jump tables. Unlike symbol-based methods, this approach captures compiler-recognized constructs and resolves overlaps, unlabeled instructions, and handwritten jump tables.

We evaluate the disassembler performance of Angr [35, 1] (9.2.166), BAP [6, 11] (2.5.0), Dyninst [4, 31] (13.0.0), Ghidra [28] (11.4.1), Objdump [18] (2.38), Radare2 [32] (6.0.3), Binary Ninja [37] (5.0.7648), and IDA Pro [22] (9.1). We evaluate these eight disassemblers on x86/x64 binaries using the instruction, function, and jump-table ground truth provided by OracleGT. Although OracleGT also supports ARM32, AArch64, MIPS32, and MIPS64, we focused on x86/x64 to compare our results to the first work of Pang et al. [29]. Further, Tigress [12] does not support AArch64, MIPS32, and MIPS64.

Table 1: Statistics of complex constructs. *Avg.* denotes the average number of occurrences within binaries that contain at least one instance, and *Bin* indicates the percentage of compiled binaries in which at least one instance.

Constructs	**Types**	**Avg.**	**Bin**
Data in Code	Padding bytes	6.01	98.02%
Indirect Jmp.	Jump tables	3.20	26.79%
	Indirect tail-calls	5.44	98.02%
Special Func.	Overlapping functions	1.00	0.08%
	Multi-entry functions		
	Non-return functions	3.97	97.99%
(Direct) Tail-calls		1.00	98.02%
Total			3,938

Tigress inserts jump labels together with self-modifying code to ensure portability of the generated C code, which enables OracleGT to include all referenced instructions, basic blocks, and functions in the ground truth. Thus, we can use the ground truth that OracleGT generates, although some obfuscations add inline assembly and runtime modifications. During assembly, OracleGT flags inline assembly because it emits dedicated metadata for these regions, and it counts their instruction bytes as part of the preceding basic block. OracleGT then performs additional post processing steps to ensure that the ground truth contains all instructions. This step matters in particular for data directives such as `.byte`, because Tigress uses these directives.

4 Obfuscation Impact

In the following, we compare our dataset and the unobfuscated results to the work of Pang et al. [29], who introduced the ground truth generation framework. Afterwards, we compare how disassemblers perform across instruction, function, and jump table recovery under different obfuscation configurations. All results in this section were obtained using the `O0` optimization level, meaning that the compiler applied no optimizations, the default setting in GCC [19]. In Section 5, we further analyze how optimization levels impact disassembly performance.

4.1 Dataset Comparison

Table 1 summarizes the program elements in our binaries as identified by OracleGT, and Table 3 (Appendix) presents the disassembly results for unobfuscated binaries in comparison with Pang et al. [29].

Dataset While our dataset included fewer programs, the applied obfuscation transformations produced 3,938 binaries, exceeding the 3,788 binaries of Pang

et al. Our samples are smaller because we focused on disassembler performance under obfuscation, which requires a merged C file, resulting in less complex structures. For example, 25.55% of binaries contained jump tables with an average of 3.04, affecting precision and recall, as errors have a proportionally larger impact.

We updated disassembler versions, which may have introduced performance changes. Prior work by Crussell [14] showed that newer Ghidra versions do not always improve performance. Developers struggle to ensure new heuristics generalize, as tested cases rarely capture real-world diversity.

Instructions, Functions and Jump Tables Instruction recovery on unobfuscated binaries showed similar precision and recall. In Pang et al. [29], only BAP and Radare2 had a recall below 99%, with values of 82.5% and 87.92% respectively. On our dataset, Radare2 dropped further to 71.24%, while BAP improved to 95.57%. Dyninst's recall worsened to 69.97%.

For function entry detection, precision remained high in both works, except for Angr and BAP, whose already low precision in Pang et al. declined further in ours. Recall changed most for Angr and Radare2. For Angr, recall decreased from 98.29% to 95.66%, and for Radare2 from 97.01% to 90.55%.

Jump table recovery was lower, likely due to reduced code complexity. Despite differences, manual analysis verified the validity of our results. These differences also do not affect our conclusions on obfuscation strength, since all comparisons use the same binaries.

4.2 Obfuscation

Table 2 displays the average results for all disassemblers on obfuscated binaries. For clearer presentation we aggregated sub-variants of obfuscation techniques, e.g. opaque predicate variants, through averaging. We omitted BAP, Dyninst and Radare2 from the table to highlight the disassemblers that perform the best and we omitted Objdump as it performs best at instruction recovery, but does not provide function or jump table analysis. We also omitted obfuscations which hardly influenced disassembler performance, i.e. *anti alias analysis* and *anti taint analysis*, and variants of virtualization that result in similar behavior in disassemblers. We provide the full table as part of our artifact [16].

Add Opaque For instructions, inserted junk data worsened recall, causing IDA Pro to drop from 99.94% to 59.23%. As OracleGT extracted basic blocks only from functions that the disassembler identified, IDA Pro excluded all instructions from blocks that it failed to assign to a function. Thus, it misclassified junk instructions in top level functions, which prevented analysis of all functions further down the call graph. The recall issues of Dyninst and Radare2 on junk data stemmed from weak performance on the unobfuscated binary.

Function recovery mirrored instruction recovery in recall, albeit with smaller reductions. Except for IDA Pro on the *junk* configuration and Dyninst, function recovery remained robust in the presence of opaque predicates. The trend of

Table 2: Average precision and recall for `OO` by obfuscation technique and disassembler. We highlight values in red when they are at least 10% lower than on unobfuscated binaries, and in green when they are 10% higher. We aggregate sub-variants by averaging include the full table in our artifact [16].

	Obfuscation	Angr		Ghidra		IDA Pro		Binary Ninja	
		Prec.	Rec.	Prec.	Rec.	Prec.	Rec.	Prec.	Rec.
Instruction	unobfuscated	99.85	99.95	99.86	99.96	99.86	99.94	99.86	99.86
	add opaque bug	99.90	99.96	99.90	99.96	99.90	99.96	99.90	99.86
	add opaque call	99.89	99.96	99.89	99.97	99.89	99.96	99.89	99.89
	add opaque junk	99.69	99.04	99.59	97.12	99.71	59.22	99.55	99.01
	add opaque question	99.95	99.96	99.95	99.98	99.95	99.94	99.95	99.74
	add opaque true	99.86	99.96	99.86	99.97	99.86	99.96	99.86	99.88
	encode branches branchFuns	97.20	99.02	99.59	76.61	99.77	58.43	98.28	92.96
	encode branches goto2nopSled	99.91	99.91	99.85	72.74	99.89	82.22	99.83	72.60
	flatten call	99.95	100	99.95	99.58	99.95	100	99.95	99.75
	flatten goto	99.90	99.68	99.90	98.29	99.85	48.64	99.79	32.63
	flatten indirect	99.92	99.98	99.88	79.13	99.89	62.81	99.78	24.88
	flatten switch	99.91	99.99	99.91	99.78	99.91	99.99	99.87	63.71
	self modify arithmetic	82.61	91.59	94.76	98.76	91.72	28.68	93.01	88.05
	self modify comparisons	90.64	97.22	94.48	93.40	87.06	35.34	94.08	96.05
	virtualize call	100	100	100	100	100	100	100	100
	virtualize switch	100	100	100	98.59	100	100	100	96.84
	virtualize interpolation	100	100	100	82.03	100	76.02	100	71.70
	virtualize indirect	100	100	100	99.99	100	99.83	100	65.67
	virtualize direct	100	100	100	93.53	100	22.04	100	18.39
Function	unobfuscated	63.11	95.66	98.88	100	99.25	95.66	98.49	99.96
	add opaque bug	64.40	96.14	98.95	100	99.33	96.14	98.45	99.93
	add opaque call	64.63	96.14	98.92	100	99.32	96.14	98.53	99.98
	add opaque junk	59.71	95.59	96.72	100	96.81	83.16	95.41	99.55
	add opaque question	73.43	97.67	99.46	99.99	99.64	97.71	96.37	99.97
	add opaque true	64.94	96.14	98.96	100	99.28	96.14	98.50	99.95
	encode branches branchFuns	55.89	95.82	99.01	100	98.89	83.35	92.53	99.81
	encode branches goto2nopSled	55.99	96.22	99.02	100	87.38	96.23	97.91	99.46
	flatten call	90.38	97.64	99.95	100	99.98	99.08	99.24	96.01
	flatten goto	19.60	95.86	99.56	100	84.46	95.40	96.07	95.04
	flatten indirect	15.07	95.86	99.56	100	38.08	94.60	93.62	94.97
	flatten switch	62.79	95.92	99.56	100	99.83	95.92	44.93	89.75
	self modify arithmetic	14.86	95.06	99.10	100	98.52	94.17	28.34	99.17
	self modify comparisons	28.54	95.18	99.14	100	98.48	93.77	53.08	99.72
	virtualize call	90.34	98.94	99.99	100	100	98.94	98.82	99.97
	virtualize switch	61.03	94.93	99.88	100	100	94.93	97.49	100
	virtualize interpolation	13.14	94.78	99.88	100	86.27	94.24	87.10	94.69
	virtualize indirect	15.23	94.88	99.88	100	18.08	94.77	95.71	94.17
	virtualize direct	14.91	94.86	99.88	100	95.07	95.01	96.72	94.35
Jump Table	unobfuscated	23.82	39.86	76.13	76.58	77.21	81.08	76.58	79.05
	add opaque bug	19.60	34.93	75.80	76.26	76.89	80.82	76.41	79.23
	add opaque call	21.06	38.24	75.80	76.26	76.89	80.82	76.49	79.45
	add opaque junk	24.11	33.98	77.26	77.38	73.02	74.86	73.50	75.30
	add opaque question	13.04	27.55	78.52	78.06	79.61	81.13	79.16	79.96
	add opaque true	22.25	38.58	75.80	76.26	76.89	80.82	76.30	78.88
	encode branches branchFuns	3.72	17.25	76.96	76.36	77.17	77.71	77.77	79.49
	encode branches goto2nopSled	23.74	39.04	67.35	68.95	76.48	78.08	53.21	57.65
	flatten call	18.46	34.25	78.49	80.82	77.92	80.82	77.92	80.82
	flatten goto	29.43	43.02	74.57	74.32	74.32	75.68	72.97	72.97
	flatten indirect	29.61	44.59	76.98	78.27	76.64	81.08	72.97	72.97
	flatten switch	67.45	100	95.73	100	94.82	99.76	95.15	100
	self modify arithmetic	29.22	43.36	8.66	18.87	71.43	71.43	75.80	79.76
	self modify comparisons	30.56	43.40	2.91	15.18	74.70	76.79	76.78	79.91
	virtualize call	76.39	100	97.44	100	95.88	100	95.88	100
	virtualize switch	81.67	99.28	97.74	93.77	96.81	100	96.81	100
	virtualize interpolation	79.12	100	78.12	78.12	78.12	78.12	82.29	90.62
	virtualize indirect	76.78	100	96.45	100	94.53	100	78.12	78.12
	virtualize direct	79.96	100	96.55	88.95	83.33	90.91	78.79	78.79

erratic behavior caused by weak baseline performance continued for Angr, as precision even improved under the *question* configuration.

Despite disassemblers not achieving high scores for jump tables, the differences remained negligible compared to the unobfuscated baseline. Additionally, the binaries contained few jump tables without opaque predicate conditionals, introducing additional ones. While Angr's jump table recovery appeared heavily impeded, the large number of False Positives (FPs) stemmed from its detection of successor basic blocks after return statements. OracleGT interpreted this behavior as if Angr had discovered a jump table, even though Angr only marked potential return addresses of a function.

Anti Alias Analysis *Anti alias analysis* had little impact on disassemblers, except for Radare2, and we therefore did not include it in Table 2. The recall of Radare2 dropped by 38.77% for instructions, 18.73% for functions, and 5.41% for jump tables, as it failed to detect indirect calls and did not follow them during recursive traversal. In addition, Ghidra dropped by 18.92% in precision and 16.22% in recall for jump tables, because it translated function calls into jump tables and enforced a size limit on jump tables during auto analysis.

Anti Taint Analysis *Anti taint analysis*, also not included in Table 2, showed minor impact, highlighting that most disassemblers in their default configuration did not rely on data flow analysis that anti taint obfuscation could disrupt. Radare2 is the only disassembler that exhibited a recall drop of more than 10% for instructions. In these cases, Radare2 struggled to identify the main function when *anti taint* was applied to the program arguments, which suggests that the initialization code of this obfuscation broke its heuristic for determining the main function.

Encode Branches *BranchFuns* caused large recall drops for BAP, Dyninst, Ghidra, IDA Pro, and Radare2 in instruction analysis by hiding jump targets and overwriting return addresses. Junk bytes inserted by Tigress desynchronized the disassembly for some instructions. This gave an advantage to linear sweep Objdump and brute force Binary Ninja, while hybrid approaches like IDA Pro stopped linear sweep once they encountered invalid instructions, missing large parts of a function. Angr offered more precise stack analysis, but its high recall relied on a linear sweep heuristic, and disabling it reduced precision.

Goto2nopSled produced similar behavior, hiding jump targets via indirection and arithmetic operations. IDA Pro improved as instruction recovery progressed further when no return address overwrites interfered with stack analysis. Other disassemblers had lower recall due to increased instruction count, with Binary Ninja dropping further as its function detection heuristic triggered less often.

Most disassemblers showed similar function level performance on *encode branches* except for IDA Pro and Binary Ninja. Binary Ninja's precision dropped for *branchFuns*, indicating that it marks code at the beginning of basic blocks

following the branch function as separate functions, increasing the FP. For *goto2nopSled* this heuristic no longer triggered, potentially because the NOP sled did not satisfy it, giving Binary Ninja higher function precision but lower recall. For *branchFuns*, the recall of IDA Pro was only 83.35% due to failing stack analysis. The precision dropped to 87.38% for *goto2nopSled*, because jump targets introduced by indirect jumps were marked as functions.

The jump table precision and recall of Angr dropped to 3.72% and 17.26% for *branchFuns* due to the transformation rewriting the return address. Thus, Angr interpreted the successive return instruction as a jump table. In the *goto2nopSled* variant, Binary Ninja struggled to determine the jump targets of the newly added indirect jumps and identified jump tables that are not part of the ground truth.

Flatten *Call* affected Radare2 and Dyninst for instruction recovery as these failed to resolve function pointer targets and did not disassemble further. *Goto* additionally affected BAP, IDA Pro, and Binary Ninja. Radare2 refused to resolve indirect jumps while Binary Ninja failed to interpret memory values encoding jump targets as offsets. In *call*, all targets represent functions, which Binary Ninja detected and linked to entries in the function pointer table. For *goto*, however, the targets are basic blocks and did not trigger the heuristic. IDA Pro disassembled referenced instructions but did not assign them to functions, causing OracleGT to ignore them. *Goto* moved dispatch logic into basic blocks, chaining it to other basic blocks through direct jumps and keeping only the initial jump indirect, whereas *indirect* used jump tables and centralized the dispatch logic in the caller. This reduced the recall of Ghidra, while that of IDA Pro improved due to its strong jump table heuristics. Ultimately, the weak performance on indirect control flow stemmed from additional indirections, with Tigress copying tables onto the stack instead of using read-only memory.

Most disassemblers improved on *switch* as it represents a common construct. Some, however, struggled with the size of the switch statements. For example, Binary Ninja did not automatically disassemble a function with more than 200 switch cases, and forcing analysis triggered an endless heuristic loop.

Function recall remained stable, because the comparison only considered function entry points. Exceptions were Dyninst and Radare2, which lacked additional heuristics for detecting function entry points and failed to identify indirect function pointers in the *call* variant. Recall also dropped for Binary Ninja in the *switch* variant.Precision, however, dropped when disassemblers could not assign basic blocks to their functions, creating single-block or overlapping functions instead. For IDA Pro, Dyninst, and Angr, this occurred for *goto* and *indirect*, whereas Binary Ninja produced many FP for *switch*. Binary Ninja's heuristics appeared to recognize parts of the large switch statements as separate functions. However, the same heuristic did not identify the basic blocks in *goto* and *indirect*.

Jump table performance mostly matched the baseline, except for Dyninst, whose results were skewed by missed disassembly outside of *switch*. These fluctuations resulted from poor performance on related tasks, since OracleGT ignored jump tables whose underlying instructions were not disassembled, amplifying the

effect of even a single False Negative (FN) or FP. *Flatten switch* introduced new jump tables. All disassemblers but Radare2 found them, but Angr and Dyninst classified indirect jumps too liberally.

Self-modify *Self-modify* strongly impacted instruction disassembly, causing the largest increase in FP among all obfuscations, by interleaving data with code with an increased impact from the high number of insertion points. This was especially visible in instruction disassembly precision, where *self-modify* was the only category in which Objdump dropped below 90%.

Linear sweep based disassemblers recovered substantially more instructions, reaching recall values above 88.05% for *arithmetic* and *comparisons*, while recursive descent disassemblers recovered around 18.97% to 39.08%. Ghidra is the exception, achieving a recall of at least 94.48% due to its hybrid architecture. While its core followed recursive descent, it also employed heuristics to "enlarge code coverage" [29]. One such heuristic, "disassembly from targets of xrefs", enabled Ghidra to discover dynamically called code blocks by analyzing data to code references that other recursive disassemblers missed.

Self-modify arithmetic array embedded arithmetic logic in inline assembly blocks connected through runtime patched indirect jumps. Each block ended with `jmp qword ptr [rip+0]` (`FF 25 00 00 00 00`), with the subsequent eight bytes overwritten at runtime to point to the next block, creating self-referential control flow edges. As shown in Listing 1, the disassembly exposed arithmetic sequences followed by self rewriting jump targets. This concealed the control flow and blocked recursive disassemblers from resolving or identifying functions with runtime-changing targets. Data embedded in code created another point of failure, as the constants used to compute jump targets desynchronized the disassembler. Linear sweep disassemblers tried to resynchronize to valid instructions, whereas recursive traversal and hybrid disassemblers usually stopped once they encountered an invalid instruction. IDA Pro lost portions of the disassembly despite knowing constant sizes and cross references.

Forcing Radare2 to continue analysis beyond the inferred function boundary increased its performance, even if no caller reached the subsequent code block. The recall for *arithmetic array* increased from 26% to 94%, while precision dropped from 95% to 78%. This effect was confirmed on curated examples by using Radare2's more aggressive analysis. The results highlight a fundamental trade off in disassembly strategies, where linear sweep methods favor coverage while recursive methods often sacrifice recall to maintain precision or soundness.

Function recall dropped slightly for most disassemblers. Only BAP deviated to recovering 70% to 80%. *Self-modify* introduced indirect jumps resisting static analysis, leaving basic blocks orphaned without a function assignment. IDA Pro missed blocks by failing to assign disassembled blocks to functions. In contrast, Angr and Binary Ninja disassembled the code behind the constants and created new functions when they could not link blocks to existing ones, inflating FP.

Jump table performance stayed near baseline, with deviations mainly due to OracleGT excluding undisassembled regions from the ground truth. This artifi-

```
00402283    STD
00402284    MOV    qword ptr [RCX]=>DAT_9cf4c3368d626989, RAX
00402287    STD
00402288    JMP    qword ptr [DAT_0040228e]
0040228e 48 ??
```

Listing 1: Ghidra disassembly code illustrating a *self modifying arithmetic array* block that ends in a indirect jump. At runtime, the eight bytes following the `jmp qword ptr [rip+0x0]` instruction are overwritten with the jump address.

cially increased recall and precision as fewer jump tables remained to identify, leaving fewer opportunities to misinterpret constructs as jump tables.

Ghidra recovered functions and instructions well, which should lead to broad coverage. OracleGT, however, treated basic blocks without function assignments as freestanding blocks and included them in the ground truth. Since Ghidra never assigned these blocks to functions, it did not analyze them entirely. As a result, it missed legitimate jump tables whose targets depended on stack variables and therefore lacked a function context. The reported FP stemmed from cases where Ghidra interpreted a fall through as a valid jump edge.

Virtualize Out of the variants we analyzed, we omitted the *ifnest* variant from Table 2 since disassembler performance was identical to the *switch* variant, as well as *linear*/*binary* which are different strategies for the *interpolation* variant. Precision remained near 100% for instruction recovery, as increased code size reduced absolute differences. Recall was mainly affected by *call*, *ifnest*, and *switch* for BAP, Dyninst, and Radare2. Other variants also impacted Ghidra, IDA Pro, and Binary Ninja. Since no junk data or self-modification was introduced, linear sweep disassemblers were largely unaffected. Radare2 was most affected due to its instruction limit per basic block. Dyninst recovered more instructions than on unobfuscated binaries, as the table setup occurred at the beginning of the first block and required no control flow recovery, but failed to recover functions referenced in the table for *call*.

Other variants used memory indirection to hide jump targets. Manual analysis of binaries with *interpolation*, *linear*, *binary*, *indirect*, and *direct* showed that all disassemblers struggled at mapping basic blocks to functions. *Direct* splits a function into basic blocks and schedules their execution by appending the same *goto* at the end as in Figure 1a. The compiled binary preserved this structure, as each block determined its successor by reading from `pc[0]`. As shown in Figure 1b, part of this indirect jump was duplicated in every basic block.

Instruction coverage dropped for BAP, Radare2, and IDA Pro. Ghidra leveraged addresses in `.rodata` and thus detected more basic blocks, yet failed to map these to functions. The evaluation of the jump indirection results improved when the obfuscation became simpler. For *interpolation*, *binary* and *linear*, jump table

```
LABEL_BLOCK_0:
  (pc[0]) ++;
  (sp[0] + -1)->_int &= (sp[0]
  ↪  + 0)->_int;
  (sp[0]) --;
  goto *(*(pc[0]));
```

(a) Tigress

```
11cb8 JMP  RAX
...
11dce MOV  RAX,qword ptr [RBP
↪  + -0x318]
11dd5 MOV  RAX,qword ptr [RAX]
11dd8 JMP  LAB_00411cb8
```

(b) Ghidra

Fig. 1: Obfuscated code transformed with *virtualize direct*. Figure 1a shows the obfuscated C code, while Figure 1b shows the disassembler output of Ghidra. `pc[0]` references an entry in an array of operands and labels defined within the function. The Ghidra disassembly shows that all basic blocks dispatch control flow through a central `JMP RAX` instruction.

entries are calculated, whereas *indirect* directly stores the program addresses in `.rodata`, benefiting Ghidra and IDA Pro but not others. Using *direct*, which employed memory indirection, yielded the worst performance in the simplest case across all disassemblers. *Direct* produced less initialization code in the first basic block of each function. The smaller first block reduced the overall code size. It lowered instruction recall, as the recursive traversal now must resolve the correct jump target to reach the same fraction of instructions. The same effect appeared in the other jump indirection variants, since the amount of initialization code in the first basic block decreased for *interpolation*, *binary*, *linear*, and *indirect*.

For functions, recall worsened slightly but remained stable except for Dyninst. The main differences appeared in precision. When Tigress split code into basic blocks behind indirect jumps, disassemblers often detected the blocks but emitted new functions. This occurred for Angr, Binary Ninja, BAP, and IDA Pro. For *call*, Tigress transformed basic blocks into functions, which improved function detection for Angr and BAP, as both tended to emit functions for stray basic blocks. Dyninst did not use code references in memory as a heuristic, missing many functions while also marking numerous stray blocks as functions.

The *call*, *ifnest*, and *switch* variants inserted new jump tables through C constructs, such as `switch` statements and `if` chains, which compilers translated into jump tables. Thus, all disassemblers improved compared to the baseline. In the other variants, Tigress interleaved `switch` statements with indirect control flow. OracleGT classified this indirect control flow as non jump table logic, so disassemblers overapproximated the reachable edges and produced FP. Recall mirrored the trends from function and instruction recovery, because jump table recovery correlated with the amount of recovered code.

4.3 Discussion

Recursive descent tied control flow recovery to disassembly, which improved precision but reduced coverage once structural obfuscation appeared. Techniques

such as *flatten indirect* and *virtualize direct* reduced recall to below 30%, as indirect jumps driving the central dispatcher could not be resolved. Ghidra, however, used data cross references to identify code entry points and extracted basic blocks that lay outside functions, outperforming other disassemblers.

Objdump achieved the strongest instruction recovery, although it did not provide control flow information. Angr bypassed this restriction by discovering everything outside the reconstructed control-flow with linear sweep. This yielded results close to Objdump while still offering flow information to analysts. As a source-to-source obfuscator, Tigress cannot apply sophisticated desynchronization attacks against linear sweep strategies. Even with junk data, linear sweep recovered quickly and outperformed most recursive descent approaches.

The results of instruction and function disassembly depended on each other. For recursive descent strategies, discovering more functions increased recall for instruction recovery. For linear sweep strategies, recovering more instructions increased the FP of function detection, because disassemblers often classified freestanding basic blocks as functions. Beyond these trends, identifying functions and jump tables reduced to matching compiler generated patterns. To do so, Binary Ninja searched for function prologues, and Ghidra identified functions by tracking memory references. Similarly, most disassemblers matched typical *switch* jump tables, but they struggled with unusual indirect control flow.

Takeaways

- The impact of obfuscation techniques depend on the strategies that disassembler apply;
- Different variants of the same transformation can lead to notably different outcomes;
- *Encode branches*, *flatten*, and *self-modify* introduced the most severe degradation in accuracy.

5 Optimization Comparison

In this section, we study how compiler optimizations influence disassembly success on obfuscated binaries. We re-analyzed the complete dataset with `O1`, `O2`, `O3`, and `Os` and report the full results as part of our artifact [16].

5.1 Add Opaque

For instruction analysis, *add opaque junk* impacted IDA Pro noticeably, whereas other *add opaque* types caused only minor deviations from `O0`. Compiler optimizations relocated junk data into non returning function regions, which reduced the need for jumps over embedded data. As a result, IDA Pro stopped disassembling prematurely and overlooked substantial portions of the instruction stream.

For *add opaque true*, Ghidra's jump table detection improved under optimization. The compiler replaced `JMP RAX` instructions and their preceding

address calculations with a single jump instruction, e.g., `JMP qword ptr [PTR_LAB_00408560 + RAX*0x8]`, which matched Ghidra's jump table patterns more reliably. Although Ghidra could not apply advanced analysis because these jumps lay in blocks outside any function, the optimized instruction revealed that at least one jump target exists. Thus, Ghidra correctly reported these jump tables.

5.2 Flatten

Optimization substantially improved Binary Ninja's instruction handling of *flatten goto* as it removed the obfuscation's dispatch mechanism. The variable used for the `goto` statement received a single assignment and never changed, eliminating the dispatcher and restoring a control flow close to the unobfuscated program. Function recovery for Angr showed the same effect, with higher F-scores starting at `O1`. *Flatten indirect* also benefited from optimization. At `O0`, the dispatch table appeared as a raw memory block. At higher levels, the compiler lowered it to a local array, making the dispatch structure explicit. *Flatten switch* resulted in an increase in the F-score at `O1` but a decrease for other levels. At `O1`, case merging and arithmetic state transitions simplified the control flow and improved analysis. At `O2`, the switch lowered to an indirect jump table, and Binary Ninja failed to reconnect table entries back into the enclosing function.

For function analysis, *flatten switch* affected Angr most strongly. Optimization turned the indirect jump of the switch statement into an indirect jump table. Manual analysis of `csplit` revealed that this led Angr to miss successors of indirect jumps and to declare these as new functions. For Binary Ninja, the results initially improved, as the optimized dispatcher became a simplified sequence of switch cases that helped the disassembler identify function entry points. At `O2`, the compiler outlined parts of the dispatcher into helper functions, such as a non returning dispatcher. Binary Ninja followed these compiler introduced function boundaries and reported additional function starts that did not appear in the ground truth, merging this code into `main`. This increased recall but introduces FPs and lowered the F-score compared to `O1`, while outperforming `O0`.

For jump tables, Angr showed a performance inversion for *flatten switch* compared to its behavior with other techniques. It reached a relatively high F-score at `O0`, then collapsed to values between 11% and 13% for `O1` through `O3`. This inversion stemmed from the fragility of Angr's `JumpTableResolver` [35]. At `O0`, the unoptimized dispatcher loop preserved a syntactic structure that allowed Angr to detect switch targets through pattern matching. Compiler optimizations disrupted this structure, breaking the resolver's assumptions.

5.3 Self-modify

Arithmetic and *comparison* yielded lower F-scores for Angr, Ghidra, and Binary Ninja for instructions at higher O-levels, while IDA Pro improved for *arithmetic*. The decline stemmed from self-modifying constructs introducing read/write operations at `O0` which were removed at higher levels, leading to earlier disassembly termination and reduced recall. IDA Pro, however, continued disassembly.

IDA Pro's F-score dropped at `O1` for *arithmetic* and *comparison* for function detection and increased at higher levels, although remaining below `O0`. All O-levels contained opaque indirect jumps, yet `O0` preserved functions with simple CFGs that IDA Pro resolved reliably. At `O1`, large parts of `main` appeared as inline patch tables behind opaque `jmp qword` instructions that IDA Pro cannot follow. At `O2`, these patterns became more regular, enabling partial recovery.

As for jump tables, Ghidra improved with optimization. For example, *comparison clobber* increased from 4% at `O0` to 40% at `O2` through `Os`. At `O0`, self-modifying memory writes disrupted Ghidra's control flow analysis and truncated function analysis, which prevented it from reaching the jump tables. At `O2`, `O3`, and `Os`, compiler optimizations removed these writes, which enabled Ghidra to disassemble the full function. Once the code containing the jump tables became reachable, Ghidra's heuristics identified and resolved them.

5.4 Virtualization

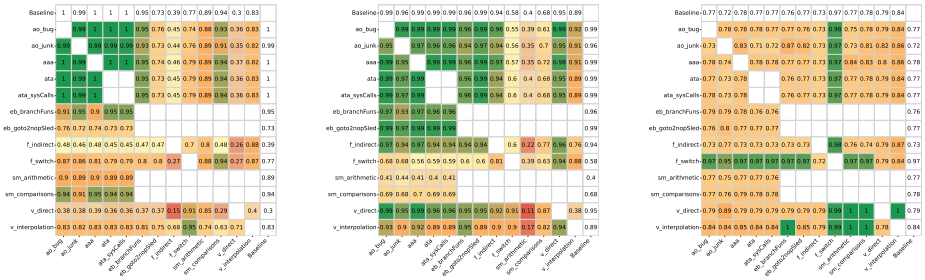
For instruction disassembly, *virtualize binary* yielded an initial increase in F-score under `O1` for Ghidra, Binary Ninja, and IDA Pro, followed by a drop at other optimization levels. At `O1`, the compiler simplified the virtualized dispatcher and removed redundant code, improving relative coverage. At `O2` through `Os`, the compiler relocated the VM tables into static data sections such as `.rodata`, leading to the observed drop in performance. The same pattern appeared for *indirect*, *interpolation*, and *linear*. For function recovery, Binary Ninja performed worse, as the tables in `.rodata`, for example, contained handler entry points as 64-bit values that are interpreted as function starts, producing FPs.

Virtualize direct increased Binary Ninja 's F-score at all optimization levels except `O1`. Without optimization, an indirect jump into a copied blob hides the virtualized code. At higher levels, the compiler restructured this into an explicit dispatcher using switch cases. Although Binary Ninja recovered fewer correct instructions in absolute terms, the reduced code size raised recall.

Virtualize switch caused a reduction in F-score at `O2` and `O3` for IDA Pro's instruction recovery. The optimizer packed the VM dispatch tables into dense static data blocks that IDA Pro no longer recognized as reachable code, resulting in a lowered recall. Similarly to *flatten switch*, the optimized control flow, which selected a jump target through an indirect jump, became increasingly convoluted, thereby degrading jump target detection. In contrast, `Os` compiled the switch statement into a chain of comparison instructions instead of a jump table, which helped Angr map basic blocks to correct functions. This was also the case for jump table recovery.

Takeaways

- Optimization can both help and hinder disassembly;
- Optimization can remove self-modifying code, which increase coverage;
- Optimization introduce indirect jumps, which reduces coverage.



(a) B. Ninja instruction (b) B. Ninja functions (c) B. Ninja jump tables

Fig. 2: Two layered obfuscation F-scores of Binary Ninja for instructions, functions, and jump tables. All scores are relative to the F-score of the first applied obfuscation. We include the plots of all disassemblers in our artifact [21].

6 Obfuscation Layering

In the following, we present how obfuscation layering impacts resilience across disassemblers. As in Section 4, the numbers refer to results without optimization, which enables direct comparison and highlights the effect of the obfuscation itself. In Figure 2, we show the F-score matrix of Binary Ninja. The y-axis lists the first applied transformation, and the x-axis lists the second transformation.

6.1 Instructions

The disassemblers' performance on layered obfuscation aligned with the lower performance of the combined transformations. An exception is when *virtualization* is followed by another transformation. For example, *add opaque junk* reduced the F-score of the instruction recovery of IDA Pro to 0.71. When layered with other obfuscations, the score ranged between 0.32 and 0.85. The score of 0.85 appeared only when the first obfuscation hid the effect of the second, in this case, *add opaque*. Excluding combinations with virtualization, the highest score is 0.76. Lower scores reflect the other baseline. As a consequence, layering can hinder more disassemblers simultaneously. For example, Angr achieved a perfect F-score for *virtualize direct* but struggled with *self-modify arithmetic*. Combining both lowered the score from 0.86 for *self-modify arithmetic* alone to 0.81. IDA Pro and Radare2 declined even stronger, with IDA Pro falling from 0.35 to 0.13 and Radare2 from 0.24 to 0.07. In contrast, Ghidra maintained its score, and Binary Ninja even improved for this combination, although it remains below the baseline. This effect occurred because only a subset of binaries compiles successfully, and the code generated for *self-modify arithmetic* outweighs the impact of *virtualize direct*. As a result, disassemblers that handled *self-modify arithmetic* well also performed well on the combination.

6.2 Functions

The function results mirrored the instruction trends. For example, *add opaque junk* reduced IDA Pro's F-score from 0.9 to 0.87, while other disassemblers remained less affected. In contrast, *self-modify arithmetic* hindered Angr, BAP, Dyninst, and Binary Ninja, but left IDA Pro mostly unchanged. The combination of both transformations increases resilience for all these disassemblers.

6.3 Jump Tables

The jump table results followed the lower scores of layered obfuscations less consistently. As discussed in Section 4, even a single transformation can increase the jump table results. Layered transformations amplified this effect, since combining obfuscations increased the likelihood of generating additional jump tables. OracleGT added jump tables to the ground truth only if the corresponding instruction appeared in the disassembly, which means weak instruction or function recovery can improve jump table scores. The combination of a subset of programs compiling correctly, where no jump tables are counted as part of the ground truth, and no FP are discovered, can lead to outliers in F-scores. For example, for the combination of *virtualize interpolation* and *encode branches branchFuns*, only eight binaries compiled, and Binary Ninja missed all instructions that contained jump tables while also producing no FP, which yielded a perfect score. Despite this partial improvement, we consider layering effective for jump tables, because the increased scores resulted from the small number of jump tables in the unobfuscated baseline rather than from improved disassembler resilience.

Takeaways

- Layering may reduce performance to the weakest obfuscation;
- Some techniques neutralize previously applied transformations;
- Combinations can confuse more disassemblers.

7 Limitation

Binaries in our dataset rely on static libraries from GNU Coreutils, which we did not obfuscate. These were therefore excluded from ground truth detection and analysis to obtain insight into actual code performance. This required minor adjustments to OracleGT to omit instructions, functions, and jump tables from static libraries, limiting comparability with Pang et al. [29, 30], particularly for jump tables. As for disassemblers, we stuck to configurations of Pang et al. Consequently, our results reflected this specific setup, and other configurations may differ.

For studying layering, we applied two Tigress transformations sequentially but abstained from applying the second one to helper functions introduced by the first transformation, potentially reducing its impact. We did not evaluate how many layers can be combined in practice. Additionally, combining two

transformations was not always possible. For example, applying *flatten* after *encode branches* or *self-modify* produced invalid binaries. Additional layers may further restrict valid combinations or degrade runtime performance.

8 Conclusion

We performed a large-scale evaluation of how eight state-of-the-art disassemblers handled obfuscated x86/x64 binaries. Using Tigress and OracleGT, we evaluated their performance across 61 obfuscation transformations, layered configurations, and compiler optimization levels. Our results show that disassembly quality depends on the underlying approach, the applied heuristics, and the obfuscation technique. Linear sweep disassemblers achieved broad instruction coverage since they continued decoding despite misleading control transfers and thus maintained high recall. Their function recovery suffered because they misidentified boundaries or emitted extra entry points. Recursive descent approaches aimed for precision, but this also created blind spots once indirect control flow became non-resolvable. Among them, Ghidra stood out with consistently high recovery, supported by heuristics that compensated for missing control flow edges. Three obfuscation techniques disrupted disassembly the most: *encode branches*, *flatten*, and *self-modify*. Virtualization also proved effective, especially variants that distributed dispatcher logic across multiple blocks. Compiler optimization influenced disassembler performance in positive and negative ways, depending on the obfuscation, optimization level, and disassembly heuristic. For some obfuscations, optimization removed constructs that previously confused disassemblers. In other cases, optimization restructured the program and reduced precision or recall. Layering strengthened protection. In most cases, the combined effect matched the stronger individual technique and widened the set of disassemblers that struggled. Some combinations even reduced performance below both baselines.

Acknowledgments

The financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association is gratefully acknowledged.

References

1. angr, GitHub – Angr, (2019). https://github.com/angr/angr/tree/76da434f
2. Banescu, S., Ochoa, M., Pretschner, A.: A Framework for Measuring Software Obfuscation Resilience against Automated Attacks. In: IEEE/ACM 1st International Workshop on Software Protection (SPRO) (2015). https://doi.org/10.1109/SPRO.2015.16
3. Basque, Z.L., Bajaj, A.P., Gibbs, W., O'Kain, J., Miao, D., Bao, T., Doupé, A., Shoshitaishvili, Y., Wang, R.: Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In: 33rd USENIX Security Symposium (USENIX Security 24) (2024)
4. Bernat, A.R., Miller, B.P.: Anywhere, any-time binary instrumentation. In: Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools (2011)
5. Bin Shamlan, M.H., Alaidaroos, A.S., Bin Merdhah, M.H., Bamatraf, M.A., Zain, A.A.: Experimental Evaluation of the Obfuscation Techniques Against Reverse Engineering. In: Advances on Smart and Soft Computing (2021)
6. Brumley, D., Jager, I., Avgerinos, T., Schwartz, E.J.: BAP: A binary analysis platform. In: International Conference on Computer Aided Verification (2011)
7. Cavallaro, L., Saxena, P., Sekar, R.: On the Limits of Information Flow Techniques for Malware Analysis and Containment. In: Detection of Intrusions and Malware, and Vulnerability Assessment (2008)
8. Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: The effectiveness of source code obfuscation: An experimental assessment. In: IEEE 17th International Conference on Program Comprehension (2009). https://doi.org/10.1109/ICPC.2009.5090041
9. Ceccato, M., Tonella, P., Basile, C., Coppens, B., De Sutter, B., Falcarin, P., Torchiano, M.: How professional hackers understand protected code while performing attack tasks. In: 2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC) (2017)
10. Ceccato, M., Tonella, P., Basile, C., Falcarin, P., Torchiano, M., Coppens, B., De Sutter, B.: Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge. Empirical Software Engineering. **24**(1) (2019). https://doi.org/10.1007/s10664-018-9625-6
11. CMU, CyLab, GitHub – bap, (2020). https://github.com/BinaryAnalysisPlatform/bap/tree/cfeacbfc
12. Collberg, C.: Tigress: The Diversifying C Virtualizer/Obfuscator, version 4.0.11. https://tigress.wtf/
13. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report 148, Department of Computer Science, University of Auckland (1997)
14. Crussell, J.: Ghidra: Is Newer Always Better? In: Proceedings of the Workshop on Binary Analysis Research (BAR) (2025). https://doi.org/10.14722/bar.2025.23003
15. De Sutter, B., Schrittwieser, S., Coppens, B., Kochberger, P.: Evaluation Methodologies in Software Protection Research. ACM Computing Surveys **57**(4) (2024). https://doi.org/10.1145/3702314
16. Disassembly Precion and Recall Table O0, https://anonymous.4open.science/r/ObfuscationGT/tables_and_figures/precision_recall_O0.csv

17. Free Software Foundation, Inc, Coreutils - GNU core utilities, version 9.7. https://www.gnu.org/software/coreutils/
18. GNU, Index of /gnu/binutils, (2020). https://ftp.gnu.org/gnu/binutils/
19. GNU Compiler Collection, GCC = Optimization Levels (GNAT User's Guide for Native Platforms), https://gcc.gnu.org/onlinedocs/gnat_ugn/Optimization-Levels.html
20. Hänsch, N., Schankin, A., Protsenko, M., Freiling, F., Benenson, Z.: Programming Experience Might Not Help in Comprehending Obfuscated Source Code Efficiently. In: Proceedings of the 14th Symposium On Usable Privacy and Security (SOUPS) (2018)
21. Heatmaps Layering, https://anonymous.4open.science/r/ObfuscationGT/tables_and_figures/heatmap_combination
22. Hex-Rays, IDA Pro: Powerful Disassembler, Decompiler & Debugger, version 9.1. https://hex-rays.com/ida-pro/
23. Jiang, M., Zhou, Y., Luo, X., Wang, R., Liu, Y., Ren, K.: An Empirical Study on ARM Disassembly Tools. In: Proceedings of the 29th International Symposium on Software Testing and Analysis (ISSTA) (2020). https://doi.org/10.1145/3395363.3397377
24. Kruegel, C., Robertson, W., Valeur, F., Vigna, G.: Static disassembly of obfuscated binaries. In: USENIX security Symposium (2004)
25. László, T., Kiss, Á.: Obfuscating C++ Programs via Control Flow Flattening. Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica **30**(1) (2009)
26. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: Proceedings of the 10th ACM conference on Computer and communications security (2003)
27. Liu, B., Shen, J., Ming, J., Zheng, Q., Li, J., Xu, D.: MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In: Proceedings of the 30th USENIX Security Symposium (USENIX Security) (2021)
28. NSA, GitHub – Ghidra, (2019). https://github.com/NationalSecurityAgency/ghidra/tree/Ghidra%5C_9.0.4%5C_build
29. Pang, C., Yu, R., Chen, Y., Koskinen, E., Portokalidis, G., Mao, B., Xu, J.: SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In: Proceedings of the 42nd IEEE Symposium on Security & Privacy (S&P) (2021)
30. Pang, C., Zhang, T., Yu, R., Mao, B., Xu, J.: Ground Truth for Binary Disassembly is Not Easy. In: Proceedings of the 31st USENIX Security Symposium (USENIX Security) (2022)
31. Paradyn, GitHub – Dyninst, (2019). https://github.com/dyninst/dyninst/tree/5d2ddacb
32. radareorg, GitHub – Radare2, (2020)
33. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting Software through Obfuscation: Can It Keep Pace with Progress in Code Analysis? ACM Computing Surveys **49**(1) (2016). https://doi.org/10.1145/2886012
34. Sebastian, B., Christian, C., Alexander, P.: Predicting the Resilience of Obfuscated Code Against Symbolic Execution Attacks via Machine Learning. In: Proceedings of the 26th USENIX Security Symposium (USENIX Security 17), Vancouver, BC, Canada (2017)

35. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: Proceedings of the 37th IEEE Symposium on Security & Privacy (S&P) (2016)
36. Taylor, C., Colberg, C.: A tool for teaching reverse engineering. In: 2016 USENIX Workshop on Advances in Security Education (ASE 16) (2016)
37. Vector 35, Binary Ninja, version 5.0.7648. https://binary.ninja/
38. Viticchié, A., Regano, L., Torchiano, M., Basile, C., Ceccato, M., Tonella, P., Tiella, R.: Assessment of Source Code Obfuscation Techniques. In: IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM) (2016). https://doi.org/10.1109/SCAM.2016.17
39. Wijayadi, L.A., Jiang, Y., Yap, R.H., Liang, Z., Liu, Z.: Evaluating Disassembly Errors With Only Binaries. In: Proceedings of the 20th ACM ASIA Conference on Computer and Communications Security (ASIACCS) (2025). https://doi.org/10.1145/3708821.3733884
40. Yadegari, B., Debray, S.: Symbolic execution of obfuscated code. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (2015)
41. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A Generic Approach to Automatic Deobfuscation of Executable Code. In: 2015 IEEE Symposium on Security and Privacy (SP) (2015). https://doi.org/10.1109/SP.2015.47

A Performance Metrics

To evaluate disassembler performance, we calculated the success rate and the F-score. OracleGT reports the number of True Positive (TP), True Negative (TN), FP, and FN for functions, instructions, and jump tables. We used these values to compute precision, recall, and the F-score.

Precision measures the proportion of TP instances among all identified positives:

$$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{1}$$

Recall measures the proportion of TP instances among all actual positives, incorporating FN:

$$Recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}} \tag{2}$$

For each disassembler and obfuscation configuration, we compute the mean precision and recall across all files and derive the F-score as their mean:

$$F_1 = \frac{2}{recall^{-1} + precision^{-1}} \tag{3}$$

The F-score balances precision, the ability to avoid FP, and to capture all TP. It is particularly suitable for disassembly tasks where FP and FN occur with different frequencies. The score accounts for both types of errors and captures how well the disassembler balances precision and recall.

B Obfuscation Impact

Table 3: Average precision and recall per disassembler for `O0` optimization level on unobfuscated binaries. Columns labeled *Our paper* report our results, while *OracleGT* shows the results reported by Pang et al. [30].

		Our paper		OracleGT	
	Type	**Prec**	**Rec**	**Prec**	**Rec**
Angr	Instructions	99.85	99.95	99.99	99.97
	Functions	63.11	95.66	91.44	98.29
	JumpTable	23.82	39.86	99.80	87.34
BAP	Instructions	99.97	95.57	99.99	82.50
	Functions	89.52	85.95	96.36	85.55
	JumpTable	–	–	–	–
Dyninst	Instructions	99.78	69.67	99.99	99.07
	Functions	98.45	69.97	99.65	97.11
	JumpTable	39.30	52.70	99.97	98.82
Ghidra	Instructions	99.86	99.96	99.99	99.65
	Functions	98.88	100	99.94	99.66
	JumpTable	76.13	76.58	98.49	75.58
IDA Pro	Instructions	99.86	99.94	99.99	99.99
	Functions	99.25	95.66	99.47	92.64
	JumpTable	77.21	81.08	99.95	99.99
Binary Ninja	Instructions	99.86	99.86	99.99	99.81
	Functions	98.49	99.96	97.24	99.78
	JumpTable	76.58	79.05	99.60	99.30
Objdump	Instructions	99.78	100	99.98	99.99
	Functions	–	–	–	–
	JumpTable	–	–	–	–
Radare2	Instructions	99.84	71.24	99.98	87.92
	Functions	99.18	90.55	98.71	97.01
	JumpTable	76.70	79.73	55.74	43.14