Polaris: MIR-Level Obfuscation in LLVM for Efficient and Robust Decompiler Resistance

Ang Zhou*
r1mao233@bupt.edu.cn
Beijing University of Posts and Telecommunications
Beijing, China

Jiongchi Yu*
jcyu.2022@phdcs.smu.edu.sg
Singapore Management University
Singapore

Yiran Zhang†
yiran002@e.ntu.edu.sg
Nanyang Technological University
Singapore

Ziming Zhao†
zhaoziming@zju.edu.cn
Zhejiang University
Zhejiang, China

Zhaoxuan Li
lizhaoxuan@iie.ac.cn
State Key Laboratory of Cyberspace Security Defense, IIE, CAS
Beijing, China

Tingting Li
litt2020@zju.edu.cn
Zhejiang University
Zhejiang, China

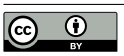
Abstract

Obfuscation techniques are widely used to hinder program analysis and reverse engineering in order to protect software security and intellectual property. High-level software obfuscation techniques (e.g., source-code- or LLVM IR-based) often leave recognizable artifacts that modern decompilers can effectively exploit, thereby limiting their long-term effectiveness. In contrast, binary-level approaches, although effective, typically incur substantial performance overheads or rely on packing, while still exhibiting structural patterns that remain amenable to automated analysis.

LLVM Machine IR (MIR), which represents programs after instruction selection in the LLVM backend, enables target-aware and fine-grained transformations before final code emission without violating compiler correctness guarantees. In this work, we present *Polaris*, an LLVM-backed MIR-level obfuscation framework that applies a set of composable MIR transformations, including function-boundary perturbation, control-flow restructuring, and data-flow noise injection, to invalidate assumptions made by mainstream reverse-engineering tools and substantially increase their analysis effort, while preserving program semantics and functionality.

We demonstrate *Polaris* on 100 real-world programs comprising 4,024 functions and evaluate its impact using three mainstream decompilers. The results show that *Polaris* significantly degrades function recovery and pseudocode quality, increases fragmentation and analysis timeouts, and incurs only practical runtime overheads. We publicly release the complete implementation of *Polaris* at https://github.com/za233/Polaris-Obfuscator, and provide an accompanying demonstration video at https://youtu.be/eALCDiEqJJw.

*Both authors contributed equally to this research.
†Corresponding authors.



ICSE-Companion '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2296-7/2026/04
https://doi.org/10.1145/3774748.3787619

CCS Concepts

• **Software and its engineering → Compilers**; • **Security and privacy → Software security engineering**.

Keywords

Binary Obfuscation, LLVM, Machine IR (MIR), Decompilers, Reverse Engineering, Program Analysis

ACM Reference Format:
Ang Zhou, Jiongchi Yu, Yiran Zhang, Ziming Zhao, Zhaoxuan Li, Tingting Li. 2026. Polaris: MIR-Level Obfuscation in LLVM for Efficient and Robust Decompiler Resistance. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion '26), April 12–18, 2026, Rio de Janeiro, Brazil*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3774748.3787619

1 Introduction

Reverse engineering (RE) [1] is a common prerequisite for software cracking and unauthorized exploit development, motivating extensive research on obfuscation techniques to protect software intellectual property [2]. While prior work has largely focused on source-code or intermediate-representation (IR)-level transformations, modern RE pipelines increasingly rely on decompilers that perform robust mid-level analyses to recover functions, variables, and control-flow structures from binaries. Consequently, many transformations that appear complex at the IR level remain easy to recover during decompilation.

At the opposite end, instruction-level obfuscation techniques, such as virtualization-based protection and packing, can substantially hinder reverse engineering but often incur significant build-time and runtime overhead. Moreover, to preserve correctness, these system-level approaches typically apply constrained transformation patterns, which may still leave recognizable artifacts exploitable by advanced decompilers.

LLVM Machine IR (MIR) offers a practical middle ground. Located after instruction selection in the LLVM back end, MIR preserves target-specific semantics while maintaining compiler correctness guarantees. This positioning enables fine-grained, target-aware transformations that directly shape low-level artifacts-such

as instruction streams and control transfers-analyzed by decompilers, while remaining largely invisible at higher IR levels.

Motivated by this opportunity, we present *Polaris*, a practical obfuscation framework that operates on LLVM [3] MIR in the compiler back end, between instruction selection and code emission. By working at the MIR level, *Polaris* preserves compiler correctness guarantees, calling conventions, and register allocation, while effectively disrupting structural assumptions leveraged by mainstream decompilers. *Polaris* applies a set of composable transformations that span both control-flow and data-flow obfuscation.

We evaluate *Polaris* on 100 real-world programs using three mainstream decompilers, including *IDA* [4], *Ghidra* [5], and *Binary Ninja* [6]. The results show that *Polaris* substantially degrades decompilation quality by reducing correctly recovered functions, increasing function fragmentation, impairing pseudocode readability, and triggering analysis slowdowns, while preserving program correctness and maintaining practical build-time and runtime overheads. Our implementation of *Polaris* is publicly released at https://github.com/za233/Polaris-Obfuscator, and the demonstration video is available at https://youtu.be/eALCDiEqJJw.

2 Background

2.1 Software Decompilation

Modern decompilers typically follow a three-stage pipeline [7, 8] . The *front-end* parses binary formats (e.g., ELF/PE/Mach-O) and lifts raw bytes into instruction sequences, constructing basic blocks and an initial control-flow graph. The *mid-end* operates on an architecture-independent intermediate representation to recover high-level constructs, as it identifies function boundaries, infers variables and types from memory and data-flow patterns, and performs control- and data-flow analyses along with IR simplifications. The *back-end* then translates the recovered representation into C-like pseudocode by structuring control flow and reducing low-level artifacts such as explicit jumps. Together, these stages aim to reconstruct readable program semantics from optimized binaries.

2.2 Software Obfuscation

Software obfuscation aims to transform programs to make them harder to analyze or reverse engineer while preserving their original semantics [2]. Obfuscators can operate at different abstraction levels [9] . Source/IR-level obfuscation (e.g., LLVM IR passes such as flattening [10], bogus edges [2], instruction substitution [11], and mixed Boolean arithmetic [12]) typically introduces transformations that remain sufficiently regular for decompilers' mid-end analyses to normalize, reidentify functions, recover types, and simplify injected noise. In contrast, binary-level techniques (e.g., junk insertion and dirty bytes insertion [13]) work directly on machine code and are toolchain independent, but they lack compilation context, such as types and register allocation, are often brittle, and may still expose recognizable artifacts that can be exploited.

3 Case Study

OLLVM. Source- and IR-level obfuscators, such as OLLVM [11], apply structural transformations during compilation to impede static analysis. Typical techniques include control-flow flattening (CFF), bogus control-flow insertion, and instruction substitution, all

```
while ( v4 != 876376421 ) {
  if ( v4 == 912523791 ) {
    v4 = 876376421;
    if ( v5 < 32 )
      v4 = 1263910219;
  } else if ( v4 == 1263910219 ) {
    v7 -= (a2[2] + (v8 » 5)) ^ (v6 + v8)
        ^ (a2[3] + 16 * v8);
    v8 -= (a2[0] + (v7 » 5)) ^ (v6 + v7)
        ^ (a2[1] + 16 * v7);
    v6 -= 0x9e3779b9;
    v4 = 2011508661;
  } else {
    ++v5;
    v4 = 912523791;
  }
}
```

Listing 1: Pseudocode after control-flow flattening (TEA).

```
for ( i = 0; i < 32; ++i ) {
  v5 -= (a2[3] + (v6 » 5)) ^ (v4 + v6)
      ^ (a2[2] + 16 * v6);
  v6 -= (a2[1] + (v5 » 5)) ^ (v4 + v5)
      ^ (*a2 + 16 * v5);
  v4 -= 0x9e3779b9;
}
```

Listing 2: Pseudocode after automated de-flattening.

performed before code generation. Listing 1 illustrates IDA Pro [4]'s decompiled output of a TEA routine after CFF. It replaces the original control-flow graph (CFG) with a dispatcher loop that routes execution among basic blocks via a dedicated state variable (e.g., v4). Although this transformation reduces human readability, it often preserves semantic cues exploitable by modern decompilers, such as characteristic constants (e.g., 0x9e3779b9) and bitwise patterns remain visible, allowing functionality to be reliably recovered.

Deobfuscation frameworks that operate on decompiler IR can often normalize such patterns. For instance, D810 [14] removes control-flow flattening by restructuring the decompiler's optimized IR, producing pseudocode that closely resembles the original program (Listing 2). This demonstrates that IR-level obfuscation can be systematically reversed once mid-level analyses succeed.

VMProtect. *VMProtect* [15] is a commercial obfuscator known for packing and virtualization, and also supports binary-level junk insertion. Due to the high overhead of virtualization, we focus on register-only junk insertion, which interleaves semantically neutral instructions to hinder decompilation. Lacking compiler metadata, such binary-only approaches typically avoid memory-access junk and rely on register shuffling, which can often be removed once data-flow information is recovered.

```
0x17c5: mov  rax, qword ptr [r11]
0x17c8: movsx ecx, r10w
0x17cc: xchg cl, bl  ; killed by 0x17d1
0x17ce: setl bh   ; killed by 0x17d5
0x17d1: mov  rcx, qword ptr [r11 + 8]
0x17d5: mov  rbx, 0x1234
```

Listing 3: Register-only junk instructions within basic block.

In our study, we simulate execution of binaries protected by VMProtect 3.5.1 using *angr* [16] and derive a CFG from instruction traces. A reaching-defs analysis over this CFG removes a large fraction of register-manipulating dead code: **47.08%** of original instructions are identified as redundant and eliminated. As highlighted in Listing 3, red instructions are overwritten by subsequent definitions, illustrating why purely pattern-based, register-only junk is vulnerable to mid-end normalization.

Takeaway: These case studies motivate *Polaris*'s design: IR-level patterns can be normalized by decompiler-centered workflows, while post-hoc binary tricks are often brittle. *Polaris* instead operates at MIR in the back-end, where it can perturb function recognition and mid-end assumptions while retaining compiler guarantees.

4 Design

4.1 Overview

Polaris operates as an LLVM back-end passes on MIR, transforming programs before code emission to disrupt decompiler recovery of functions, control flow, and data flow while preserving program semantics and compiler correctness.

4.2 Defeat Function Recognition

In stripped binaries, procedure metadata is unavailable, and decompilers infer function boundaries using instruction-level heuristics, e.g., treating `call` as an entry, `ret` as an exit, and `jmp` as intra-procedural control transfer. Polaris disrupts these assumptions at the MIR level by rewriting control-transfer cues while preserving the original runtime control flow. Specifically, Polaris randomly splits selected basic blocks and probabilistically replaces eligible `jmp` instructions with semantically equivalent `call` sequences followed by a balancing `pop`, subject to liveness and ABI checks. To prevent decompilers from folding the injected `call`/`pop` back into canonical control flow, Polaris inserts a never-executed fake basic block containing well-formed but atypical instruction patterns (e.g., early `ret`, off-pattern transfers, benign calls, or non-prologue `push`/`pop`), enabled by the fine-grained control and compilation context available only at the MIR level.

4.3 Defeating Data-Flow Analysis.

Decompilers rely heavily on accurate data-flow information to recover variables and types and to simplify intermediate representations. Polaris targets data-flow analysis at the MIR level by injecting context-aware junk instructions and introducing indirect stack-based memory accesses, while preserving program semantics. Specifically, *Polaris* inserts single-instruction junk operations synthesized from a restricted opcode set, with operands bound using compiler metadata such as frame layout, calling conventions, liveness, and aliasing information, which reads preferentially use already-entangled registers, writes target non-live registers, stack reads are confined to known-safe regions, and stack writes allocate fresh frame slots. A per-register entanglement map biases subsequent insertions, forming plausible use-def chains that resist trivial peephole elimination without affecting program outputs. Although the effective address remains unchanged, data dependencies are redirected through arithmetic on opaque values, degrading alias analysis and variable mapping. Together, these transformations resist immediate simplification, pollute reaching definitions and memory SSA, and complicate variable and type recovery, while remaining semantics preserving by construction.

4.4 Defeating Control-Flow Analysis

Polaris introduces lightweight bogus control flow using opaque predicates over an internal state to disrupt static control-flow reasoning without affecting runtime behavior. Each basic block updates a hidden variable X with a block-specific constant chosen to keep X path independent at block entry, enabling predicates that are always true at runtime yet difficult for decompilers to simplify after lowering. To avoid explicit cycle analysis, *Polaris* splits blocks into short linear chains, assigns random constants to intermediate blocks, and balances the final update. Bogus branches guarded by predicates such as $(X{=}S)$ or $(X \oplus K){=}(S \oplus K)$ lead to small unreachable blocks. Although execution always follows the original path with minimal overhead, the interleaved state updates and predicate arithmetic impede mid-end control-flow structuring and region formation while preserving semantics.

4.5 Defeating Variable Recovery

Decompilers recover stack variables via stack-pointer (SP) analysis by tracking per-block SP deltas and normalizing stack accesses at control-flow joins; inconsistent deltas collapse to $\top$, preventing precise frame reconstruction. *Polaris* exploits this behavior at the MIR level by skewing abstract SP deltas while preserving concrete memory accesses. For selected blocks, *Polaris* temporarily rebases the stack pointer by a block-specific offset Δ_B, rewrites stack accesses accordingly, and restores the original SP value, with Δ_B chosen under strict liveness and frame-safety checks. Although executed addresses remain unchanged, predecessors at joins now carry distinct abstract deltas, which-when combined with bogus control-flow edges-force $\top$ propagation, destabilize reaching definitions over frame slots, and degrade variable recovery and pseudocode quality, while remaining ABI compliant and semantics preserving.

5 Evaluation

5.1 Experiment Setup

All experiments are conducted on a Windows x64 host equipped with an Intel i7-13700 CPU and 32 GB of RAM, running an Ubuntu 20.04 virtual machine with 16 GB of memory. The evaluated programs are generated using *Csmith* [17], and the resulting binaries are analyzed using three mainstream decompilers, specifically for IDA, Ghidra, and Binary Ninja.

5.2 Effectiveness Evaluation

Structure-Oriented Obfuscation. We evaluate *Polaris* under `-O0` by focusing on structure-oriented obfuscation targeting function recognition. Since function splitting yields misleading pseudocode fragments, we measure the number and average size of functions recovered across 100 programs. As shown in Table 1, without obfuscation, all decompilers correctly recover the ground truth (4,024

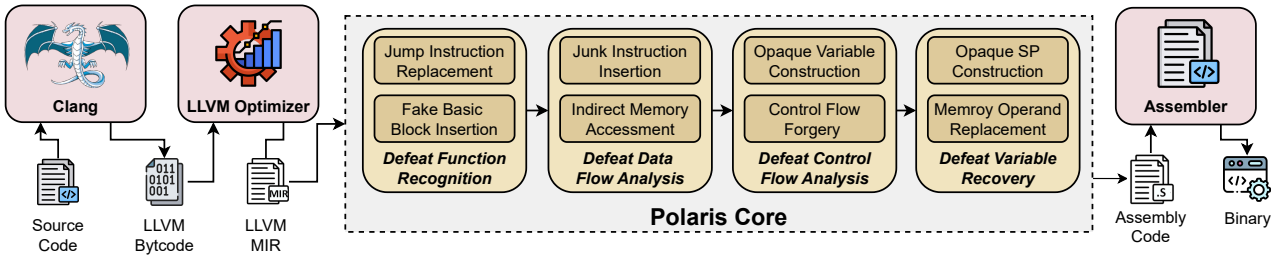


Figure 1: The overall workflow of Polaris.

Table 1: Performance summary of evaluated decompilers.

Tool	Method	Succ	Func	AF Size	AC Size	CSF	Time	TCF	Err
IDA	*Polaris*	100	3994	8705	39070	4.49	264.14	30.34	9.39
	OLLVM	100	3994	8832	13669	1.55	3.55	0.40	0.00
	Unobf	100	3991	702	1577	2.25	0.04	0.05	2.51
Ghidra	*Polaris*	100	12399	2327	132882	57.11	8.62	3.70	42.33
	OLLVM	100	3994	5076	7071	1.39	1.27	0.25	0.00
	Unobf	100	3991	702	1742	2.48	0.31	0.44	0.00
Binja	*Polaris*	75	2622	7205	52092	7.23	366.88	50.92	63.88
	OLLVM	100	3994	8832	77774	8.81	0.54	0.06	83.73
	Unobf	100	3991	702	1846	2.63	0.01	0.02	0.00

functions, 2.8 MB). With *Polaris*, the number of recognized functions increases by one to two orders of magnitude (e.g., 67K in IDA, 82K in Ghidra, and 210K in Binary Ninja), while average function size drops to 4–7% of the baseline. This severe fragmentation disrupts boundary reconstruction, causing missed procedures in IDA and excessive spurious functions in Binary Ninja.

Semantics-Oriented Obfuscation. We further assess the data-flow, control-flow, and stack-pointer defenses on the same 100 programs. We examine the quality of recovered pseudocode in terms of function recovery, code expansion, and analysis time. *Polaris* consistently reduces successful recovery while increasing pseudocode size and decompilation time relative to both unprotected binaries and *OLLVM*, indicating substantial degradation of semantic reconstruction beyond simple code expansion.

5.3 Efficiency Evaluation

For 100 programs, we measured average size and time for *Baseline*, *Polaris*, and *OLLVM* at O0/O1/O2 (timeouts excluded). Table 2 shows increases for both metrics, mitigated by optimization. At O0, *Polaris* averages ~421,647 B (≈ 5.95× baseline) and 10.99 ms (≈ 2.15×); OLLVM averages ~342,510 B (≈ 4.84×) and 6.73 ms (≈ 1.32×). Higher levels reduce costs, more for Baseline/OLLVM than *Polaris*. Overall, *Polaris* yields stronger obfuscation signals with modest runtime overhead versus virtualization-style approaches; OLLVM offers a lighter trade-off. Both are practical for offline builds.

6 Conclusions

Polaris is a composable, MIR-level obfuscation framework integrated into the LLVM backend that disrupts function boundaries, data/control flow, and stack-frame recovery while preserving program semantics and ABI guarantees. Our evaluation shows it scales across large codebases with acceptable overhead and composes safely with existing tools, substantially degrading modern decompilers' mid-end analyses. These results highlight the sweet spot between robustness and efficiency of the MIR perspective. Future work will broaden architecture support, introduce adaptive tuning, and incorporate dynamic-resistance and automated benchmarking.

Table 2: Result of efficiency evaluation.

Method	Opt.	Avg. Size (bytes)	Avg. Time (ms)	Count
No Obfuscation	-O0	70,815.57 (1.00x)	5.11 (1.00x)	93
	-O1	30,342.45 (1.00x)	4.75 (1.00x)	
	-O2	27,532.22 (1.00x)	4.48 (1.00x)	
Polaris	-O0	421,647.10 (5.95x)	10.99 (2.15x)	89
	-O1	112,511.28 (3.71x)	6.92 (1.46x)	
	-O2	93,334.11 (3.39x)	6.62 (1.48x)	
OLLVM	-O0	342,510.46 (4.84x)	6.73 (1.32x)	90
	-O1	90,271.47 (2.98x)	5.16 (1.09x)	
	-O2	79,115.07 (2.87x)	4.47 (1.00x)	

References

[1] Daniel Votipka et al. An observational investigation of reverse Engineers' processes. In *USENIX Security*. USENIX Association, August 2020.
[2] Christian Collberg et al. A taxonomy of obfuscating transformations, 1997.
[3] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization*. IEEE.
[4] Hex-Rays SA. Ida pro: Interactive disassembler. [EB/OL]. https://hex-rays.com/ida-pro/ Accessed September 27, 2025.
[5] National Security Agency. Ghidra software reverse engineering framework. [EB/OL]. https://ghidra-sre.org/ Accessed September 27, 2025.
[6] Vector 35 Inc. Binary ninja: A reverse engineering platform. [EB/OL]. https://binary.ninja/ Accessed September 27, 2025.
[7] Zhibo Liu and Shuai Wang. How far we have come: Testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487, 2020.
[8] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software—Practice & Experience*, 25(7):811–829, 1995.
[9] Sampsa Rauti Shohreh Hosseinzadeh et al. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 104:72–93, 2018.
[10] Tímea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. volume 30, 06 2007.
[11] Pascal Junod et al. Obfuscator-llvm – software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*.
[12] Yongxin Zhou et al. Information hiding in software with mixed boolean-arithmetic transforms. In *Information Security Applications*, pages 61–75, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
[13] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS*, 2003.
[14] IDAPluginProject. D-810: An ida pro plugin for deobfuscation. [EB/OL]. ttps://github.com/IDAPluginProject/d810 Accessed September 27, 2025.
[15] VMProtect Software. Vmprotect version 3.8.7. [EB/OL]. https://vmpsoft.com/ Accessed September 27, 2025.
[16] Fish Wang et al. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, 2017.
[17] Xuejun Yang et al. Finding and understanding bugs in c compilers. In *PLDI*, 2011.