Stop Paying for Whitespace: Token-Efficient Decompiler Output for LLM-Assisted Binary Analysis

Steffen Enders
Fraunhofer FKIE
Bonn, Germany
steffen.enders@fkie.fraunhofer.de

Eva-Maria C. Behner
Fraunhofer FKIE
Bonn, Germany
eva-maria.behner@fkie.fraunhofer.de

Elmar Padilla
Fraunhofer FKIE
Bonn, Germany
elmar.padilla@fkie.fraunhofer.de

Abstract

Large language models are now standard in binary reverse-engineering pipelines, consuming decompiler output to drive vulnerability detection, malware triage, and function summarization. At scale, this costs thousands of dollars and billions of tokens. Yet this output is written for human analysts, not language models. It therefore carries presentation formatting and decompiler bookkeeping, such as layout whitespace, placeholder names, and speculative casts, that inflate the token count without adding usable information. Because compilation discards the developer's names, types, and structure, much of this text is the decompiler's own synthesis rather than recovered signal, so it can be compressed more aggressively than authored source code.

We introduce Deflate-D, a deterministic, model-free reformatter that reduces this overhead through four cumulative tiers ordered by information loss. Two semantics-preserving tiers normalize cosmetic layout and apply structural rewrites, a third compresses machine-generated names and types while preserving meaning, and a fourth removes decompiler boilerplate and other low-value detail. Across 109 GNU coreutils binaries, the semantics-preserving tiers alone reduce input tokens by roughly a fifth to a third, and the lossy tiers reach up to 61%. To our knowledge, Deflate-D is the first to treat decompiler output as a compression target in its own right. On three function-level downstream tasks with a frontier model, these savings leave task quality largely intact. We release Deflate-D and our benchmark for reproducible, cost- and latency-aware binary analysis.

CCS Concepts

• **Software and its engineering** → **Software notations and tools**; • **Computing methodologies** → *Artificial intelligence.*

Keywords

decompilation, binary analysis, large language models, reverse engineering, token efficiency, prompt compression

ACM Reference Format:
Steffen Enders, Eva-Maria C. Behner, and Elmar Padilla. 2026. Stop Paying for Whitespace: Token-Efficient Decompiler Output for LLM-Assisted Binary Analysis. In *2nd Workshop on Software Understanding and Reverse Engineering (SURE '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3847158.3847384



This work is licensed under a Creative Commons Attribution 4.0 International License.
SURE '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3019-1/2026/11
https://doi.org/10.1145/3847158.3847384

```
1 void FUN_0010b1a0(void)
2 {
3   DAT_00124988 = FUN_0010d530(0x67,0,FUN_0010b050,F
↪ UN_0010b060,FUN_0010b080);
4   if (DAT_00124988 != 0) {
5   return;
6   }
7   /* WARNING: Subroutine does not return */
8   FUN_00113270();
9 }
```

T0 *raw* 89 tokens

Figure 1: A stripped `ginstall` helper (Ghidra, RQ1 corpus) as seen by the GPT tokenizer: each colored box is one token. Soft-wrapped lines are marked with ↪.

1 Introduction

Reverse engineers increasingly rely on large language models to make sense of compiled binaries, now embedded throughout many binary-analysis pipelines. Across vulnerability detection, malware triage, and function summarization, their dominant input is *decompiler output*, such as the C-like pseudocode emitted by Ghidra, IDA Pro (Hex-Rays), and Binary Ninja [13, 20, 24, 47]. However, at scale, feeding this pseudocode to a model is expensive. A single benchmarking study consumed thousands of dollars and billions of tokens passing decompiled functions to LLMs [24], and agentic reverse-engineering systems repeatedly flag limited token budgets as a core challenge [41]. Even as context windows grow, fewer tokens per function directly lower cost, speed up responses, and leave room to fit more of a binary into a single context. The problem is especially acute for the reverse-engineering MCP servers that integrate decompilers into LLM agents and forward pseudocode to the model essentially unchanged. Practitioners report that a single such prompt can consume hundreds of thousands of tokens, enough to exceed a local model's context window and truncate the input, degrading the output [52]. Reducing the tokens of functions therefore reduces price, latency, and context pressure at once, and shorter inputs can even *improve* recall on downstream tasks [44].

This cost comes not from the program itself, but from the way a model reads its text. Specifically, a language model does not consume characters or C tokens but a sequence of *tokens* drawn from a fixed byte-pair vocabulary of subword fragments. Both API billing and the context-window limit are counted in these tokens. The strings that dominate the decompiler output are the ones this encoding handles worst. Figure 1 shows a placeholder identifier such as `FUN_0010b1a0` split into eight tokens because no fragment

matches an arbitrary address, each indentation run taking a token of its own, and a single banner `/* WARNING */` tokenized in full despite carrying no program logic. Different model families also tokenize the same text differently, since each ships its own byte-pair vocabulary, so token counts vary between encoders [30].

Importantly, a large portion of this token cost is avoidable, because decompiler output is mainly written for a human analyst rather than for the model, so it carries presentation and formatting that the model does not need. Authored source code holds little beyond such formatting that can be cut safely, since its names, comments, types, and structure all carry developer intent [39]. In contrast, decompiler pseudocode carries little such intent, since compilation discards the developer's names, comments, types, and structure. Instead, the decompiler emits largely its own synthesis, rendered from the internal representation it recovers, namely a control-flow graph with inferred types and data-flow facts. Every gap that representation leaves is filled with a placeholder name or a speculative type unrelated to the original source or its semantics, alongside low-level artifacts the renderer adds for manual analysis.

Compressing decompiler output can therefore go further than reducing authored source, as long as the genuine signal is preserved. For example, default identifiers, such as `local_18` and `uVar3`, are *mechanically generated* from stack offsets and SSA indices, "nothing more than arbitrary placeholders such as `VAR1` and `VAR2`" [26]. Name and type recovery research on binaries confirms how little survives compilation, since even dedicated recovery models reconstruct the original developer name only ∼56–66% of the time [8, 55], and recovered types are likewise speculative [28, 37]. The genuine signal lies instead in the program's logic, since its control and data flow are reconstructed directly from the binary and carry the behavior an analysis actually reasons over. Thus, names and types are the weaker part, and even there genuine signal comes only from those anchored to a real *symbol*, whose removal causes large semantic loss [24]. Every major decompiler already marks this distinction with a confidence or source-type signal [14, 16, 50].

Consequently, a reformatter for decompiler output can leave the control and data flow and the symbol-anchored names and types intact while compressing only the low-confidence, mechanically generated material. Prior token-reduction work, however, targets neither decompiler output nor this asymmetry. Trained prompt compressors drop tokens from arbitrary text [21, 40], and a parallel line reformats authored source code, where little beyond formatting can be removed safely [39, 45, 46, 53]. Others select which decompiled functions reach the model through slicing [29, 33] or enrich the output rather than reduce it [19], leaving the rendering each function costs untouched. The closest work, source-code formatting removal [39], strips only cosmetic layout, leaving the structural rewrites and decompiler-aware lossy compression untouched.

The wasted tokens fall into distinct categories, namely layout, redundant structure, synthesized names and types, and low-confidence annotations, so a single blunt minifier is the wrong instrument. We introduce Deflate-D, a deterministic, model-free reformatter that applies this principle through four cumulative tiers, forming a taxonomy of token-reduction transformations ordered by information loss. The first two tiers are semantics-preserving, normalizing cosmetic formatting, then eliminating structural redundancy the decompiler emits for readability, not meaning. The latter two are lossy, compressing or removing machine-generated bookkeeping and low-confidence boilerplate while preserving symbol-anchored content. Section 2 specifies each tier.

We evaluate Deflate-D on 109 GNU coreutils binaries decompiled with three mainstream decompilers, with savings measured on the GPT, Claude, and Gemini tokenizers. The semantics-preserving tiers alone cut a fifth to a third of input tokens, and the lossy tiers reach 45–61%, all at near-zero measurement cost because token counting is decoupled from generation. Next, using downstream-quality check across three published binary-analysis tasks, function naming, variable naming, and function summarization, we show that these savings largely preserve task quality. However, because these tasks are short, function-level, and use a single frontier model, this evaluation bounds the quality cost of compression only for that setting. It does not certify the lossy tiers safe for tasks with different demands, such as vulnerability analysis or malware triage.

To our knowledge, this is the first approach to treat decompiler output as a token-compression target of its own, characterizing token-reduction transformations by their information loss and confining the lossy ones to material the decompiler synthesized. The asymmetry between synthesized boilerplate and recovered signal is what uniquely enables this framing. We are also the first to evaluate whether such compression preserves quality on downstream binary-analysis tasks, rather than assuming the savings are free. Overall, we make the following contributions:

- **I**: A **four-tier taxonomy** of token reduction for decompiler output, ordered by information loss, comprising the semantics-preserving cosmetic and structural tiers and the lossy contextual and reductive tiers that discard machine-generated artifacts and low-confidence signal, respectively.
- **II**: **Deflate-D**, an implementation of the taxonomy for the three most widely used decompilers, Ghidra, IDA Pro, and Binary Ninja, targeting the lossy tiers at machine-generated material identified by its spelling.
- **III**: A **token-savings evaluation** measuring reduction at each tier, per decompiler, across multiple tokenizers at near-zero measurement cost.
- **IV**: A **downstream-quality check** that reuses the datasets from three published binary-analysis evaluations, function naming, variable naming, and function summarization, measuring whether the token reduction affects task quality.

2 Deflate-D: Four-Tier Token Reduction

Decompiler output is verbose, and a language model pays for that verbosity in tokens, both in API cost and in consumed context. Deflate-D is a preprocessing stage that sits between a decompiler and the LLM-assisted binary-analysis pipeline it feeds, shrinking the decompiled code so that fewer tokens reach the model while preserving the information a downstream task relies on. Because it is a deterministic, model-free transformation on the decompiler's textual output, it never invokes a model, alters the prompt, or modifies the decompiler. It therefore drops in front of any approach that feeds decompiler output to an LLM, from a single function-summarization prompt to an agentic reverse-engineering loop, and applies to the output of any major decompiler.

```
__int64 __fastcall sub_3030(__int64 a1, char a2,
__int64 a3)\n
{\n
  __int64 result; // rax
  const void *v5; // rdi
  __int64 v6; // rdx
  __int64 v7; // rcx
\n
  if ( a3 )\n
  {\n
    if ( a2 )\n
      result = (unsigned __int8)byte_C2D9 ^ 1u;\n
    else\n
      result = (unsigned __int8)byte_C1F0 ^ 1u;\n
  }\n
  else\n
  {\n
    result = (unsigned __int8)byte_C1F1 ^ 1u;\n
  }\n
  if ( !(_BYTE)result )\n
  {\n
    if ( byte_C2DA )\n
      __printf_chk();\n
    v5 = *(const void **)(a1 + 16);\n
    result = fwrite_unlocked(v5, 1u, *(_QWORD *)(
a1 + 8), stdout);\n
    if ( result != *(_QWORD *)(a1 + 8) )\n
    {\n
      sub_2F70();\n
      return sub_30C0(v5, 1, v6, v7);\n
    }\n
  }\n
  return result;\n
}
```

·\n T1 cosmetic (whitespace) T2 structural T3 contextual T4 reductive

Figure 2: The edits each DEFLATE-D tier makes to one Hex-Rays function, with each token colored by tier.

```
void FUN_0010b1a0(void){DAT_00124988=FUN_0010d530(0x
67,0,FUN_0010b050,FUN_0010b060,FUN_0010b080);if(DAT_
00124988!=0){return;}/*WARNING: Subroutine does not
 return*/FUN_00113270();}
```

T1 *cosmetic* 79 tokens (−11%)

```
void FUN_0010b1a0(void){DAT_00124988=FUN_0010d530(
103,0,FUN_0010b050,FUN_0010b060,FUN_0010b080);if(DAT
_00124988!=0)return;/*WARNING: Subroutine does not
 return*/FUN_00113270();}
```

T2 *structural* 76 tokens (−15%)

```
void a(void){g=b(103,0,c,d,e);if(g!=0)return;/*
WARNING: Subroutine does not return*/f();}
```

T3 *contextual* 32 tokens (−64%)

```
void a(void){g=b(103,0,c,d,e);if(g!=0)return;f();}
```

T4 *reductive* 23 tokens (−74%)

Figure 3: The function from Figure 1 after each tier of DEFLATE-D. Token counts and reductions versus raw (T0) are noted below each panel.

We group the transformations of DEFLATE-D into four cumulative tiers, ordered so that *information loss* is non-decreasing. Each tier admits only transformations whose loss is clearly bounded, and includes every transform of the tiers below it, so a caller selects one tier and receives all of them. While each tier is a convenient preset, every transform is individually named in Table 1 and can be enabled or disabled for finer-grained control. Despite their number, the transforms follow one organizing principle, since each either normalizes rendering that no tokenizer needs or discards material the decompiler synthesized and marked low-confidence. Facts anchored to real symbols are never touched. That boundary is a design constraint, not a side effect, since indiscriminate minification cuts tokens but can measurably degrade task quality [18]. A downstream analysis task that relies on the low-level information the later tiers truncate can therefore stop at T2, whereas a task that does not can take the larger savings of T3 and T4.

The four tiers escalate from purely cosmetic normalization to the removal of low-confidence analyst signal. Figure 2 previews all four tiers applied to one Hex-Rays function, coloring each token by the tier that compresses it. *T1 (Cosmetic)* normalizes formatting that is invisible to program semantics, namely whitespace, blank lines, indentation, and alignment, and is strictly semantics-preserving. *T2 (Structural)* adds semantics-preserving rewrites such as coalescing declarations, collapsing `if/else` chains into ternary expressions, and removing auto-generated address comments. *T3 (Contextual)* is the first lossy tier, discarding machine-generated decompiler bookkeeping, namely placeholder names and low-confidence type spellings, while preserving meaning. It acts only on identifiers and types whose machine-generated spelling marks them as synthesized, so that symbol-anchored names and types are preserved. *T4 (Reductive)* goes furthest, removing low-confidence analyst signal that is not recoverable from the text, such as calling-convention and ABI keywords and decompiler warning banners. Figure 3 extends the example of Figure 1 through all four tiers. Because it is dense in placeholder identifiers, it compresses more than a typical input, shrinking from 89 to 23 tokens by T4 on the GPT tokenizer, a 74% reduction. Table 1 lists every transformation in each tier of DEFLATE-D. We do not claim this set is exhaustive or that every pass is the right choice for every task, since the lossy tiers in particular trade signal that some downstream tasks need, as Section 6 examines. Rather, it is the collection of text-local, semantics-bounded rewrites we found effective on the studied renderings.

We use the terms *semantics-preserving* and *lossy* to classify the transformation tiers. A tier is semantics-preserving when it leaves the rendering's C semantics unchanged, and every T1 and T2 rewrite is guarded to meet this bar, as Section 2.2 and Section 4 detail. Of course, preserving semantics does not generally imply preserving information. However, the two coincide across T1 and T2 with one exception, the T2 comment pass. Removing the decompiler's address and storage-location comments does not alter behavior, but drops analyst-facing metadata that binary diffing or calling-convention recovery would keep. Therefore, we treat it as the single boundary case rather than a fully information-preserving rewrite. The T3 and T4 tiers are *lossy* because they discard decompiler-emitted material, such as placeholders or speculative casts, that the remaining text cannot reconstruct.

2.1 T1: Cosmetic (semantics-preserving)

First, cosmetic transformations normalize the formatting decompiler output carries purely for readability, namely whitespace, indentation, alignment, blank lines, and line breaks. Because the downstream reader is a tokenizer rather than an analyst, we argue this layout is pure overhead, and removing it is strictly semantics-preserving. On hand-written source the same normalization already compresses roughly a quarter of input tokens with negligible quality change [39]. Although decompiler output is not hand-written, the layout it carries is at least as redundant. It also removes a known source of tokenization instability, since byte-pair encoders fragment text differently around whitespace [30].

The one subtlety is that whitespace cannot always be dropped, so Deflate-D tightens spacing around punctuation and operators only where it leaves the C token stream unchanged, never gluing two identifiers or forming a longer operator token. The token savings from the whitespace passes are strongly tokenizer-dependent, near-free on encoders that already merge whitespace runs into a single token but material on those that tokenize each character separately.

2.2 T2: Structural (semantics-preserving)

Second, structural transformations rewrite the program text into a more compact but semantics-preserving form, consolidating declarations, eliding redundant braces and casts, folding assignments and same-target branches, and canonicalizing control-flow, all itemized in Table 1. Every such rewrite preserves the program semantics by construction, guarded against the cases that would otherwise change meaning or validity, such as the dangling `-else` and side-effect conditions Section 4 discusses. For instance, `inline-temps` folds `u = f(); return u;` into `return f();` only when the target is a locally declared identifier assigned once and read once, never a global, a struct member, a dereference, or any other lvalue whose store could be observed elsewhere. A text-local check cannot see the decompiler's data-flow, so it approximates this conservatively and declines to fire whenever the pattern is not unambiguously a single-use local spill. The comment pass is the one boundary case, removing the decompiler's auto-generated address and storage-location comments while keeping the `/* WARNING */` banners until T4, since a warning signals unreliable decompilation rather than mere rendering noise. These rewrites echo source-level simplifiers that strip redundant syntax [43, 45, 46, 53], but target patterns far denser in decompiler output than in authored code.

Richer rewrites stay outside our scope and are left to future work. These include `goto`-to-structured-control reconstruction [3, 5, 7, 9, 57], de-duplication of cloned blocks [15], semantics-preserving condition simplification [57], folding the cast-and-dereference idiom once a struct layout is recovered [8, 28, 37, 55], and normalizing Ghidra's p-code spill intrinsics to the equivalent C.

Finally, because Deflate-D is model-free, it does not consult a tokenizer. Its one length-driven pass, integer-literal minimization, therefore optimizes character count as a proxy for token count. Even so, the rewrite is value-preserving, and Section 4 discusses how well character count tracks token count. Across T2, we aim not to remove any information a model could use, so only the rendering shrinks while the content stays intact.

2.3 T3: Contextual (lossy)

Third, we remove contextual transformations *synthesized* by the decompiler, identifying it by its machine-generated spelling, so that symbol-anchored information is preserved. Meaning and internal consistency are untouched; what is lost is only the *identity* of a machine-generated artifact, such as the address in a `FUN_*` name.

Most of the saving comes from *placeholder-name compression*, which remaps the address- and offset-derived identifiers that dominate stripped output, namely default local, parameter, register, label, function, and global-data names, to the shortest collision-free tokens; Table 1 catalogs the spellings it recognizes. These names are synthesized from stack offsets and SSA indices rather than recovered from source, so compressing them discards little, a trade we quantify in Section 3.2. Two choices keep this safe even though the tool sees only text. A function name is a single file-global symbol, so it is remapped consistently across its definition, prototype, and every call site, while a thunk that wraps a genuine symbol such as `j_std::exception::exception` is left intact because the wrapped name is a real signal. An optional definition-gated mode renames only locally defined functions so that cross-file references never silently diverge. *Type simplification* re-spells the decompiler's verbose, inferred width types to compact aliases, but the admissible set is empirical rather than intuitive, since we measured every candidate on all three tokenizers and kept only those that never increase the token count. The wins are dominated by IDA's verbose `__intN` and `_QWORD` spellings and Binary Ninja's `intN_t` family, with a few narrower additions such as Ghidra's one-byte `undefined1` and a value-preserving rewrite of glibc's `__`-prefixed aliases to their POSIX typedefs. By contrast, spellings already compact for the tokenizer, such as Ghidra's `undefined8`, are excluded.

A handful of narrower re-spellings remove further machine-generated bookkeeping that survives the earlier tiers while preserving meaning. We only match mechanically generated spellings, so a DWARF/PDB-anchored name or a defined type, the genuine signal that symbol stripping is known to destroy [24], is never matched in the first place. This pattern check is what an integrated deployment could replace by reading the decompiler's confidence metadata directly, an option we return to in Section 6.

2.4 T4: Reductive (lossy)

Finally, the reductive tier T4 crosses a boundary the earlier tiers do not. It discards information a reverse engineer might want and that, unlike a placeholder's offset, cannot be recovered.

Most of its passes strip decompiler decorations that recur on nearly every function and only restate what the code already says, namely calling-convention and ABI keywords, the low-signal `const` qualifier, Hex-Rays' pseudo-width casts, Ghidra's synthetic `(code *)` cast, and glibc's `_chk` fortify suffix, all listed in Table 1. Each is cheap to drop and costs only a low-confidence hint, such as whether a binary was hardened, rather than program meaning. Two passes, `gettext` unwrapping and surplus-argument truncation, are heuristics a text-local tool cannot fully justify and are confined to this tier for that reason. Unwrapping the `gettext` translation-lookup family keeps the message string and discards only the runtime lookup. Truncating the surplus arguments a decompiler spills onto a call, by contrast, assumes the call really is the named

T1 – Cosmetic	**Semantics-preserving: cosmetic layout**
`ws-collapse`	Collapse consecutive horizontal spaces to a single space.
`ws-indent`	Strip leading indentation from every line.
`ws-trailing`	Strip trailing whitespace from every line.
`ws-blanklines`	Collapse consecutive blank lines.
`ws-newlines`	Join line breaks so the body spans as few physical lines as possible, keeping those ending `//` comments.
`ws-comments`	Collapse redundant whitespace inside comment bodies.
`ws-tighten`	Tighten space around punctuation and operators where it cannot change tokenization (never gluing `int d` or forming `a--b`).
T2 – Structural	**Semantics-preserving: structural rewrites**
`decl-coalesce`	Merge same-type uninitialized declarations, even non-adjacent: `int a; long b; int c;` → `int a, c; long b;`.
`brace-elision`	Drop braces around a single simple statement: `if (c) { x = 1; }` → `if (c) x = 1;` (guarding the dangling `else`).
`cast-elision`	Collapse stacked identical casts: `(int)(int)x` → `(int)x`.
`compound-assign`	Fold an expanded assignment into compound form: `x = x + y;` → `x += y;`.
`ternary`	Fold a same-target `if`/`else` assignment into a ternary: `if (c) x = a; else x = b;` → `x = c ? a : b;`.
`inline-temps`	Inline a single-use spill temporary: `u = f(); return u;` → `return f();`.
`cfg-canon`	Remove no-op `goto`s and labels left with no referencing `goto`.
`int-minform`	Re-spell a hex literal in decimal when that is shorter: `0x10` → `16`.
`deref-offset`	Rewrite a dereferenced pointer-add as a subscript: `*(p + 4)` → `p[4]`.
`drop-trailing-return`	Drop a redundant trailing `return;` at the end of a function body.
`comments`	Remove auto-generated address (`/* 0x4011a0 */`) and storage-location (`// rax`) comments; `/* WARNING */` is kept until T4.
`flag-temps`	Normalize Binary Ninja colon-flag temporaries (`cond:0`) into plain identifiers.
T3 – Contextual	**Lossy: machine-generated bookkeeping**
`compress-funcs`	Rename address-placeholder function names (`FUN_*`, `sub_*`, `j_sub_*`) across the unit; real wrapped symbols are kept.
`compress-names`	Remap placeholder locals, labels, and globals (`local_18`, `uVar3`, `LAB_*`, `DAT_*`) to the shortest collision-free tokens.
`simplify-types`	Re-spell verbose width types to compact aliases (`unsigned __int64` → `u64`), only where it reduces tokens.
`null-cast`	Drop a redundant pointer cast on a null constant: `(char *)0x0` → `0`.
`addr-of-index`	Rewrite the address-of-a-literal-index idiom: `&buf[0x10]` → `(buf + 0x10)`.
`strip-ptr-addr`	Drop the `.got.plt` slot address from a recovered import pointer: `PTR_free_0010b000` → `PTR_free`.
`piece-access`	Drop the trailing separator of a Ghidra piece access: `v._8_8_` → `v._8_8`.
T4 – Reductive	**Lossy: low-confidence analyst signal**
`strip-callconv`	Remove calling-convention and ABI keywords (`__cdecl`, `__fastcall`, `__noreturn`, IDA's `__hidden`).
`strip-const`	Remove the low-signal `const` Binary Ninja recovers on most pointer parameters.
`strip-width-cast`	Remove Hex-Rays pseudo-width casts (`(_BYTE)`, `(_DWORD)`); meaningful pointer casts are kept.
`drop-code-cast`	Drop Ghidra's `(code *)` cast on an indirect call: `(*(code *)PTR_x)()` → `(*PTR_x)()`.
`strip-chk`	Strip glibc's `_chk` FORTIFY suffix: `__printf_chk` → `printf`.
`strip-i18n`	Unwrap a `gettext`-family lookup to its message string: `dcgettext(0, "msg", 5)` → `"msg"`.
`trim-spurious-args`	Truncate spilled surplus arguments on curated fixed-arity libc calls: `setlocale(6, s, mc, mb, ...)` → `setlocale(6, s)`.
`comments-warning`	Remove the decompiler's `/* WARNING */` unreliability banners.
`thunk-elision`	Collapse a forwarding import trampoline to its prototype: `int f(...) { return f(...); }` → `int f(...);`.
`drop-crt`	Delete C-runtime and ELF scaffolding functions (`_start`, `frame_dummy`, `register_tm_clones`).
`drop-resolver-stubs`	Delete Binary Ninja lazy-binding resolver-stub families (≥ 3 constant-load tail-calls to one shared resolver).

Table 1: Transformations implemented in DEFLATE-D, grouped into four cumulative tiers of increasing information loss.

libc function. It is therefore gated on a curated set of fixed-arity functions and excludes prototypes and definitions, a heuristic whose risk Section 4 bounds. At T4, DEFLATE-D also drops the `/* WARNING */` banners. Unlike the address comments T2 removes without altering semantics, these can signal unreliable output, so removing them is a genuine loss, though usually acceptable.

T4 also removes whole boilerplate functions that carry no analyst signal beyond their signature, forwarding thunks collapsed to their prototype together with the C-runtime, ELF-scaffolding, and lazy-binding resolver functions deleted outright. Each deletion fires only on an unmistakable structural signature, and a reference guard keeps any function a surviving caller still references, so no removal leaves a dangling call. In the samples used for evaluation, the resolver stubs alone are the largest source of avoidable Binary Ninja tokens. We leave collapsing compiler idioms inside a function body, such as stack-canary scaffolding, for future work.

2.5 Implementation

Deflate-D is a deterministic, model-free reformatter in which every transform is a rule, with no trained compressor and no per-token API spend, unlike learned prompt compressors [21, 40]. Transformations operate on the textual rendering through a string-safe lexer that protects string, character, and comment regions, so content inside a literal is never altered. This lexer is deliberately not a full C front-end, since decompiler output is frequently not valid, compilable C. Deflate-D therefore tracks only enough lexical state to keep every edit literal-safe, rather than committing to a grammar the input would routinely violate. The T1 and T2 passes are backend-agnostic, and only the T3 and T4 placeholder patterns and ABI keywords are configured per decompiler. We target the three dominant backends, Ghidra, IDA Pro (Hex-Rays), and Binary Ninja. The per-decompiler spelling vocabularies are the text-local proxy for each tool's provenance metadata, which an integrated deployment could read directly instead, as Section 6 discusses.

The pipeline is single-pass, not iterated to a fixed point. A T4 whole-function deletion can expose a pattern an earlier pass walked past, so re-running on its own output compresses marginally further on a minority of files, about 18 of 327. We keep and report a single pass, since the gain does not justify the extra processing.

Deflate-D is a prototype implementation, released as open source [11]. The safety guards, the dangling-`else` and arity conditions of Section 4 together with the reference guard of Section 2.4, are exercised by accompanying regression tests.

3 Evaluation

Compressing decompiler output is only worthwhile if it saves enough tokens to matter and does so without degrading the analysis that consumes it. We frame these two conditions as our research questions, with RQ1 measuring how many tokens each tier removes across decompilers and tokenizers, and RQ2 testing whether that compression preserves downstream task quality for the LLM.

3.1 RQ1: Token savings

First, we assess Deflate-D's token savings in isolation, with no model in the loop. We measure how much each tier shrinks decompiler output across three decompilers and several tokenizers.

3.1.1 Corpus and decompilation. We use GNU coreutils 9.5 as our corpus, a widely used, freely redistributable suite distributed with its source, so the study is exact and reproducible. From it we build the suite at the autotools default of `-O2`, strip the executables, and compile to x86-64 Linux ELF. This is the configuration an analyst commonly meets, and underlies the RQ2 datasets of Section 3.2.

We decompile every binary with the three mainstream decompilers, Ghidra 12.1.2 [36], Binary Ninja 5.3 [51], and IDA Pro/Hex-Rays 9.3 [17], each through its standard headless export, so the output is the ordinary C pseudocode an analyst reads. Because the binaries are stripped, we run Hex-Rays with its Lumina metadata service disabled, so it does not pull recovered symbol names from the cloud and stays placeholder-named like Ghidra's and Binary Ninja's, keeping the three renderings comparable to the stripped setting. All three produced output for every binary, so the cross-decompiler comparisons span the same set, not disjoint subsets.

3.1.2 Token counting. We count under the tokenizers of three widely used commercial model families, OpenAI, Anthropic, and Google, so the savings are not an artifact of any single vendor's vocabulary. OpenAI's GPT tokenizer, the `o200k_base` encoding that Figure 1 visualizes, runs locally through `tiktoken` [38] at version 0.13.0, while we reach the Anthropic and Google tokenizers through their free `count_tokens` API endpoints, so counting never incurs model cost. Counting is a deterministic, generation-free function of the text, exact and cheap to repeat, which is what makes a 109-binary, five-tier, three-decompiler sweep affordable. The API-side counts reflect those endpoints as accessed during our evaluation and, unlike the version-pinned `tiktoken`, are not themselves pinned, so a later revision could shift them slightly. For every binary and decompiler we record the per-tier count at T0 through T4 and report the percentage reduction versus the raw decompiler output, with headline numbers on the GPT tokenizer.

3.1.3 Results. We report the token reduction per decompiler and tier in Table 2, split into per-binary (left) and per-function (right) groups. In the per-binary group, the semantics-preserving tiers T1/T2 already remove a fifth to a third of all tokens. The contextual tier T3 then lifts every decompiler to around half, by compressing the machine-generated placeholder names, function names, and types that dominate stripped output. Finally, the reductive tier T4 pushes a little beyond that, gaining the most on Ghidra. These reductions hold under all three considered tokenizers, with every decompiler's T4 median in the 45–61% band. The per-binary medians even preserve the strict ordering $T1 < T2 < T3 < T4$ in every decompiler-by-tokenizer combination. A coarser or finer vocabulary shifts the absolute savings but never reorders the tiers, making the reduction a property of the rendering, not any single vocabulary. Of course, these figures are absolute reductions for a single corpus and build. The absolute savings may therefore shift for other languages, compilers, optimization levels, or architectures, as Section 4 briefly discusses.

Because not all binary-analysis tasks operate on whole binaries, we now turn to the per-function token reduction. Here, we count each function on its own, above a 32-token floor below which the percentage is mostly noise. The per-function reduction then fans out far more than the tight per-binary one, with Ghidra's T3 window running from about 3% to 67% between the 5th and 95th percentiles. In contrast to binaries, functions differ considerably in how much compressible material they carry, from placeholder names and verbose types to formatting and boilerplate. Thus, T3 tends to reclaim more on large functions and T4 more on small ones, where fixed ABI boilerplate and deletable stub functions make up a larger share of the total. This wide upper end is inflated further by the small fraction of functions that T4 removes outright, the C-runtime and ELF scaffolding of Section 2.4. Each is a 100% reduction that inflates the average, so we report the median as typical.

Matched tier by tier against prior compression work, our reductions are competitive. The semantics-preserving tiers line up with reversible source-code rewriters, which report roughly 9–25% on authored source [39, 45, 46], and T1 alone reproduces the formatting-removal baseline of [39] at a 17–24% median on the GPT tokenizer. The lossy tiers, whose natural comparison is pruning compressors [40, 53], roughly double that to about half. We

		*per-binary (P5 / **med** / P95, n=109)*									*per-function (P5 / **med** / P95, n≤28453)*								
		gpt-5.1			claude-opus-4-8			gemini-3.1-pro-preview			gpt-5.1			claude-opus-4-8			gemini-3.1-pro-preview		
Ghidra	T1	-16.4%	**-17.3%**	-18.1%	-17.8%	**-18.7%**	-20.4%	-19.3%	**-20.6%**	-21.2%	-02.9%	**-15.0%**	-22.8%	-09.9%	**-15.6%**	-23.0%	-14.4%	**-21.1%**	-25.7%
	T2	-20.7%	**-21.8%**	-23.3%	-20.9%	**-22.2%**	-24.6%	-22.5%	**-23.7%**	-24.7%	-02.9%	**-21.9%**	-41.9%	-10.3%	**-20.5%**	-39.4%	-15.4%	**-24.0%**	-44.1%
	T3	-44.2%	**-45.3%**	-47.2%	-39.9%	**-41.3%**	-44.2%	-50.5%	**-52.3%**	-53.5%	-02.9%	**-51.1%**	-66.1%	-10.8%	**-43.2%**	-59.1%	-17.4%	**-52.8%**	-67.2%
	T4	-51.4%	**-56.4%**	-58.5%	-50.4%	**-55.7%**	-57.8%	-56.4%	**-61.3%**	-63.3%	-48.6%	**-60.4%**	-81.6%	-46.0%	**-62.2%**	-74.6%	-49.3%	**-65.9%**	-81.8%
Binja	T1	-21.1%	**-22.5%**	-23.5%	-24.4%	**-26.5%**	-30.8%	-23.0%	**-25.9%**	-28.2%	-15.0%	**-18.6%**	-27.0%	-08.8%	**-17.5%**	-26.1%	-13.9%	**-21.8%**	-30.2%
	T2	-26.3%	**-28.1%**	-29.2%	-29.3%	**-31.9%**	-35.5%	-26.1%	**-29.5%**	-31.7%	-19.7%	**-29.4%**	-35.1%	-16.7%	**-28.0%**	-37.5%	-18.2%	**-30.2%**	-38.2%
	T3	-49.6%	**-51.2%**	-52.0%	-46.5%	**-47.9%**	-50.8%	-54.6%	**-56.0%**	-57.0%	-30.3%	**-58.2%**	-64.1%	-23.8%	**-50.0%**	-59.6%	-31.7%	**-54.7%**	-69.1%
	T4	-52.5%	**-55.1%**	-56.7%	-50.0%	**-52.4%**	-54.1%	-57.1%	**-59.5%**	-60.9%	-47.1%	**-60.0%**	-69.4%	-42.5%	**-55.4%**	-63.6%	-47.8%	**-60.0%**	-69.2%
Hex-Rays	T1	-22.8%	**-24.1%**	-25.7%	-21.4%	**-22.5%**	-24.3%	-21.3%	**-23.3%**	-24.2%	-11.7%	**-21.9%**	-29.3%	-07.6%	**-14.7%**	-26.0%	-11.3%	**-20.6%**	-29.0%
	T2	-34.6%	**-35.3%**	-37.0%	-32.9%	**-34.0%**	-35.8%	-31.8%	**-33.1%**	-34.1%	-15.2%	**-29.6%**	-44.6%	-07.6%	**-21.4%**	-40.0%	-12.5%	**-26.8%**	-40.9%
	T3	-45.7%	**-46.5%**	-48.0%	-39.6%	**-41.1%**	-42.7%	-49.8%	**-51.2%**	-54.0%	-20.0%	**-49.2%**	-61.1%	-09.1%	**-37.9%**	-54.0%	-14.3%	**-44.8%**	-58.2%
	T4	-48.0%	**-49.5%**	-50.2%	-42.8%	**-45.1%**	-46.0%	-52.6%	**-53.9%**	-56.1%	-39.1%	**-52.9%**	-64.1%	-25.0%	**-45.1%**	-57.9%	-35.6%	**-49.9%**	-60.5%

Table 2: Token reduction vs. raw decompiler output across 109 GNU coreutils binaries, per decompiler and tier. *Left*: per-binary spread; *right*: per-function spread, far wider as savings track a function's placeholder-name density. One column per tokenizer; cells shaded pale (low) to solid green (high). T3 and T4 are lossy.

read the larger reductions conservatively, since decompiler output is more redundant than authored source and the lossy tiers do not round-trip like a reversible rewriter. The claim is therefore not a bigger headline number but that a deterministic, model-free, semantics-preserving-by-default reformatter matches or exceeds the reversible source-code methods at no model cost, and without the task-quality regression that aggressive minification incurs [18].

These savings are consistent across the corpus and substantial at scale. The per-binary spread is tight at every tier, with the middle 90% within a few points of the median. The three decompilers reach broadly comparable reductions by different routes, with Hex-Rays gaining through the verbose layout the semantics-preserving tiers normalize, Binary Ninja through the semantics-preserving and contextual tiers, and Ghidra, the most placeholder-heavy, through T3 and T4. In dollars, at Opus 4.8's input rate of $5 per million tokens [2] and counted on its own tokenizer, one pass over the corpus through one decompiler costs about $52 for the most verbose, Ghidra, and less for the rest, falling to about $24 at T4. Because output tokens stay flat across tiers for simple tasks, as Section 3.2 shows, the input saving carries to the total.

3.2 RQ2: Downstream task quality

Second, we test whether those savings come at a downstream cost, measuring how well an LLM performs the analyst's task on each tier, not how small the text is, across three binary-analysis tasks.

3.2.1 Tasks and datasets. We evaluate each task against a different kind of ground truth, each backed by a dataset released with its paper, and present them below in order of increasing tier sensitivity. Section 3.2.3 scores *function summarization* against human-written source comments from CAPYBARA [1], Section 3.2.4 scores *variable naming* against DWARF debug symbols from ReSym [55], and Section 3.2.5 scores *function naming* against the original identifier from SymGen [23]. We evaluate 400 functions per task, the full size of SymGen's released sample test set, so that each task contributes equally to the convergent-evidence claim and the study stays within a fixed evaluation budget. For the two larger datasets, we draw the 400 functions by a fixed-seed permutation rather than taking them in file order, since ReSym groups its files by binary. This keeps the subset reproducible and unbiased with respect to that ordering.

3.2.2 Experimental design. We use a single general-purpose model, Claude Opus 4.8, with one minimal, format-constrained prompt per task, identical across all tiers, so the decompiler rendering is the only input that varies. We revisit this single-model scope in Section 4. This is deliberately a general frontier model rather than the fine-tuned, task-specific models the source papers trained, for two reasons. First, it gives ecological validity, since the agentic and MCP pipelines that motivate this work, which Section 5 discusses, feed DEFLATE-D's output to exactly such general models, not to per-task fine-tunes. Second, the source papers fine-tuned their models on *raw* decompiler output, so feeding them a compressed rendering would measure distribution shift from their training format, not information loss, confounding even the semantics-preserving tiers. We argue that using one of the most capable available model also makes the test conservative, because we would expect it to be most able to exploit residual signal in the tokens DEFLATE-D discards.

For every task, we render each function at every tier from raw T0 through T4 and prompt the model with each rendering independently, a *paired*, within-subject design that contrasts tiers on the *same* code rather than on disjoint samples. Each prompt runs without extended thinking, which is opt-in on Opus 4.8 and we leave at its default of off. These tasks ask for only a short answer, and the fine-tuned source models likewise produce theirs directly, with no reasoning step to spend output tokens on. To damp the model's residual run-to-run variance, we issue three prompts per function-tier pair and average them into one per-function score, a deliberate variance-reduction measure within our evaluation budget. From these scores we report each tier's mean with a 95% percentile bootstrap confidence interval, together with the mean per-function *delta versus raw* and its 95% interval, on which the non-inferiority

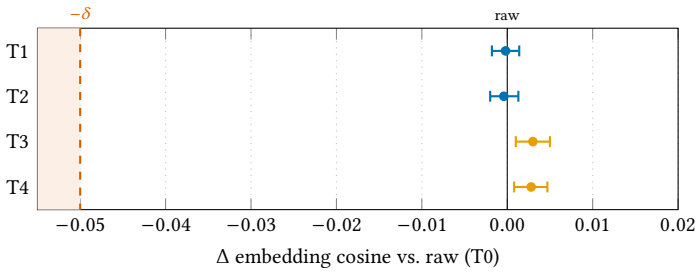

Figure 4: Function summarization: per-tier Δ in embedding cosine vs. raw. Dots are the mean per-function Δ and bars the 95% CI; the dashed line is the non-inferiority margin.

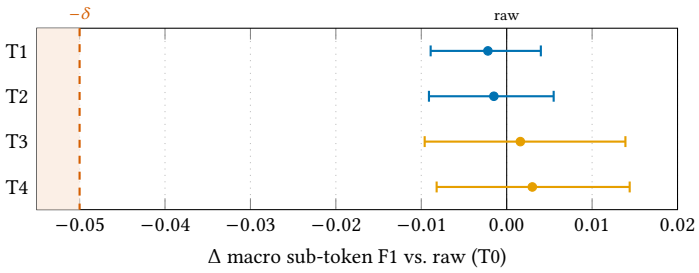

Figure 5: Variable naming: per-tier Δ in macro sub-token F1 vs. raw. Dots are the mean per-function Δ and bars the 95% CI; the dashed line is the non-inferiority margin.

decision below is based. The claim under test is *non-inferiority* against a single margin $\delta = 0.05$, fixed in advance of scoring any tier and shared by all three tasks. We treat δ as a pragmatic decision threshold rather than a validated equivalence bound. Establishing a per-task margin anchored to a wrong-input floor remains future work. A tier counts as non-inferior when the lower bound of its 95% delta interval lies above $-\delta$. Because we read the decision off the lower limit of a two-sided 95% interval, this is a one-sided non-inferiority test at $\alpha = 0.025$, stricter than the conventional 5% level, which would instead use the lower bound of a 90% interval. We defer the threats it carries, mainly the human-comment references and the two-decompiler span of the RQ2 datasets, to Section 4.

3.2.3 Function summarization. Given a stripped function, the model writes a short natural-language comment describing what it does, and because the reference is a written summary rather than a single ground-truth string, the metric must tolerate paraphrase. For that reference we use CAPYBARA, the dataset released with BinT5, which pairs Ghidra-decompiled functions with the documentation their original authors wrote, extracted from the source as single- and multi-line comments [1]. This matters because the reference is a genuine human comment, not a model-generated one, so it sidesteps the validity threat that prior binary-summarization references are themselves LLM output [42]. We take its deduplicated, fully stripped variant, so the input is Ghidra output with placeholders such as `FUN_*` and `param_*`, just as an analyst meets it.

Two choices are specific to this task, in how we prepare and score the input. First, CAPYBARA stores its code as a flat token stream, so we reconstruct conventional C layout once before applying the tiers. T0 therefore resembles real decompiler output, and this is the only preprocessing. Second, we ask only for a comment of at most 96 words, and score with embedding cosine against the human reference, paraphrase-robust and insensitive to lexical drift [24].

Across the 400 evaluated functions, compression leaves the score essentially unchanged. Raw decompiler output scores a mean embedding cosine of 0.60 against the human reference, and Figure 4 shows every tier non-inferior to it within the margin. The semantics-preserving tiers T1 and T2 are indistinguishable from raw, within 0.001, and even the lossy tiers T3 and T4 differ by a marginal +0.003, sitting if anything just above raw, with every CI about an order of magnitude inside the margin. Notably, embedding cosine stays high between any two pieces of same-domain text, so against these short human comments the scores sit high and close together, leaving the metric little room to separate the tiers. Even within that limited range no tier differs from raw by more than 0.003, so our compression leaves summarization quality untouched.

3.2.4 Variable naming. Given a decompiled function whose locals and parameters carry meaningless placeholders, the model infers a concise, descriptive name for each variable from how it is used. Generally, variable naming produces many answers per function rather than one, and it probes the tokens DEFLATE-D most aggressively rewrites at T3, namely the placeholder identifiers themselves.

ReSym fine-tunes two LLMs, a VarDecoder for variables and a FieldDecoder for struct fields, and aggregates their outputs with a Prolog-based cross-checking step [55]. Its scorer is deliberately strict, counting a predicted variable name as correct only on *exact* string equality with the DWARF name, with type accuracy scored likewise, and reporting per-variable name accuracy.

We reuse ReSym's VarDecoder test data verbatim, namely functions decompiled with IDA/Hex-Rays, paired with DWARF variable names and types as the reference. But we depart from ReSym in two ways that the general-model setting forces, namely the scoring metric and the prompt. For scoring, we replace exact match with macro sub-token F1 under CodeWordNet synonym matching [23, 25]. Exact match suits a model fine-tuned to reproduce DWARF spellings, but it is too brittle for a general model and would conflate a tier's effect with surface lexical drift. Sub-token, synonym-aware F1 instead isolates whether the renaming *information* survives the rendering. For the prompt, ReSym prelists the specific placeholder identifiers it asks about, naming particular variables such as `a1` and `dest`, but DEFLATE-D's T3 renaming rewrites those very identifiers, turning `a1` into `a`. A verbatim prelisted prompt would therefore name variables that no longer occur in the compressed body. The only alternatives, regenerating the variable list per tier or abandoning the paired design, would let the prompt vary across conditions and confound the rendering effect we are isolating. Instead we hold one fixed prompt across all tiers that asks for one `<placeholder>: <name>` line per variable, and prompt Opus rather than ReSym's fine-tuned decoders. At T3, DEFLATE-D renames placeholders deterministically and one-to-one within each function, `v4` to `a` and so on, so the map inverts exactly and we map predictions back through it before scoring. At T0 the map is empty and names match the reference directly. Because this protocol scores only names and not types, its absolute

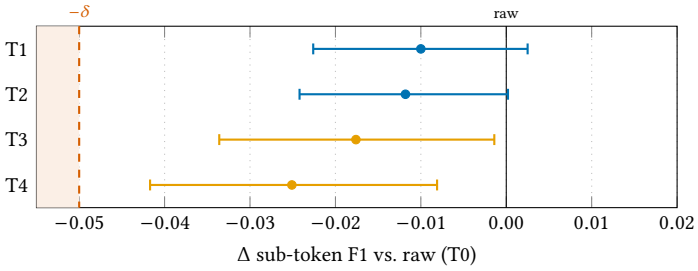

Figure 6: Function naming: per-tier Δ in sub-token F1 vs. raw. Dots are the mean per-function Δ and bars the 95% CI; the dashed line is the non-inferiority margin.

	Function naming	Variable naming	Function summarization
T0	262,131	169,271	644,600
T1	223,106 (−12% / −18%)	150,680 (−8% / −13%)	507,118 (−17% / −21%)
T2	212,828 (−15% / −22%)	131,498 (−16% / −24%)	472,633 (−21% / −26%)
T3	160,355 (−31% / −45%)	110,815 (−28% / −44%)	335,889 (−40% / −51%)
T4	157,586 (−31% / −46%)	108,587 (−30% / −48%)	330,546 (−41% / −52%)

Table 3: Total billed *input* tokens over the evaluated functions, with the per-function reduction vs. raw (T0) over the full prompt / over the code alone (fixed instruction removed).

numbers are not directly comparable to ReSym's name-and-type accuracy, so the comparison we draw stays within our own tiers.

Variable naming is intrinsically hard for a general model, as the absolute scores show. More than half of the functions score zero even on raw output, with a raw T0 mean macro F1 of 0.147 and a median of 0, so Opus recovers a recognizable name for only a minority of variables. What RQ2 tests, though, is the tier effect, and that is flat. Every tier, including the lossy ones, is statistically indistinguishable from raw, as Figure 5 shows, with per-function deltas all within ±0.003, namely −0.002 at T1, −0.002 at T2, +0.002 at T3, and +0.003 at T4, and 95% bootstrap CIs that straddle zero.

That flat result is the meaningful part, because T3 rewrites exactly the placeholder identifiers this task is trying to recover, yet those placeholders never held the answer to begin with. Hex-Rays assigns generic placeholders like `a1` because the binary carries no symbol names, and the real names live only in the DWARF reference we score against. At every tier, therefore, the model infers each name from how the variable is used, and T3's rewrite of them takes nothing away. The low absolute scores then reflect how hard variable naming is for a general model, not any loss from compression. Because a function scoring zero at every tier contributes nothing to its delta, these zeros lower the absolute level without distorting the tier comparison RQ2 tests. The floor effect does weaken the evidence, since a function that fails on both raw and compressed output yields a zero delta whether or not the rewrite preserved anything. Thus, we read the flat result as consistent with preservation rather than as strong proof of it, and lean the preservation claim on function naming, where the scores are high enough to expose a loss if one occurred.

3.2.5 Function naming. Given only a stripped function body, the model predicts the identifier the function bore in source. Here, SymGen fine-tunes a Code Llama model on decompiled functions to emit the source name for each held-out function [23]. Its scorer does not demand an exact match but a synonym-aware sub-token F1, crediting near-synonyms like `send` and `dispatch` via the CodeWordNet synonym clusters it adopts from SymLM [25].

We reuse SymGen's released test split and its DWARF-derived reference names verbatim. For scoring, we use the same sub-token F1, dropping only SymGen's lemmatization and sentencepiece segmentation, which apply identically to every tier and so cancel in the comparison, and would otherwise require shipping a trained segmentation model. Where SymGen scores its own fine-tuned model, we prompt Opus and reuse its instruction verbatim, including the `[MASK]` clause, which refers to the function's hidden name, since that marker survives in the body at every tier. We append one output-format line asking for a single response of the form "The predicted function name is *name*". This is our only change to SymGen's wording, for clean parsing without altering the task.

Function naming is the one task where the rendering leaves a mark, though a small one. Raw decompiler output scores a mean sub-token F1 of 0.46, a solid result for identifier recovery, and Figure 6 shows every tier remaining non-inferior within the margin. The semantics-preserving tiers are statistically indistinguishable from raw, −0.010 at T1 and −0.012 at T2 with CIs that straddle zero, but the lossy tiers bring a small, monotone decline, −0.018 at T3 and −0.025 at T4. This is a small but statistically resolvable decline whose delta CIs at T3 and T4 exclude zero, and it is the only such tier effect we observe anywhere in RQ2. That the decline appears here is expected, because function naming asks the model to reduce the whole body to a single identifier and so draws on the widest range of cues. The low-confidence signal the lossy tiers strip is immaterial for the other two tasks and affects only this one. Negligible as it is, the worst-case CI bound of −0.042 still clears δ, so even the most aggressive tier stays non-inferior.

3.2.6 Synthesis. We designed the study around two expectations, that the semantics-preserving tiers would track raw and that any degradation would stay confined to the lossy tiers and deepest at the reductive T4. Both hold across the three tasks, where function summarization and variable naming stay flat at every tier and only function naming declines, by a small amount that stays within the margin and is deepest at T4. Two of these flat results carry caveats that make function naming the most informative of the three. Summarization sits high and close across all tiers because the human reference comments are short. Variable naming is flat largely because more than half of the functions already score zero on raw output, so those functions contribute a zero delta for reasons unrelated to preservation.

Function naming, the most compression-sensitive of the three and the one free of these floor and ceiling effects, is thus the decisive test. Even there, the compression discards no signal the downstream reader can use, since the decline stays within the margin fixed in advance. This near-flat quality saves real tokens. Table 3 reports the per-task input-token reduction, from 8% at T1 to 41% at T4, largest on summarization, while output stays near raw.

The reduction over the full prompt is lower than the per-code reduction of Table 2 because the fixed task instruction is a considerable share of the input in these short-function benchmarks, and, being identical across tiers, cannot compress. This dilution is an artifact of the benchmark shape, not of Deflate-D. A session that amortizes the instruction over more decompiled code drives the realized saving toward the code-only figure, while a pipeline dominated by a large fixed system prompt realizes correspondingly less. Measured on the code alone, the reductions reach 46–52% at T4, consistent with the coreutils study in Table 2, where every decompiler's T4 median falls in the 45–61% band, confirming that Deflate-D compresses this input as it does the corpus.

4 Threats to Validity

Our main metric throughout the paper is the input-token count, a proxy for the price, latency, and context pressure of a prompt. It is exact for context occupancy and input-side billing, but excludes output tokens, which providers bill at a higher rate. Realized cost savings therefore depend on the input-to-output ratio, largest for ingestion-heavy reverse-engineering workloads, while the latency and context-window benefits hold unconditionally. It is also one step removed from what Deflate-D optimizes, since the tool reduces a tokenizer-independent proxy, the character count, rather than any single vocabulary. A rewrite therefore lowers tokens on average but not always, as Section 2 explains. We bound this where the risk concentrates, admitting into T3's lossy type vocabulary only rewrites measured never to raise the token count on all three tokenizers, and reporting every saving against a concrete tokenizer, not the character count.

A low token count, however, says nothing about whether a leaner rendering remains useful, so we treat usefulness as something to preserve and to measure rather than to assume. On preservation, our claim to preserve information holds only relative to these tasks, since the T2 comment pass discards the decompiler's address and storage-location notes. Those are harmless for the tasks here but carry signal that binary diffing or calling-convention recovery would retain. On measurement, the RQ2 downstream tasks report non-inferiority on the compacted output, so each saving is tied to a quality signal rather than inferred from the token count alone. These signals are themselves imperfect proxies, sub-token F1 against DWARF for naming and embedding cosine for summarization, so our claim rests on agreement across the three tasks rather than any single signal. CAPYBARA's human source comments [1] also let us avoid the model-generated-reference threat that affects prior binary-summarization data [42]. Those comments are short, so summarization cosine sits high and close across all tiers, which limits the metric's separating power but does so uniformly. Variable-naming F1 faces the opposite bound, since more than half of the functions score zero, so it too has limited separating power. Its non-inferiority is weak evidence, leaving function naming the most discriminating of the three.

Even granting the metrics, the model's score could move for reasons other than the rendering tier, primarily its run-to-run randomness and its reasoning mode. We control both, averaging three runs per function to damp this randomness and prompting Claude Opus 4.8 without extended thinking, which is opt-in and we leave at its default of off, so that the rendering is the only input that varies. The second control comes at a cost to scope, since it leaves open whether the same tier effect holds with extended thinking enabled, a setting we leave to future work. Enabling it could also shift the token accounting rather than only the quality, since any extra reasoning is billed as output at a higher rate, need not shrink with the input, and could therefore narrow the net saving.

A second internal threat concerns the implementation rather than the design. Deflate-D's rewrites are meant to be semantics-preserving, and each carries an explicit guard for the cases that would change meaning, such as the dangling-`else` and sole-declaration conditions of Section 2, or the single-use-local condition that keeps `inline-temps` from folding away a store to aliased or otherwise observable memory. Because the rewrites match text locally rather than transforming a type-checked AST, each guard approximates from spelling what the decompiler's data-flow would establish directly. We therefore validate them by test rather than by a differential equivalence check, and rare edge cases can survive until that check is in place, which we leave to future work. A related risk affects the passes that key on a function's name, such as the `_chk`-suffix strip of Section 2.4, which assume the call reaches that libc function, an assumption the curated name set and conservative firing bound but do not remove.

Beyond our setup, the results speak only to the corpora we ran. RQ1 measures one corpus, GNU coreutils 9.5 compiled at `-O2` to stripped x86-64 ELF, so the absolute reductions may shift for other languages, compilers, optimization levels, architectures, or obfuscated binaries. RQ2 fixes a single model, by design to isolate the rendering effect, and draws on datasets that span only two decompilers. Because each task is tied to one decompiler and the datasets differ from the RQ1 corpus, the agreement across decompilers is a consistency check rather than a controlled comparison. We therefore observe a consistent tier pattern under this setting rather than a guarantee that the non-inferiority transfers to other models or decompilers. Because the pass set is tuned to these renderings, matching the savings on other decompilers or models may require additional passes rather than this set verbatim. The three tasks are also comparatively simple, one short answer per function, so whether the savings hold without quality loss on larger tasks that reason across a whole binary remains open.

We also cannot rule out that the model saw these public datasets in training, since RQ2's datasets are widely available, so the absolute scores may be optimistic. Such memorization, however, would recall the same answer regardless of rendering, so it lifts every tier alike and cancels in the within-function comparison our claim rests on. Only the absolute levels are exposed to it, not the tier deltas. Evaluation on less widely available binaries remains future work.

The study is also budget-bounded, so each task uses a 400-function split rather than the full released datasets, which costs statistical power. Our paired within-function design recovers much of it by comparing tiers on the same function rather than across disjoint samples. We also do not compare against a learned or LLM-driven compressor prompted to shorten the same code, a baseline that would test whether a model-based reduction preserves more signal per token. We leave that comparison to future work, noting that Deflate-D's model-free design is a deliberate trade for determinism and zero per-token cost.

5 Related Work

Our work bridges two research areas that rarely overlap, the methods that compress the input to a code LLM and the binary-analysis pipelines that feed such models decompiler output. The first spans prompt and source-code compression and the tokenization effects that make rendering matter. The second spans the LLM-over-decompilation systems, the slicing methods that choose which code to send, and the type- and name-recovery work whose findings tell us which signal is safe to drop. We position Deflate-D against each below. None targets the rendered decompiler output that Deflate-D optimizes, but each addresses a neighboring problem.

General-purpose prompt compressors such as LLMLingua [21] and LLMLingua-2 [40] prune low-information tokens with a trained auxiliary model. Recent surveys review the broader area [31]. A parallel line shrinks *source code* for LLMs, where Token Sugar introduces reversible shorthand for verbose patterns (∼15%) [46], AI-oriented grammars strip redundant syntax (∼9–35%, tokenizer-dependent) [45], and attention- or structure-guided simplifiers drop the least informative tokens [43, 53]. The closest neighbor [39] strips the same formatting elements (indentation, whitespace, and newlines, comments preserved) that our cosmetic tier T1 normalizes, reporting ≈24.5% token savings with negligible quality loss. T1 therefore carries that method over to decompiler output, where the 17–24% per-binary median reduction that Table 2 records on the GPT tokenizer reproduces their source-code result in the binary-analysis domain. It is lower there because the bulk of decompiler output is identifiers and casts rather than indentation.

Our contribution lies *above* T1, along four axes: (i) it targets *decompiler output*, which carries far more structural noise (placeholder names, speculative casts, auto-comments) than hand-written source; (ii) it is deterministic and model-free, with no trained compressor and no per-token API spend; (iii) it adds the strictly semantics-preserving structural rewrites of T2 and the lossy tiers T3 and T4, which discard only machine-generated bookkeeping and low-confidence signal respectively; and (iv) it is free to be *irreversible*. These axes make a single head-to-head reduction percentage misleading, so the meaningful comparison is *tier-matched*. It places our semantics-preserving T1/T2 next to the reductions reported for reversible source rewriters (∼9–35% on dense, authored source [39, 45, 46]), and our lossy T3/T4 next to learned or attention-guided pruners [40, 53] that can compress further but need a trained model and produce degraded, less readable output. Because decompiler output is more redundant, a deterministic, model-free reformatter recovers a comparable *semantics-preserving* fraction here at no model cost.

Source-code compressors must be reversible, because the source they shrink is a codebase that will be edited and recompiled. Token Sugar's shorthand is reversible [46], another keeps a human-readable copy via a bidirectional tool [39], and a compiler-tokenization scheme restores its compressed tokens from a contextual dictionary [32]. Decompiler output, by contrast, is regenerated on demand from the binary, so there is nothing to restore it to, and for stripped or third-party binaries no original source exists at all. This frees Deflate-D to discard information that source-code reformatters cannot. Still, to our knowledge, no prior work treats decompiler output as a compression target in its own right, characterizing token-reduction transforms by information loss and restricting the lossy ones to speculative content, as opposed to compressing source code or selecting which code to send. None evaluates the downstream binary-analysis impact of such compression either. The semantics-preserving/lossy distinction matters in practice, since naively minifying agent prompts saves 42% of tokens but costs 12 percentage points of task success [18], the regression our semantics-preserving tiers avoid.

Tokenizer choice materially affects models on binary code [34], and BPE tokenization is unstable under semantically irrelevant changes such as whitespace or identifier renaming, which shifts model behavior [30]. This motivates T3, our contextual tier, where rewriting placeholder identifiers to short, uniform tokens reduces the token count and removes the arbitrary spellings behind this instability. LLMs also struggle to reason over low-level representations [22], motivating reformatting rather than feeding raw IR.

A growing body of work feeds decompiler output to LLMs for function summarization, vulnerability detection, and decompilation refinement [13, 20, 24, 47]. BinSum shows decompiled C is the most effective input modality and quantifies the cost-at-scale of these pipelines (thousands of dollars and billions of tokens for one study) [24]. A human study of LLM-assisted reverse engineering finds analysts feed decompiled code to LLMs across the workflow, and that large or unfamiliar inputs degrade output [4]. DeGPT takes the opposite direction, *enriching* decompiler output with comments and better names [19], whereas Deflate-D makes ingestion *cheaper and faster*. The two are complementary. Other work improves LLM-based decompilation itself by preprocessing the assembly the model consumes [54], separate from reducing the token cost of the rendered result. Production RE assistants embed this ingestion in interactive and agentic loops [48, 49], where surveys flag limited token budgets as a core challenge [41] and right-sizing inputs measurably improves recall in vulnerability detection [44]. This ingestion increasingly runs through reverse-engineering MCP servers that hand decompiler output to LLM agents, with widely used ones across Ghidra, IDA Pro, and Binary Ninja alike [6, 10, 27, 35]. Inspecting their source, we find they forward that output essentially raw, applying at most trailing-whitespace trimming. What little token-aware handling exists is ad hoc, such as collapsing repeated spaces or suppressing the address column, both already covered by T1. None of them, though, reformats the pseudocode by its information content. Deflate-D supplies that missing format-level optimization, sitting upstream of every such consumer.

A parallel way to cut tokens is to send less code rather than to render it more compactly. Slicing takes that route, with LATTE constructing taint-driven slices of decompiled code [33] and LLMxCPG selecting code-property-graph subgraphs that cut 68–91% of source code [29]. These are *selectors*; Deflate-D, in contrast, is a *reformatter* of whatever code is retained. The two are independent and composable, since a slice can still be rendered verbosely.

Binary type inference and variable-name recovery, from constraint-based methods [28, 37] to neural predictors [8, 26, 56], report substantial residual uncertainty, recovering the original developer name only ∼56–66% of the time even with dedicated models [8, 55]. T3 builds on this, since the default identifiers it rewrites, like `local_18` or `uVar3`, are *mechanically derived* from stack offsets and SSA indices rather than authored. Compressing

them therefore discards little signal, a claim we verify in Section 3.2. Within names and types, the genuine signal is the part anchored to a real symbol, and symbol stripping is known to cause large semantic loss [24]. This is exactly why the lossy tiers preserve symbol-anchored names and compress only the speculative ones.

6 Discussion

Taken together, our results carry implications beyond the raw token counts. They indicate that much of the redundancy in decompiler output can be removed without harm, beyond a small, margin-bounded decline on the most compression-sensitive task. The tokens that compress away tend to be the ones that carry the least task-relevant signal.

First, the savings profile in Section 3.1 shows that the redundancy sits in two distinct places, one addressed by the semantics-preserving tiers and the other by the lossy tiers. The semantics-preserving tiers remove the cost of human-oriented rendering, whitespace, layout, and address comments, while the lossy tiers compress the machine-generated placeholder names and verbose types that dominate stripped output. Which tier captures the most depends on the decompiler, with Binary Ninja gaining most from the semantics-preserving tiers and Ghidra, denser in placeholders, gaining most at T3. Which tier to use, in turn, depends on the application and task. We argue that the semantics-preserving tiers are largely unconditional because they change almost only the rendering. The lossy tiers instead target tasks where machine-generated names and verbose types carry little signal, so a task that turns on exactly that material, such as recovering a calling convention, is where T4 should be left off. Across our evaluations in Section 3.2, the lossy tiers preserve quality on summarization and variable naming and stay within the margin on function naming, where a small monotone decline appears. This holds because a placeholder name is derived mechanically from a stack offset or an SSA index, encoding the decompiler's bookkeeping, not anything the author intended or execution requires.

Second, the same opportunity is not specific to decompilation. Many representations produced for a non-LLM consumer, such as assembly listings, compiler IR dumps, or AST pretty-prints, carriy structure no tokenizer needs. Decompiler output is simply the case where that consumer has become an LLM often enough to matter.

Third, the deeper point is that decompiler output was never built for the model now reading it. Its formatting optimizes readability, whereas what is legible to a tokenizer and useful to a model is a different objective. Deflate-D pursues one axis of that objective, efficiency, shrinking the input to cut spend, latency, and context pressure by removing what carries little task-relevant signal. A second axis, output quality, runs in the other direction, since artifacts that carry wrong or low-confidence signal can mislead the model rather than merely inflate it. Marking speculative casts and guessed types, surfacing decompiler confidence, and normalizing the patterns the model reasons about could therefore raise accuracy rather than only cut tokens.

In practice, the efficiency axis probably pays off most in the agentic setting. Section 5 notes that production reverse-engineering assistants send decompiler output to a model in interactive and agentic loops, often through MCP servers, where the same function recurs across prompts. The per-prompt saving therefore compounds rather than being paid once. That setting also reframes the deferred cost T3 introduces, since a task that later needs the original names must carry the rename map alongside the rendering. In the agentic loop, though, that map is carried once per binary and reused across every call that touches it, so it amortizes in exactly the regime where savings accumulate.

Finally, this raises the question of where such an optimization should eventually be integrated. One option is the decompiler itself, as an alternative rendering offered alongside the human-facing one. Its advantage is access, since every major decompiler already records which names and types it recovered and which it guessed, namely Ghidra's symbol *source type* [14], Hex-Rays' `TINFO_GUESSED` versus `TINFO_DEFINITE` flag [16], and Binary Ninja's 0–255 confidence score [50]. An integrated reformatter could therefore gate on that metadata directly, where a standalone tool only approximates it from spelling. Its drawback is coupling, since each decompiler has its own release cycle and internals. The other option is the MCP servers that already mediate decompiler output to agents [6, 10, 27, 35], whose advantage is independence, since one implementation reformats whatever any decompiler emits. Their drawback is that a server sees only rendered text and must rederive what the decompiler knew. Rather than choosing, we expect the work to divide along the semantics-preserving/lossy line, with the pure-text semantics-preserving reformatting fitting the MCP layer and the lossy tiers needing the confidence metadata only the decompiler holds.

7 Conclusion

Much of what a decompiler emits is its own synthesis rather than recovered signal, and because that rendering is regenerated on demand rather than maintained as source, it can be compressed far more aggressively than source-code reformatters allow. We turn this observation into Deflate-D, a deterministic, model-free reformatter whose transformations form four cumulative tiers ordered by information loss, two semantics-preserving and two lossy, where the lossy tiers drop only machine-generated, speculative content. The implementation focuses on the three dominant decompilers, Ghidra, IDA Pro (Hex-Rays), and Binary Ninja. Across 109 GNU coreutils binaries the semantics-preserving tiers alone remove a fifth to a third of all input tokens and the lossy tiers reach up to 61%, at negligible measurement cost and with no model in the loop. On three short, function-level binary-analysis tasks with a single frontier model, these savings leave task quality non-inferior within a margin fixed in advance. The tokens Deflate-D discards are therefore largely not the ones these tasks relied on, except for a small decline on function naming. Whether the same holds for larger tasks that reason across a whole binary, other models, or other decompilers is left open.

The wider point is that decompiler output is now read by a model as often as by an analyst, yet was never built for that reader. Reducing its tokens is one axis of designing it for the model, and raising the quality of its signal is the complementary axis reduction alone cannot reach. We release Deflate-D [11] and our benchmark [12] so this trade-off can be re-measured on new tasks, and decompiler output can be studied as a target in its own right.

References

[1] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. 2023. Extending Source Code Pre-Trained Language Models to Summarise Decompiled Binaries. In *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 260–271. doi:10.1109/SANER56733.2023.00033

[2] Anthropic. 2026. Claude Developer Platform Pricing. https://www.anthropic.com/pricing. Claude Opus 4.8 list price $5 per million input tokens and $25 per million output tokens; retrieved June 2026.

[3] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation. In *Proc. of the 33rd USENIX Security Symposium*.

[4] Zion Leonahenahe Basque, Samuele Doria, Ananta Soneji, Wil Gibbs, Adam Doupé, Yan Shoshitaishvili, Eleonora Losiouk, Ruoyu Wang, and Simone Aonzo. 2026. Decompiling the Synergy: An Empirical Study of Human–LLM Teaming in Software Reverse Engineering. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. doi:10.14722/ndss.2026.240380

[5] Eva-Maria C. Behner, Steffen Enders, and Elmar Padilla. 2025. SoK: No Goto, No Cry? The Fairy Tale of Flawless Control-Flow Structuring. In *Proc. of the 2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P)*.

[6] Bethington. 2026. Ghidra MCP: Tools for AI-Powered Reverse Engineering. https://github.com/bethington/ghidra-mcp. Open-source software.

[7] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proc. of the 22nd USENIX Security Symposium*.

[8] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting Decompiler Output with Learned Variable Names and Types. In *Proc. of the 31st USENIX Security Symposium*. 4327–4343. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin

[9] Steffen Enders, Eva-Maria C. Behner, Niklas Bergmann, Mariia Rybalka, Elmar Padilla, Er Xue Hui, Henry Low, and Nicholas Sim. 2023. dewolf: Improving Decompilation by leveraging User Surveys. In *Workshop on Binary Analysis Research (BAR), co-located with NDSS*.

[10] fosdickio. 2025. Binary Ninja MCP: A Model Context Protocol Server for Binary Ninja. https://github.com/fosdickio/binary_ninja_mcp. Open-source software.

[11] Fraunhofer FKIE. 2026. Deflate-D: A Deterministic, Model-Free Reformatter for Token-Efficient Decompiler Output. Source code (MIT license); https://github.com/fkie-cad/deflate-d. software.

[12] Fraunhofer FKIE. 2026. Deflate-D: Evaluation Harness and Benchmark. Evaluation code; https://github.com/fkie-cad/deflate-d-evaluation. software.

[13] Zeyu Gao, Yuxin Cui, Hao Wang, Siliang Qin, Yuanda Wang, Bolun Zhang, and Chao Zhang. 2025. DecompileBench: A Comprehensive Benchmark for Evaluating Decompilers in Real-World Scenarios. In *Findings of the Association for Computational Linguistics: ACL 2025*. 23250–23267. doi:10.18653/v1/2025.findings-acl.1194

[14] Ghidra. 2025. Ghidra API: `SourceType` (symbol/type provenance). https://ghidra.re/ghidra_docs/api/ghidra/program/model/symbol/SourceType.html. API documentation.

[15] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *Proc. of the 15th ACM Asia Conf. on Computer and Communications Security (ASIA CCS)*. doi:10.1145/3320269.3384766

[16] Hex-Rays. 2025. Hex-Rays/IDA API: `TINFO_GUESSED` vs. `TINFO_DEFINITE`. https://python.docs.hex-rays.com/ida_typeinf/. API documentation.

[17] Hex-Rays. 2026. IDA Pro and the Hex-Rays Decompiler. https://hex-rays.com/ida-pro/. Version 9.3.

[18] Nicolas Hrubec and Jürgen Cito. 2026. Reducing Token Usage of State-in-Context Agents using Minification. In *Proc. of the 34th IEEE/ACM Int. Conf. on Program Comprehension (ICPC)*. doi:10.1145/3794763.3798174

[19] Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. DeGPT: Optimizing Decompiler Output with LLM. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[20] Nasir Hussain, Haohan Chen, Chanh Tran, Philip Huang, Zhuohao Li, et al. 2025. VulBinLLM: LLM-powered Vulnerability Detection for Stripped Binaries. arXiv:2505.22010.

[21] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. In *Proc. of the 2023 Conf. on Empirical Methods in Natural Language Processing (EMNLP)*. 13358–13376. doi:10.18653/v1/2023.emnlp-main.825

[22] Hailong Jiang, Jianfeng Zhu, Yao Wan, Bo Fang, Hongyu Zhang, Ruoming Jin, and Qiang Guan. 2025. Can Large Language Models Understand Intermediate Representations in Compilers?. In *Proc. of the 42nd Int. Conf. on Machine Learning (ICML) (Proceedings of Machine Learning Research, Vol. 267)*. PMLR, 27851–27872. https://proceedings.mlr.press/v267/jiang25p.html

[23] Linxi Jiang, Xin Jin, and Zhiqiang Lin. 2025. Beyond Classification: Inferring Function Names in Stripped Binaries via Domain Adapted LLMs. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[24] Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary Code Summarization: Benchmarking ChatGPT/GPT-4 and Other Large Language Models. arXiv:2312.09601.

[25] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. SymLM: Predicting Function Names in Stripped Binaries via Context-Sensitive Execution-Aware Code Embeddings. In *Proc. of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. doi:10.1145/3548606.3560612

[26] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *Proc. of the 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*. 628–639. doi:10.1109/ASE.2019.00064

[27] LaurieWired. 2025. GhidraMCP: A Model Context Protocol Server for Ghidra. https://github.com/LaurieWired/GhidraMCP. Open-source software.

[28] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[29] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. In *Proc. of the 34th USENIX Security Symposium*. 489–507. https://www.usenix.org/conference/usenixsecurity25/presentation/lekssays

[30] Yinxi Li, Yuntian Deng, and Pengyu Nie. 2026. TokDrift: When LLM Speaks in Subwords but Code Speaks in Grammar. In *Proc. of the Annual Meeting of the Association for Computational Linguistics (ACL)*. To appear; arXiv:2510.14972.

[31] Zongqian Li, Yinhong Liu, Yixuan Su, and Nigel Collier. 2025. Prompt Compression for Large Language Models: A Survey. In *Proc. of the 2025 Conf. of the Nations of the Americas Chapter of the ACL (NAACL): Human Language Technologies (Volume 1: Long Papers)*. 7182–7195. doi:10.18653/v1/2025.naacl-long.368

[32] Zhe Li and Xinxi Lu. 2025. Research on Compressed Input Sequences Based on Compiler Tokenization. *Information* 16, 2 (2025), 73. doi:10.3390/info16020073

[33] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, et al. 2025. LLM-Powered Static Binary Taint Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 34, 3, Article 83 (2025), 36 pages. doi:10.1145/3711816

[34] Ahmed Mostafa, Raisul Arefin Nahid, and Samuel Mulder. 2025. How Different Tokenization Algorithms Impact LLMs and Transformer Models for Binary Code Analysis. In *Workshop on Binary Analysis Research (BAR), co-located with NDSS*. arXiv:2511.03825.

[35] mrexodia. 2025. IDA Pro MCP: A Model Context Protocol Server for IDA Pro. https://github.com/mrexodia/ida-pro-mcp. Open-source software.

[36] National Security Agency. 2026. Ghidra Software Reverse Engineering Framework. https://ghidra-sre.org/. Version 12.1.2.

[37] Matthew Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *Proc. of the 37th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*. 27–41. doi:10.1145/2908080.2908119

[38] OpenAI. 2024. tiktoken: A Fast BPE Tokeniser for use with OpenAI's Models. https://github.com/openai/tiktoken. Version 0.13.0, pinned for reproducible counts.

[39] Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, David Lo, and Xiaoning Du. 2026. The Hidden Cost of Readability: How Code Formatting Silently Consumes Your LLM Budget. In *Proc. of the 48th IEEE/ACM Int. Conf. on Software Engineering (ICSE)*. arXiv:2508.13666.

[40] Zhuoshi Pan, Qianhui Wu, Huiqiang Jiang, et al. 2024. LLMLingua-2: Data Distillation for Efficient and Faithful Task-Agnostic Prompt Compression. In *Findings of the Association for Computational Linguistics: ACL 2024*. 963–981. doi:10.18653/v1/2024.findings-acl.57

[41] Salem Radey, Jack West, and Kassem Fawaz. 2026. Challenges and Future Directions in Agentic Reverse Engineering Systems. In *Proc. of the 2nd Int. Workshop on Software Architecture and Generative AI (SAGAI), co-located with IEEE ICSA*. arXiv:2604.14317.

[42] Xiuwei Shang, Zhenkan Fu, Shaoyin Cheng, Guoqiang Chen, Gangyang Li, Li Hu, Weiming Zhang, and Nenghai Yu. 2026. An Empirical Study on the Effectiveness of Large Language Models for Binary Code Understanding. *Empirical Software Engineering* 31, Article 14 (2026). doi:10.1007/s10664-025-10748-5

[43] Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2025. LongCodeZip: Compress Long Context for Code Language Models. In *Proc. of the 40th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*. 141–153. doi:10.1109/ASE63991.2025.00020

[44] Francesco Sovrano, Adam Bauer, and Alberto Bacchelli. 2025. Large Language Models for In-File Vulnerability Localization Can Be "Lost in the End". In *Proc. of the ACM Int. Conf. on the Foundations of Software Engineering (FSE)*. 891–913. doi:10.1145/3715758

[45] Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2024. AI Coders Are Among Us: Rethinking Programming Language Grammar Towards Efficient Code Generation. In *Proc. of the 33rd ACM SIGSOFT Int. Symp. on Software Testing and Analysis (ISSTA)*. 1124–1136. doi:10.1145/3650212.3680347

[46] Zhensu Sun, Xiaoning Du, Zhou Yang, Li Li, and David Lo. 2025. Token Sugar: Making Source Code Sweeter for LLMs through Token-Efficient Shorthand. In *Proc. of the 40th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*. 2440–2451. doi:10.1109/ASE63991.2025.00201

[47] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. In *Proc. of the 2024 Conf. on Empirical Methods in Natural Language Processing (EMNLP)*. 3473–3487. doi:10.18653/v1/2024.emnlp-main.203

[48] Jason Tang. 2024. GhidrAssist: An LLM-assisted Reverse Engineering Plugin for Ghidra. https://github.com/symgraph/GhidrAssist. software.

[49] Muhammad Umair, Genwei Jiang, and Mustafa Nasser. 2024. XRefer: The Gemini-Assisted Binary Navigator. Mandiant/Google Cloud Threat Intelligence blog; https://github.com/mandiant/xrefer. tool and blog post.

[50] Vector 35. 2025. Binary Ninja API: Type Confidence. https://api.binary.ninja/cpp/group__confidence.html. API documentation.

[51] Vector 35. 2026. Binary Ninja. https://binary.ninja/. Version 5.3.

[52] Guilherme Venere. 2025. Using LLMs as a Reverse Engineering Sidekick. https://blog.talosintelligence.com/using-llm-as-a-reverse-engineering-sidekick/. Cisco Talos Intelligence blog.

[53] Yan Wang, Ling Ding, Tien N. Nguyen, Shaohua Wang, and Yanan Zheng. 2025. LeanCode: Understanding Models Better for Code Simplification of Pre-trained Large Language Models. In *Proc. of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL): Volume 1, Long Papers*. 1551–1567. doi:10.18653/v1/2025.acl-long.78

[54] Yongpan Wang, Xin Xu, Xiaojie Zhu, Xiaodong Gu, and Beijun Shen. 2025. SALT4Decompile: Inferring Source-level Abstract Logic Tree for LLM-Based Binary Decompilation. arXiv:2509.14646.

[55] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *Proc. of the 2024 ACM SIGSAC Conf. on Computer and Communications Security (CCS)*. doi:10.1145/3658644.3670340

[56] Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, et al. 2025. Unleashing the Power of Generative Model in Recovering Variable Names from Stripped Binary. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*. https://www.ndss-symposium.org/ndss-paper/unleashing-the-power-of-generative-model-in-recovering-variable-names-from-stripped-binary/

[57] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

Generative AI Usage

The authors used a large language model, Anthropic's Claude, as a tool throughout the preparation of this work. Its use spanned three areas: writing and editing the text of the paper, implementing and refactoring DEFLATE-D and its evaluation harness, and preparing the tables and figures. The research design, the claims, and the conclusions are the authors' own, and all AI-assisted output was treated as a draft to be checked rather than as finished material. We validated it accordingly: every passage of text was read, edited, and confirmed by the authors; the implementation is exercised by an automated test suite and its transformations were inspected on real decompiler output; the reported measurements were produced by re-running our own experiments; and every citation was checked against its original source.