BIN2WRONG: a Unified Fuzzing Framework for Uncovering Semantic Errors in Binary-to-C Decompilers

Zao Yang
University of Utah

Stefan Nagy
University of Utah

Abstract

Binary *decompilation* is central to many systems tasks that rely on analyzing or modifying closed-source software, such as debugging, performance tuning, and security hardening. Decompilers translate executables into C code with the goal of reconstructing a semantically-equivalent form of the original program's source. Unfortunately, when challenged by intricate program logic, data structures, and diverse executable layouts, decompilers often produce semantically-*wrong* code. Proactively detecting such decompilation defects is critical for ensuring the success of downstream tasks that depend on precise binary analysis. Yet, current methods for assessing decompiler correctness only narrowly explore the variety of source constructs, compilers, optimization levels, executable formats, and *combinations thereof* that influence binary code. Fully guaranteeing decompilation precision—and, by extension, supporting all tasks that hinge on accurate binary-to-source recovery—demands a testing approach that unifies *all* factors affecting binary code, extending practical, systematic correctness testing to all decompilers today.

To accelerate discovery of decompilation defects, this paper introduces BIN2WRONG: a general-purpose decompiler fuzzer combining systematic binary mutation with practical, decompiler-agnostic support. Our approach coalesces *all* factors of binary generation—source, compiler, optimization, and executable format—into a novel, unified testcase structure for mutation. Beyond enabling deeper exploration along these individual dimensions, BIN2WRONG finds unique *combinations* exposing complex, multi-dimensional errors that elude prior decompiler testing approaches. In evaluating BIN2WRONG alongside state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer across seven free and commercial decompilers, BIN2WRONG achieves upwards of **10.39**× and **17.18**× higher binary diversity and **1.16**× and **1.32**× more decompiler code coverage, respectively, whilst uncovering **the most** decompilation bugs. Beyond finding **48 *new* bugs**, **with 30 confirmed**, BIN2WRONG spurred a major redesign of the commercial decompiler Binary Ninja—showing its utility in **uncovering critical defects in mainstream decompilers**.

1 Introduction

Binary *decompilers* are crucial for analyzing, maintaining, and modifying modern systems software, translating raw machine code into more human-readable higher-level representations, most commonly C. By bridging this gap, decompilers become indispensable for tasks that must proceed without a program's original source, such as performance tuning of closed-source components [37], malware analysis [136], security assessment of proprietary kernels [34], and third-party patching of production systems [40]. Popular decompilers today include Binary Ninja [108], Angr [132], Ghidra [78], and RetDec [6].

Decompilers' ultimate goal is recovering code identical or semantically-equivalent to the program's original source. Recent advancements are vastly improving decompilation accuracy through neural networks [30, 141], LLMs [5, 135], and heuristic code restructuring [15, 33]. Yet in practice, decompilers face steep challenges due to the lossy nature of compilation, which strips source-level semantics like type information, function and variable names, and high-level program structures. Moreover, diverse executable formats (e.g., ELF, PE, Mach-O), compiler-specific code layout patterns (e.g. variable arrangement, function ordering), and optimizations (e.g., inlining, loop jamming, dead code elimination) further impede precise decompilation, requiring significant specialized handling. As even subtle mistakes (e.g., a misrecovered branch condition) cause trickle-down failures (e.g., incorrect control flow) in downstream binary analysis tasks, **vetting decompilers' correctness is critical to the success of many systems engineering and reliability efforts today**.

Inspired by the success of automated testing—namely fuzzing—in other software domains, recent works propose testing strategies specially tailored to finding semantic errors in decompilers [20, 54, 89, 142]. While such approaches have historically uncovered bugs, they are overwhelmingly ineffective at finding the vast majority of decompiler defects today. Executables are influenced by four key dimensions: (1) *source*, the fundamental blueprint behind binary generation; (2) *compilers*, which emit and structure the bi-

nary's code; (3) *optimizations*, which restructure, expand, or trim the binary accordingly; and (4) the *executable format*, which encapsulates platform-specific code layout and calling conventions. Yet, despite the equal importance of each of these aspects, current decompiler fuzzers fail to explore them comprehensively—hardcoding themselves to only specific, narrow subsets that ultimately highly overlap with those already-covered by past testing efforts. Worse yet, these restrictive approaches cannot reveal the many decompiler bugs caused by specific source and compilation parameter *combinations*—leaving these edge-case errors unlikely to be found until they inevitably confront decompilers' own users. Thus, preventing the failure of critical efforts reliant on precise decompilation demands **a systematic correctness-testing approach spanning *all* aspects of binary generation—whilst supporting all of today's diverse decompiler ecosystem**.

To accelerate discovery of decompiler errors producing semantically-wrong code, this paper introduces BIN2WRONG: a general-purpose fuzzer that explores *all* aspects of binary generation—source, compiler, optimizations, and executable formats—in a synergistic *and* decompiler-agnostic manner. Unlike prior decompiler fuzzers that view these dimensions individually, BIN2WRONG unifies and mutates them *altogether*, facilitating more flexible and thorough exploration of diverse binary semantics as well as the unique interplay arising from distinct *combinations* of all four factors. When applied to state-of-the-art decompilers, **BIN2WRONG reveals many more decompilation errors** than prior overly-restrictive approaches, assisting decompiler developers in strengthening their tools against complex edge-cases that otherwise derail their users with real-world inaccuracies.

We evaluate BIN2WRONG alongside the state-of-the-art decompiler fuzzers DecFuzzer [54] and Cornucopia [89] on seven free and industrial x86 decompilers: Binary Ninja, Angr, Radare2/Ghidra, Reko, Relyze, RetDec, and Rev.Ng. Overall, BIN2WRONG's holistic, binary-centric mutation strategy vastly outperforms DecFuzzer and Cornucopia in generated binary diversity by upwards of **10.39**× and **17.18**×, respectively. When used to test decompilers, BIN2WRONG's higher-diversity binaries achieve a mean **1.16**× and **1.32**× decompiler code coverage relative to DecFuzzer and Cornucopia—while uncovering **38** and **48** more decompilation semantic errors, respectively. After analyzing BIN2WRONG's **48 newly-found bugs** and reporting them to their decompilers' developers, **30 are so far confirmed or fixed**. Importantly, BIN2WRONG *uniquely* discovered many critical bugs, including one that prompted **a total overhaul** of the industrial decompiler Binary Ninja's core code structuring techniques.

In summary, our paper makes the following contributions:

- **Insights:** We analyze and categorize 64 publicly-reported decompiler semantic defects; and are the first to shed light on the importance of *all four* factors behind binary generation—source, compiler, optimization, and executable format—in facilitating decompiler defect discovery.
- **Challenges:** We weigh the limitations of prior decompiler testing efforts in considering our aforementioned four factors of binary generation; as well as their capability of supporting *all* of today's diverse decompiler ecosystem spanning both free and industrial decompilers.
- **Solutions:** We design BIN2WRONG: the first general-purpose decompiler fuzzer to balance automated, large-scale exploration of diverse binary semantics stemming from distinct source code and compilation parameter combinations—in a fully decompiler-agnostic manner. Beyond supporting virtually all of C's control and data constructs (e.g., floating-point and string literals, `switch-case` statements, `goto` branches, etc.), BIN2WRONG integrates six different C compilers, three binary executable formats, and 5,183 distinct compiler optimizations.
- **Results:** In an evaluation of BIN2WRONG versus state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer, we show that BIN2WRONG achieves the highest-overall binary diversity, code coverage, and semantic bug discovery across seven mainstream binary-to-C decompilers: Binary Ninja, Angr, Radare2/Ghidra, Reko, Relyze, RetDec, and Rev.Ng. Moreover, while prior decompiler testing works explore only limited sets of compilers and optimizations, we present the *first* comprehensive study of the distinct impacts of varying source code—combined with the largest-ever tested set of compilers, executable formats, optimizations, and *their combinations*—on the discovery of semantic errors within contemporary binary decompilers.
- **Impacts:** We report all discovered bugs to their respective decompiler developers. So far, **30** of **48** BIN2WRONG-found bugs are confirmed, with many since fixed. Several BIN2WRONG-uncovered bugs motivated significant changes to mainstream decompilers, including a major restructuring of the commercial decompiler Binary Ninja.

We will publicly release BIN2WRONG, as well as all evaluation artifacts and uncovered bugs at the following URL: `https://github.com/FuturesLab/Bin2Wrong`.

2 Background

This section provides a brief overview of the process of binary decompilation, as well as the most common classes of correctness defects affecting today's various binary decompilers.

2.1 An Overview of Binary Decompilation

Decompilers' goal is *inverting* program compilation: consuming a pre-built executable and recovering its source code [22]. For systems, decompilers critically facilitate human understanding and reasoning about software whose source is unavailable, such as legacy or proprietary components, and obfuscated malware. In the following, we outline and discuss the four main steps central to virtually all decompilers today: **(1) Control Flow Recovery**, **(2) Control Flow Analysis**, **(3) Data Flow Analysis**, and **(4) Code Generation**.

1. **Control Flow Recovery.** Control Flow Recovery aims to understand programs' underlying flow of execution. Instruction disassemblers (e.g., Capstone [3], Intel XED [39]) are used to translate raw binary code into machine instructions—and further—group them into *basic blocks*: sequences of consecutive instructions both targeted-by and terminating-in a *control-flow transfer* (i.e., branches, fall-throughs, calls, or returns). Additional analyses are performed to group basic blocks into functions, culminating in the overall program Control-flow Graph (CFG), where nodes and edges denote basic blocks and control-flow transfers between them, respectively.
2. **Control Flow Analysis.** With the CFG in-hand, decompilers begin abstracting basic block groupings into higher-level *control structures* like loops, switch-cases, and dispatch tables. Critical to CFG analysis is recognizing high-level structures via known patterns: for example, standard techniques for loop identification leverage control-flow dominance between basic blocks to determine the loop's overarching header [90]. Ultimately, recovery of higher-level control structures offers a richer understanding of program semantics, forming the backbone of decompilers' subsequent data flow recovery and code generation steps.
3. **Data Flow Analysis.** Following control structure recovery, decompilers shift focus to examining how *data* is read, written, and propagated throughout program execution. Primary goals of these analyses include identifying distinct data objects, such as function arguments, return values, and both local and global variables, along with determining their associated data *types*. These elements are critical to decompilers' final code generation phase as they directly impact virtually all aspects of program semantics: control-flow logic, data expressions, and more.
4. **Code Generation.** Decompilers' final stage leverages all collected control- and data-flow artifacts to generate the program's final *source* representation. Yet, as all prior steps are inherently undecidable, decompilers' generated code is unlikely to be identical to the program's original source. To this end, decompilers still aim to yield semantically-equivalent source representations which—while visually different from the original—*behaves* the same.

2.2 Semantics-affecting Decompilation Defects

Decompiler defects range from cosmetic inconsistencies affecting human readability of decompiled code—for example, haphazard ordering of `switch-case` labels (Binary Ninja issue [113])—to **insidious *semantics-affecting* errors that completely misrepresent program behavior**. Semantics-affecting defects uniquely burden decompiler users with *incorrect* results, thus impeding the success of downstream systems-oriented binary analysis tasks. To better understand these obstacles, we review public issue trackers of mainstream decompilers and distill reported bugs into four high-level categories of semantics-affecting defects shown in Table 1: **(1) Data Expressions**, **(2) Control-flow Logic & Structures**, **(3) Data Objects, Types, & Values**, and **(4) Compilation-specific Defects**. We briefly discuss these below.

Category	Origin(s)	Decompilers and Examples of Relevant Defects
Data Expression	Instructions	**Ghidra** [75], **Radare2** [82], **Reko** [48], **RetDec** [7, 13]
	Refinement	**Angr** [95, 106], **Binary Ninja** [120, 121, 125], **RetDec** [12]
Control Logic, Structures	If / Else	**Binary Ninja** [114, 126, 130], **Ghidra** [65], **RetDec** [8, 9]
	Loops	**Angr** [94, 99, 103], **Binary Ninja** [112, 116], **Reko** [47]
	Goto	**Angr** [104, 105], **Binary Ninja** [122], **Reko** [44], **RetDec** [11]
	Switches	**Binary Ninja** [111, 115, 116], **Radare2** [84], **RetDec** [10]
Objects, Types, Values	Arguments	**Angr** [93, 101, 102], **Binary Ninja** [117, 123, 127, 129]
	Variables	**Angr** [97, 100], **Binary Ninja** [124], **Ghidra** [72–74]
	Literals	**Angr** [92], **Binary Ninja** [119], **Ghidra** [66, 77], **Reko** [46]
Compile Configs	Code Opts	**Angr** [91, 93, 98], **Binary Ninja** [110], **Ghidra** [71, 76]
	Format	**Angr** [96], **Binary Ninja** [118, 128, 131], **Ghidra** [68, 70]

Table 1: Reported semantic bugs in mainstream decompilers.

- **Defects in Recovering Data Expressions.** Errors in instruction disassembly or handling of instruction semantics often lead to data inconsistencies in decompilers' higher-level code generation. For example, in Ghidra [75], incorrect handling of x86's `FCOMP` (floating-point comparison) results in comparison operands being *swapped*; and as this comparison governs a conditional branch, the code guarded by this branch ends up only being reachable under conditions totally different from the original program's. Additionally, decompilers often refine data expressions using techniques such as constant folding and propagation to enhance code readability [33], conciseness [137], or better align with real-world source code patterns [15]. However, subtle errors in these optimizations also lead to incorrect data. For example, in Binary Ninja [121], constant propagation for x86's `ROL` (rotate-left) and `SHL` (shift-left) instructions inaccurately simulates 32-bit registers, yielding divergent expression results when analyzed on 64-bit systems.
- **Defects in Recovering Control Logic & Structures.** In addition to data expressions, we identify numerous instances of semantics-affecting defects in decompilers related to the recovery of control logic and structures. These include missing bodies for conditional branches (e.g., RetDec [9]); incorrect recovery of loop starting conditions (e.g., Reko [47]); and missing, spurious, or misdirected `goto` branches (e.g., Angr [105]). Handling of `gotos` is particularly contentious—and error-prone—in decompilation research and design, with current decompilers employing vastly different strategies for their recovery and structuring (e.g., [15, 137]). Furthermore, we see that `switch-case` statements remain one of the most universally-challenging control structures across all decompilers, leading to downstream errors that result in unrecoverable code blocks (e.g., Radare2 [84], RetDec [10]), or complicating recovery of other intermingled control logic (e.g., Binary Ninja [116]).
- **Defects in Recovering Data Objects, Types, & Values.**

While recovery and refinement of data expressions pose challenges for decompilers, even identifying and reconstructing their underlying *data objects* (e.g., string, integer, and floating-point variables and literals)—along with their corresponding types and values—presents significant difficulties that directly impact the accuracy of decompiled code semantics. We observe many instances of functions missing arguments (e.g., Angr [101], Binary Ninja [127]) or return values (e.g., Angr [100]), as well as general type incompatibilities on local variables (e.g., Binary Ninja [124], Ghidra [74]). Strings and floating-point literals also present distinct challenges for decompilers, which commonly miss (e.g., Binary Ninja [119]), truncate (e.g., Ghidra [77]), or otherwise erroneously perturb them (e.g., Reko [45]).

- **Defects from Specific Compilation Configurations.** Lastly, we see many issues stemming from *compilation* diversity, such as function recovery defects specific to MacOS Mach-O (e.g., Angr [96], Ghidra [68]) and Windows PE binaries (e.g., Binary Ninja [128]). Additional challenges arise under certain compiler optimizations; for instance, Ghidra [71] and Angr [93] face difficulties with calling convention recovery under MSVC's `/LTCG` (link-time code generation) and GCC's `-Ofast` optimizations, respectively. Furthermore, we see source constructs present unique semantic recovery challenges depending on the executable format they are compiled into, as demonstrated by Binary Ninja's and Ghidra's errors on floating-point function arguments from PE binaries ([69] and [70], respectively).

As binary-oriented systems tasks such as malware analysis, vulnerability discovery, and security hardening all depend on *accurate* decompilation, decompiler developers are working aggressively to fix errors and prevent misleading their users with inaccuracies or failures. However, the lack of a systematic testing approach leaves today's decompiler developers overwhelmingly relying on *manual* bug discovery (e.g., Table 1), limiting their ability to proactively identify critical, semantics-affecting defects. Enhancing the reliability of decompilers thus demands moving beyond cumbersome manual bug-finding and adopting **automated, *practical* testing techniques that thoroughly explore the complex interplay between source *and* compilation diversity** in binary code.

3 Challenges of Testing Decompilers

To automate discovery of semantics-affecting decompiler defects, a handful of decompiler testing efforts have emerged over the years (Table 2): Cornucopia [89], DecFuzzer [54], D-Helix [142], and DSmith [20]. Unfortunately, as we show below, these approaches are fundamentally limited in achieving systematic, practical decompiler defect discovery. In the following, we weigh their trade-offs with respect to addressing the three main challenges of automated decompiler defect discovery: **(1) source construct diversity**, **(2) compilation diversity**, and **(3) supporting *all* decompilers**.

Approach	Source Diversity			Compilation Diversity					Decomp.
	Exprs	Ctrl	Data	Comp	Opts	ELF	PE	Mach-O	Agnostic
DecFuzzer	✔	👎	👎	1	✗	✔	✗	✗	✔
Cornucopia	✔	✔	✔	2	✔	✔	✗	✗	✔
D-Helix	✔	✔	👎	2	👎	✔	✗	✗	✗
DSmith	✔	👎	👎	1	👎	✔	✗	✗	✔
BIN2WRONG	✔	✔	✔	6	✔	✔	✔	✔	✔

Table 2: Survey of state-of-the-art decompiler testing works. ✔ = full support; ✗ = no support; 👎 = limited or poor support.

3.1 Challenge 1: Source Construct Diversity

Varying source constructs pose unique hurdles for decompilation, complicating reconstruction of high-level program logic and structure. For example, both `switch-case` and `if-else` statements are often compiled-down into jump tables [80], challenging decompilers' differentiation of the two. To this end, modern decompilers devote significant analyses and heuristics to recovery and refinement of distinct source constructs [50]. Moreover, decompilers frequently differ in handling specific constructs (e.g., `gotos` [15]), meaning that the same source often produces disagreeing results among decompilers. Thus, maximizing diversity of input source code is crucial to scrutinizing decompilers' underlying capabilities and accuracy in handling real-world binaries.

Limitations: In examining today's leading automated decompiler testing efforts (Table 2), we observe varying support for many of today's most common source code constructs. Although each considers control-flow like loops, conditional branches, function calls and returns, and `gotos`, both DSmith and DecFuzzer entirely omit `switch-case` statements, limiting their ability to thoroughly evaluate decompilers' control-flow recovery. Further, almost all exhibit limitations in supporting various types of literals: DecFuzzer, DSmith, and D-Helix all fail to support floating-point literals; while DecFuzzer and DSmith additionally exclude string literals because their underlying source code generation tooling does not yet support them [139]. Omitting common source constructs drastically reduces the scope of previous methods, **leaving them unable to catch many critical decompiler defects likely to emerge in practice**.

3.2 Challenge 2: Compilation Diversity

Modern compilers have unique code generation strategies and internal algorithms, creating vast differences in their resulting binary code. For example, virtual dispatch tables take on fundamentally disparate forms when compiled by MSVC versus GCC [14]. As a result, identical source overwhelmingly yields *different* binaries when compiled by multiple compilers. This entropy is further increased by compiler *optimizations*, which additionally obscure precise disambiguation of control and data constructs; for example, MSVC's aggressive data and code inlining [60]. Furthermore, platform-specific executable file formats—Windows' PE, Linux's ELF, and MacOS's Mach-O—have fundamentally unique binary layouts,

necessitating decompilers' specialized handling of each [25]. Because reverse engineering—and thus, decompilers—targets *every* software ecosystem, it is thus crucial to identify and characterize the kinds of errors that decompilers encounter across the vast space of possible compilation configurations.

Limitations: While prior decompiler testing works vary in the diversity of the source constructs they examine, they are even more restricted in their exploration of binary compilation parameters—**hardcoding themselves specific, hand-chosen compilation configurations, making them unable to automatically explore *others***. All four cover no more than *two* compilers: D-Helix and Cornucopia focus exclusively on GCC and Clang, while DSmith and DecFuzzer limit themselves to only GCC. In terms of optimizations, Cornucopia examines nearly all possible GCC and Clang optimization levels, whereas the others are more constrained: D-Helix is limited to `O0` and `O2-O3`, D-Smith explores `O0-O3` and `Os`, and DecFuzzer omits optimizations altogether. Additionally, these approaches consider only the Linux ELF executable format, entirely neglecting PE and Mach-O binaries. Moreover, their designs are rigid and hardcoded to specific compilation parameters, **preventing them from automatically exploring the diversity of binaries generated from a single input source code**—and the decompilation defects these variations might introduce—without additional effort and re-engineering.

3.3 Challenge 3: Broad Decompiler Support

Modern binary decompilers exhibit considerable diversity in their implementations, encompassing different programming languages, internal code representations, and degrees of transparency. While many are open-source (e.g., Angr [132], Ghidra [78], and Rev.Ng [24]), some of today's most widely-used decompilers are commercial products and hence remain fully or partially closed-source, limiting insight into their underlying internals (e.g., Binary Ninja [108], IDA Pro [36], and JEB3 [81]). Therefore, a comprehensive vetting approach is required that is supportive of *all* decompilers—regardless of their architectural design variations or transparency.

Limitations: Most previous approaches prioritize scalability by treating decompilers as black-box systems, analyzing only their output C code. In contrast, D-Helix operates directly on the *non-C* Intermediate Representations (IRs) used internally by decompilers prior to final code generation. Yet, such approach requires costly, *per-instruction semantic modeling*—further burdened by the vast differences between decompiler-specific IRs (e.g., Angr's `VEX` vs. Binary Ninja's `LLIL`, `MLIL`, and `HLIL` IRs). Currently, D-Helix only supports Ghidra's `P-code` and Angr's `VEX` IRs, leaving others (e.g., Rev.Ng's LLVM IR [24]) **beyond its reach**. According to its developers [142], extending D-Helix to *other* decompilers' IRs—which often differ significantly in semantics [42]—demands considerable manual effort and domain expertise on the order of **40 days' time per unique decompiler integration**.

Motivation: Thoroughly vetting binary decompilers requires a multifaceted testing approach that accommodates ***all* decompilers** whilst accounting for every factor influencing executable code generation. We propose a holistic view of binary mutation—**integrating diverse source code, multiple compilers, and varying optimizations and executable formats *simultaneously***—to maximize the detection of semantic decompilation defects across *different* decompiler platforms, irrespective of their internal differences or level of transparency.

4 BIN2WRONG: Systematic, Practical Fuzzing for Binary Decompilers

To tackle the limitations of prior decompiler testing approaches, we present BIN2WRONG (Figure 1): a general-purpose fuzzer for uncovering decompilation semantic bugs whilst supporting *any* decompiler—irrespective of internals or transparency. We detail our high-level design of BIN2WRONG's key components below.

4.1 Mutation of Binary-affecting Dimensions

Recent work shows [57, 133] fuzzing is most effective when mutation is *maximized*—randomly distributing mutations across a target program's *entire* input space rather than only along a small portion. Yet, while binaries are distinctly influenced by source, compilers, optimizations, and executable formats, **current decompiler testing approaches limit exploration to only a *small* part of this vast input space**. DecFuzzer [54] focuses on source alone, eluding optimizations entirely; and similarly, prior approaches [20, 89, 142] consider only GCC and/or Clang, excluding other compilers—and consequently—other binary formats like PE and Mach-O.

To overcome these limitations, we propose a holistic approach to decompiler-oriented fuzzing, where source code and compilation parameters are treated and mutated as a *single, unified* testcase. Below, we describe BIN2WRONG's overarching **unified binary testcase construction** (Figure 2), and explain how its components work together to enable a deeper exploration of more diverse binary semantics—and more effective decompiler fuzzing—than previously possible.

4.1.1 Unifying Source and Compilation Mutation

With AFL++ [29] came new possibilities in implementing domain-specific mutators through its versatile *custom mutator API* (e.g., [86, 140]). Building on these capabilities, we develop BIN2WRONG's decompiler-oriented binary mutation via a suite of custom AFL++ mutators targeting both source- and compilation-level modifications, detailed below.

4.1.2 Mutating Compilation

As § 3.2 shows, different compilers, formats, and optimizations produce unique binaries even from identical source, chal-

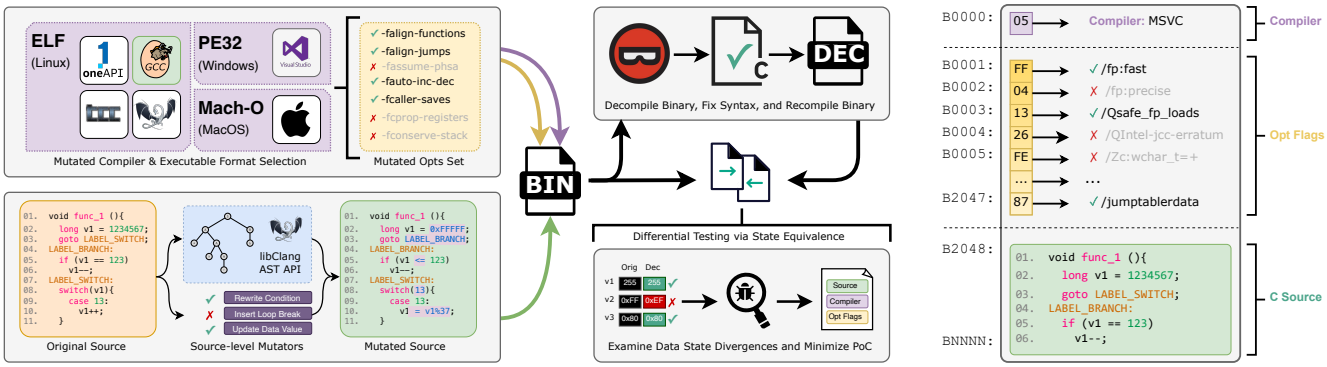

Figure 1: BIN2WRONG's high-level fuzzing workflow in a setup targeting Binary Ninja. Figure 2: Our unified testcase design.

lenging decompilers' recovery of higher-level code from varied binary structures (Table 1). Rather than restrictively *hard-code* compilation parameters like prior approaches (Table 2), BIN2WRONG reshapes compilation diversity as a fuzzing *mutation* technique: randomly selecting the compiler—and further—randomly toggling-on its available optimizations. Accordingly, BIN2WRONG's unified testcase construction (Figure 2) dedicates **the first 2048 bytes** to encode the fuzzer-mutated compilation parameters: byte `0` specifies the compiler, while bytes `1-2047` are each mapped to individual optimization flags specific to the chosen compiler. Upon selecting a testcase from its queue, BIN2WRONG modifies the compiler byte to one of its six supported compilers; while mutating optimization bytes to either *odd* or *even* numbers, denoting whether that optimization is enabled or disabled, respectively.

Compiler and Executable Format: Since BIN2WRONG is built atop the Linux-based fuzzer AFL++ v4.09c [29], it naturally supports Linux-native compilers such as GCC and Clang. In further maximizing compilation diversity across Linux's ELF binary format, we incorporate several mainstream Linux ELF compilers **unexplored by prior decompiler testing approaches**: Tiny C (TCC) [17], which aims to speeds-up compilation performance; as well as Intel's oneAPI DPC++ (ICX) [38], which uses advanced x86-specific optimizations.

Unfortunately, no Linux-based compilers support *non-*Linux formats MacOS's Mach-O and Windows' PE, leaving such approach unable to find their related bugs (e.g., Angr [96], Ghidra [71]). Yet, we see opportunities in cross-platform compatibility layers that run non-Linux code *on Linux* (e.g., WINE [2], Darling [26]). We leverage these tools to extend BIN2WRONG to non-ELF compilers—Apple's Xcode compiler (AppleClang) [4] for Mach-O, and Microsoft's Visual Studio Compiler (MSVC) [61] for PE—making BIN2WRONG **the first decompiler fuzzer spanning ELF *and* non-ELF formats *altogether***.

Optimization Selection: After selecting a compiler for the eventual binary, BIN2WRONG mutates its choice of the optimization options specific to that compiler, spanning a total of **1,029** total options for GCC v11.4.0, **1,378** for Clang v18.0.0, **7** for TCC v0.9.27, **1,174** for ICX v2024.0.2, **217** for MSVC v19.38.33134, and **1,378** for AppleClang v18.0.0. Should BIN2WRONG's compilation mutation later select a *different* compiler (e.g., MSVC→ ICX), BIN2WRONG remaps its 2,047 optimization bytes to the corresponding set for the *new* compiler. If incompatible optimizations result that prevent compilation, BIN2WRONG defaults to one of the compiler's standard optimizations (e.g., `-O1` to `-O3` for ICX), thus resolving 100% of optimization-related compile issues.

While BIN2WRONG's high-level optimization mutation is inspired by Cornucopia's [89], Cornucopia is built to explore only a single compiler's optimization suite at a time, and therefore cannot generalize across different compilers and formats in any one campaign. Comparatively, BIN2WRONG's ability to explore *multiple* compilers' optimizations simultaneously enables it to encompass **5,183** optimizations in total versus Cornucopia's 892 explored Clang-based optimizations—resulting in a far greater binary diversity from **BIN2WRONG's inclusion of both ELF and non-ELF compilers *and* their optimizations**.

Our Approach: Framing compilation diversity as a fuzzing mutation allows for **dynamic exploration of varied, compilation-dependent binary semantics**—enabling broader exploration than prior decompiler testing approaches that rely on hardcoded compilation parameters.

4.1.3 Mutating Source Code

Following compilation mutation, BIN2WRONG explores varied code semantics through its source-level mutators. We model BIN2WRONG's testcases after those in prior compiler [139] and decompiler fuzzers [54], each consisting of a single function targeted for source mutation. To minimize risk of syntax-breaking mutations, we use the LLVM [52] framework's libClang abstract syntax tree (AST) API [56] alongside our own suite of AFL++ custom mutators.

Informed by our survey of real-world semantic bugs (Table 1), we implement the following *expression*-level mutators to probe decompilers' recovery of complex data-affecting instruction sequences (e.g., Ghidra [75], Binary Ninja [121]):

- **Expressions:** Deleting an entire expression; or duplicating it immediately after the original one:
  e.g., `v0 = v1++;` → `v0 = v1++; v0 = v1++;`
- **Sub-expressions:** Deleting or duplicating sub-expressions; or expanding with an existing one:
  e.g., `v1/v4;` → `v1/v4 + (v1%v2);`
- **Operators:** Randomly replacing arithmetic/bitwise operators; or flipping increments/decrements:
  e.g., `(v1*v2)-(v3++);` → `(v1/v2)*(v3-);`

To test decompilers' recovery of complex control-flow logic (e.g., RetDec [9], Angr [105]), we additionally implement the following *control structure* mutators:

- **Loops:** Inserting a conditional `break` or `continue` with a random threshold at a random location:
  e.g., `for (i=0){}` → `for (i=0){ if(i>37) break; }`
- **Conditional Branches:** Replacing the condition with a always-`true` or always-`false` condition:
  e.g., `if(v2/v3 == 0){}` → `if(1){}`
- **Unconditional Branches:** Replacing a `goto` destination with a randomly-chosen existing label:
  e.g., `goto LABEL_BRANCH;` → `goto LABEL_SWITCH;`
- **Switch-Cases:** Replacing the `switch`'s expression with a randomly-selected existing case value:
  e.g., `switch(v1){ case 2: ... }` → `switch(2){...}`

Lastly, to find bugs stemming from imprecise data object, type, and value recovery (e.g., Binary Ninja [119], Ghidra [74]), we implement the following *data-level* mutators:

- **String Literals:** Replacing with a randomly-chosen string literal; or flipping one or more bytes:
  e.g., `char *v0="Global";` → `char *v0="0xFFFFal";`
- **Numerical Literals:** Replacing with a random number; or flipping its sign or one or more bytes:
  e.g., `(v1/55)+(0xFF*v2);` → `(v1/-55)+(0x00*v2)`
- **Numerical Typecasts:** Replacing with a random numerical type (e.g., `int`, `short`, `float`, `long`):
  e.g., `(short) v3/25);` → `(float) v3/25);`

Preventing Semantically-invalid Mutations: To avoid the risk of semantically-invalid mutations that lead to runtime crashes when executing generated binaries, BIN2WRONG's source mutators are designed to operate **conservatively**—deliberately avoiding injecting, deleting, or moving variable *declarations*. Without such safeguards, BIN2WRONG rapidly encounters semantics-related errors from missing or erroneously-placed declarations. We further build our mutators around AST-node-specific guardrails, such as placing `continue` or `break` statements only within loop bodies. While we see opportunities for more guardrails—for example, deliberately injecting source code that will not be optimized-out by the corresponding optimization flags selection—we posit that the runtime costs of additional checks likely outweighs their benefits, and that mutation quality ultimately does not hinder BIN2WRONG's overall performance, as shown by its strong results in our evaluation (§ 5).

Supported Constructs and Flexibility: While BIN2WRONG supports common source constructs—expressions, control logic and structures, and data objects, types, and values—we see room for incorporating additional ones (e.g., `union`, `struct`, and arrays). Whereas other fuzzers inherit limitations from their bespoke, non-exhaustive AST manipulation tooling (§ 3.1), an advantage of BIN2WRONG's design is that its source mutators operate **directly on Clang's robust AST** interface [56]. Adding new mutators targeting other AST node types requires minimal engineering, as Clang's AST already provides access to these and all other AST nodes.

Our Approach: By bridging AST-level source mutations with compilation mutations, general-purpose fuzzing can **explore the expansive range of diverse source and compilation *combinations***—achieving unprecedented binary diversity for *systematic* decompiler testing.

4.2 Decompiler-agnostic Semantic Testing

Following testcase mutation, BIN2WRONG compiles the binary according to the specified compilation parameters and forwards it to the targeted decompiler. Drawing inspiration from previous works [54, 139], BIN2WRONG employs *differential testing* to detect cases where decompiler-emitted code semantics *diverge* from the original program's. Below, we outline the core techniques enabling BIN2WRONG's discovery of semantics-affecting decompiler bugs.

4.2.1 Recompilation-oriented Syntax Patching

Since small syntactical errors prevent entire programs from being recompilable, we perform minor syntax corrections on decompiled code. Our high-level recompilation-oriented syntax patching spans the following basic strategies:

- Resolving compilation-blocking variable type mismatches:
  e.g., `void* const str;` → `char *str;`
- Renaming objects that differ from their intended names;
  e.g., `void __cdecl func_1()` → `void func_1()`
- Injecting macros to resolve decompilers' custom types:
  e.g., `word32 v0;` → `typedef int32_t word32;`
- Culling, replacing, or adding externs for decompiler-introduced macros, symbols, and functions:
  e.g., `movss(a);` → `extern float movss(float a);`

While some recompilation failures are not easily solved via basic syntax patching, we see a major improvement in post-patching recompilation success, rising from 11.60% to **47.8%** on average—comparable to D-Helix's reported 45% recompilation rate for Angr [142]. Importantly, such failures do not prevent BIN2WRONG's finding of *the most* bugs in our evaluation (Table 6); and moreover, no developers reported any false-positives caused by our syntax patching. We weigh additional strategies for improving recompilation success in § 6.2.

4.2.2 Bug-finding via Differential Execution

To uncover subtle semantic bugs (e.g., Table 1), we adopt Csmith [139]'s approach of *variable*-level differential execution checking. Like other fuzzers', BIN2WRONG's generated programs serve as the *decompilers'* inputs and are designed to be fully self-contained, requiring no external inputs. Each program deterministically exercises its logic based solely on internal control and data flows, ensuring reliable and repeatable semantic comparison across original and decompiled forms. Accordingly, we modify BIN2WRONG-generated testcases' source functions to initialize a suite of "placeholder" globals, which eventually assume their values from the mutated testcase's *local* variables. Post-decompilation, we instrument all globals in the original and recompiled programs to self-report value-dependent checksums. After executing both, if the recompiled program's checksum *differs* from the original's, we flag it as a *semantics-affecting* defect in the decompiled code.

By focusing solely on decompiler-emitted C code, BIN2WRONG extends support to *any* C-targeting decompiler. In contrast, D-Helix's [142] symbolic-execution-based approach needs ***per-decompiler, per-instruction*** **semantic modeling** for decompilers' internal Intermediate Representations (IRs), requiring considerable retooling to support more semantics, as evidenced by their lack of *floating-point* instructions (Table 2), and moreover, its estimated **40 days' time** to support *new decompilers*. BIN2WRONG avoids invasive bug-detection mechanisms, extending semantic correctness testing to *all* C decompilers—**both open-** ***and*** **closed-source**—whilst being the first to achieve systematic, unified mutation across all major dimensions of binary generation.

Our Approach: Utilizing the original program as *the correctness oracle* allows for decompiler-*agnostic* testing, eliminating any need for impractical tailoring to decompilers' dissimilar internal IRs—whilst enabling **practical testing of open-** ***and*** **closed-source decompilers**.

4.2.3 Post-fuzzing Manual Bug Analysis

Like other fuzzers [29], BIN2WRONG offloads bug post-processing to manual analysis. We spent about 10 minutes manually minimizing each bug-exposing testcase, in-line with the duration reported by DecFuzzer's authors [54], before deduplicating them into distinct *unique* bugs. While conventional fuzzers benefit from a rich ecosystem of automated fault localization tools (e.g., [87]), no analogous tools exist for binary decompilers; hence, we relegate fault localization to the best-equipped domain experts—decompilers' own developers—and promptly report all bugs we find.

5 Evaluation

Our evaluation of BIN2WRONG is guided by the following fundamental questions:

- **Q1. Testing Depth:** Is BIN2WRONG more effective at exercising decompilers' internals?
- **Q2. Bug Discovery:** Does BIN2WRONG reveal more decompiler semantic correctness bugs?
- **Q3. New Results:** Where do decompilers struggle in handling source and compilation diversity?

5.1 Experimental Setup

We evaluate BIN2WRONG alongside state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer across seven highly popular decompilers: the free and open-source Angr [132], R2Ghidra (Radare2+Ghidra) [1], Reko [49], Relyze [85], RetDec [6], and Rev.Ng [24]; as well as the commercially-sold decompiler Binary Ninja [108]. We omit D-Helix [142] as it unfortunately only supports two of our seven decompilers; as well as DSmith [20] as its source diversity is far more constrained than DecFuzzer's, excluding basic increment/decrement operators, jumps, and many common numerical data types [19]. As BIN2WRONG is built atop AFL++ v4.09c [29], we configure it in grey-box mode for the five decompilers supportive of its QEMU-based coverage tracing [16]—Binary Ninja, R2Ghidra, Reko, RetDec, and Rev.Ng—and coverage-agnostic black-box mode for Angr and Relyze.

We configure DecFuzzer's and Cornucopia's compilation parameters according to their respective papers: DecFuzzer with GCC and `-O0`, and Cornucopia with Clang and its full optimization suite. Per DecFuzzer's procedure [54], we seed all fuzzers with a corpus of Csmith-generated C programs (10 in total). Because Csmith omits certain constructs—most notably string literals—we manually augment its generated seed programs to include these elements. Following [43], we repeat all experiments for five trials; and compute Mann-Whitney U's statistical test at p=0.05 significance in comparing BIN2WRONG to Cornucopia and DecFuzzer (with statistically-significant p-values shown **bolded**). We perform all experiments on Ubuntu 22.04 machines, each with an Intel Core i9-12900K CPU and 64GB RAM.

5.2 Q1: Decompiler Testing Depth

In the following sections, we weigh BIN2WRONG's effectiveness in enabling in-depth decompiler testing through its impact on binary diversity and decompiler code coverage.

5.2.1 Binary Diversity

Rigorously testing decompilers demands binaries that *maximize* internal diversity. To assess BIN2WRONG's diversity, we measure binary similarity scores against Cornucopia and DecFuzzer using three industry-standard diffing algorithms:

DiffTool	BIN2WRONG Binary Diffing Scores per Dimension(s) of Source, Compiler, Optimization							BIN2WRONG-ALL Relative Mean Diffing Score Δ			
	ALL THREE	SRC+COM	SRC+OPT	COM+OPT	ONLY SRC	ONLY COM	ONLY OPT	over Cornucopia	MWU p	over DecFuzzer	MWU p
BinDiff	0.627	0.544	0.332	0.623	0.321	0.539	0.326	10.398 ×	**0.009**	17.189 ×	**0.011**
Radiff2-M	0.490	0.368	0.096	0.489	0.086	0.366	0.094	9.119 ×	**0.010**	16.941 ×	**0.007**
Radiff2-L	0.572	0.437	0.120	0.571	0.107	0.436	0.116	7.131 ×	**0.009**	16.089 ×	**0.011**

Table 3: Comparison between BIN2WRONG's mean binary diffing scores across all mutation dimension(s) for each diffing algorithm; and BIN2WRONG's overall scores relative to state-of-the-art decompiler fuzzers Cornucopia and DecFuzzer.

(1) Zynamics' BinDiff [143]; **(2)** Radiff2's Eugene W. Myers' $O(ND)$ Diffing Algorithm [62] (**Radiff2-M**); and **(3)** Radiff2's Levenshtein's Edit Distance [53] (**Radiff2-L**). We configure each to generate 1,000 testcases for each of their 10 Csmith-generated seed programs, totaling 10,000 testcases per competitor. We compute all three diff scores per generated binary relative to its seed binary, and report mean score across all comparisons. Table 3 shows the five-trial mean diversity per BIN2WRONG's supported mutation configurations—source, compiler, optimization, and all combinations thereof—alongside those of Cornucopia and DecFuzzer.

Results: Overall, BIN2WRONG-generated programs achieve **7.13–10.39**× and **16.08–17.18**× binary diversity relative to Cornucopia and DecFuzzer, respectively. In our ablation study of BIN2WRONG's full mutation modes, we see that mutating across *all* dimensions—source, compiler, *and* optimizations (column "ALL THREE" in Table 3)—yields the highest binary diversity across all three diffing algorithms, underscoring that **systematic, unified mutation *maximizes* binary diversity**.

5.2.2 Decompiler Code Coverage

As prior work shows [27], higher code coverage increases the likelihood of finding semantics-affecting defects. To determine whether BIN2WRONG's high-diversity inputs yield greater decompiler code coverage, we measure the coverage of BIN2WRONG, Cornucopia, and DecFuzzer on all coverage-tracing-compatible decompilers: Binary Ninja, R2Ghidra, Reko, RetDec, and Rev.Ng. For tracing coverage, we utilize the AFL-QEMU-Cov [28] tool, as it supports both open- and closed-source decompilers; however, because AFL-QEMU-Cov does not support Reko, we instead use AFL++'s built-in AFL-Showmap [29] utility to trace Reko's code coverage. We report the mean edge coverage results across five 24-hour trials per competitor, as shown in Table 4.

Decompiler	BIN2WRONG's Relative Increase over Cornucopia				over DecFuzzer			
	Δ COV	MWUp	Δ BIN	MWUp	Δ COV	MWUp	Δ BIN	MWUp
Bin.Ninja	1.091 ×	**0.008**	8.189 ×	**0.008**	1.215 ×	**0.008**	123.694 ×	**0.011**
Reko	0.989 ×	**0.008**	4.035 ×	**0.008**	0.973 ×	**0.008**	18.335 ×	**0.012**
R2Ghidra	1.344 ×	**0.012**	1.294 ×	0.421	1.451 ×	**0.008**	42.522 ×	**0.011**
RetDec	1.341 ×	**0.008**	0.798 ×	0.222	1.704 ×	**0.008**	26.201 ×	**0.007**
Rev.Ng	1.073 ×	**0.008**	16.836 ×	**0.008**	1.395 ×	**0.008**	297.763 ×	**0.011**
GeoMean:	**1.16** ×		**3.56** ×		**1.32** ×		**59.61** ×	

Table 4: BIN2WRONG's mean coverage (COV) and ratio of coverage-increasing binaries (BIN) relative to competitors.

Results: BIN2WRONG achieves mean **1.16**× and **1.32**× code coverage relative to Cornucopia and DecFuzzer, respectively, with all improvements showing statistically-significant p-values (< 0.05). Although BIN2WRONG shows statistically-significant lower coverage on Reko, it demonstrates substantially higher binary *quality*—the ratio of coverage-*increasing* binaries to total binaries generated—with a mean **3.56**× and **59.61**× relative to Cornucopia's and DecFuzzer's, respectively. This indicates that **BIN2WRONG's binaries exercise considerably more decompiler internals** compared to those generated by prior, *un*-systematic decompiler fuzzers.

Decompiler	Coverage-increasing Binaries by Mutation Configuration						
	ALL THREE	SRC +COM	SRC +OPT	COM +OPT	ONLY SRC	ONLY COM	ONLY OPT
Bin.Ninja	73.93%	0.04%	22.31%	3.26%	0.00%	0.00%	0.45%
Reko	82.47%	0.00%	16.64%	0.88%	0.00%	0.00%	0.00%
R2Ghidra	79.32%	0.02%	20.59%	0.07%	0.00%	0.00%	0.00%
RetDec	77.44%	0.00%	22.52%	0.03%	0.00%	0.00%	0.01%
Rev.Ng	68.56%	0.08%	30.96%	0.39%	0.00%	0.00%	0.00%
Mean:	**76.34%**	**0.03%**	**22.61%**	**0.93%**	**0.00%**	**0.00%**	**0.09%**

Table 5: BIN2WRONG's mean distributions of coverage-increasing binaries across all mutation dimension(s).

Furthermore, in an ablation study measuring the coverage impacts of different mutation configurations (Table 5), we observe that mutating source, compiler, and optimization *altogether* consistently produces the most coverage-enhancing binaries. Consequently, **by maximizing binary diversity, BIN2WRONG achieves the highest-overall coverage** across both open- and closed-source decompilers.

5.3 Q2: Decompiler Bug Discovery

We assess BIN2WRONG's effectiveness in identifying *semantic decompilation errors* across seven free and commercial decompilers: Angr, Binary Ninja, R2Ghidra, Reko, Relyze, RetDec, and Rev.Ng. We perform a subsequent manual root cause analysis to deduplicate all fuzzers' uncovered runtime divergences into their *unique* semantics-affecting decompiler defects. We report all identified decompiler correctness errors (Table 6) to their respective developers and maintainers.

Results: Overall, BIN2WRONG uncovers a total of **48** semantic decompilation bugs, **42** of which are uniquely found by BIN2WRONG alone, with **30** confirmed by their respective developers. Comparatively, mutation-restricted approaches Cornucopia and DecFuzzer reveal far fewer bugs—ten and zero, respectively—demonstrating the advantages of **BIN2WRONG's high-diversity, high-coverage systematic binary mutation** in thorough decompiler defect discovery. While § 5.2 suggests many of BIN2WRONG's binaries cover much of the same underlying decompiler logic—a common

Decompiler	Total Bugs Found, Confirmed, and Unique per Approach								
	BIN2WRONG			Cornucopia			DecFuzzer		
	BUGS	CONF	UNIQ	BUGS	CONF	UNIQ	BUGS	CONF	UNIQ
Angr	9	9	8	2	2	1	0	0	0
BinaryNinja	11	11	11	0	0	0	0	0	0
Reko	6	6	4	3	2	1	0	0	0
R2Ghidra	2	2	1	1	1	0	0	0	0
Relyze	7	N/A	6	2	N/A	1	0	0	0
RetDec	11	N/A	10	2	N/A	1	0	0	0
Rev.Ng	2	2	2	0	0	0	0	0	0
Total:	**48**	**30**	**42**	**10**	**5**	**4**	**0**	**0**	**0**

Table 6: Total decompiler bugs found (BUGS), total confirmed (CONF), and total unique (UNIQ) per each testing approach. N/A = instances where our correspondence went unanswered.

phenomenon in fuzzing as a whole—BIN2WRONG's broader strength is in its surfacing of subtle semantic bugs triggered by diverse source, compiler, optimization, and format *combinations*. Thus, we reason that BIN2WRONG's higher bug discovery is more directly related to its higher binary diversity (Table 3), rather than its higher code coverage (Table 4).

5.4 Q3: Analysis of Uncovered Bugs

While prior works categorize decompiler bugs into broad high-level classes (e.g., "Type Recovery" [54]), none delve into the specific *low-level causes*—or their prevalence—behind these critical failures. In the following, we conduct a comprehensive analysis of the semantics-affecting decompiler defects uncovered by BIN2WRONG, identifying their distinct source-related and compilation-related origins, supported by case studies on unique bugs discovered by BIN2WRONG.

5.4.1 Causes of Source-related Defects

Among the 48 semantics-affecting bugs discovered by BIN2WRONG, we identify **data recovery** as the most frequent error source (**68.75%**), followed by **expression** recovery (**20.83%**) and **control** structure recovery (**10.42%**).

Data Object, Type, and Value Recovery: Of the 33 data recovery bugs, we identify two involving incorrectly-recovered data *objects*: the first stems from Binary Ninja erroneously interpreting the `truncf()` function as taking multiple arguments instead of one [58]; while the second arises from Reko misordering function arguments, despite recovering them correctly otherwise. Table 7 presents a breakdown of the remaining 31 data-related bugs, categorized by *type* and *value* errors across integer, float, and string variables and literals. Although decompiler performance varies across data constructs, BIN2WRONG reveals several type recovery issues specifically related to **floating-point** data, which we discuss below.

Case Study: Floating-point Recovery. Figure 3 shows a BIN2WRONG-found error in floating-point recovery affecting Binary Ninja, Angr, and Reko. In binaries, floating-point values follow the IEEE-754 format, with instructions like `movss` for single-precision and `movsd` for double-precision, distinct from those used for integers. Although decompilers

	Integers				Floats				Strings			
	Value		Type		Value		Type		Value		Type	
	VAR	LIT	VAR	LIT	VAR	LIT	VAR	LIT	VAR	LIT	VAR	LIT
Angr	1	3	2	1	0	0	0	1	0	0	0	0
Bin.Ninja	0	0	1	2	0	0	0	1	0	1	0	0
Reko	0	0	0	0	0	1	0	0	0	2	0	0
R2Ghidra	0	1	0	0	0	0	0	0	0	0	0	0
Relyze	1	1	1	1	0	0	0	0	0	2	0	0
RetDec	1	4	2	0	0	0	0	1	0	0	0	0
Rev.Ng	0	0	0	0	0	0	0	0	0	0	0	0

Table 7: Summary of BIN2WRONG-uncovered data *type* and *value* bugs spanning variables (VAR) and literals (LIT).

accurately recognize floating-point *instructions* in disassembly (e.g., line#2 in Figure 3), they often fail to convert the hexadecimal representation of floating-point values *back* to their original form; instead, they output the *hex* form directly, incorrectly interpreting these values as integers rather than recovering them as floating-point numbers. These issues highlight the critical need to consider **diverse data constructs**, underscoring limitations of approaches like DecFuzzer, DSmith, and D-Helix which fail to support floating-point data (§ 3.1).

```
1  movsd xmm0,[rbp];
2  double v0 = 1.0;
```
(a) Original

```
1  int64_t v1 =
2  0x3FF0000000000000;
```
(b) Decompiled

Figure 3: Erroneous type recovery of float-point data as integers, affecting Angr, Binary Ninja, and Reko.

Expression and Control Recovery: We analyze the eight expression-related semantic errors identified by BIN2WRONG and categorize them into two groups. Four involve misrecoveries of **individual arithmetic operators**: incorrect handling of the modulo operator and interpreting signed comparisons as unsigned (Binary Ninja); unnecessary multiplication symbols (Reko); and reversing arithmetic operators to their opposites (RetDec). Our remaining four errors arise from mishandled parsing and refinement of **data-affecting instruction sequences**, including: inaccurate propagation of modulo results (Angr); erroneous omission of instructions preceding inlined function code (Binary Ninja); extraneous operations (Relyze); missed constant-affecting operations (RetDec); and functions that fail to parse correctly (both Reko and RetDec).

Of the five bugs arising from control structure recovery, only one involves loops (R2Ghidra), while the remaining four are related to `switch-case` structures (2×Binary Ninja, 2×Rev.Ng). Figure 4 presents a BIN2WRONG-uncovered bug exposing `switch-case` recovery challenges uniquely faced by commercial decompiler Binary Ninja.

Case Study: Switch-Case Logic Recovery. In programs with `switch-case`s, Binary Ninja's medium-level intermediate representation (IR) accurately translates `case` statements into semantically-equivalent nested `if-else` statements, preserving the intended logic. However, during translation to higher-level IR, Binary Ninja incorrectly *breaks* these into separate `if` statements. Although the conditions—the original `case` values—remain correct, this restructuring makes

```
1  int var = 0;
2  switch(var){
3    case 0:    Reached?
4      var = 5;  ✔
5      break;
6    case 1:
7      ...
8    case 2:
9      ...
10   default:
11     idx = 0;  ✗
12     break;
13 }
```
(a) Original

```
1  int var = 0;
2  if (var == 2)
3    ...
4  }
5  else{
6    if (var == 0) Reached?
7      var = 5;      ✔
8    if (var == 1)
9      ...
10   if (var > 2){
11     idx = 0;      ✔
12   }
13 }
```
(b) Decompiled

Figure 4: Incorrect switch-case recovery, making the original "case 0" and "default" blocks erroneously reachable.

multiple code blocks *reachable*, leading to divergent execution and incorrect program outputs. Binary Ninja's developers classified this bug as **High Severity**, indicating the need for a substantial overhaul of their core control-flow restructuring—underscoring the critical role of **diverse control constructs** in BIN2WRONG's systematic decompiler correctness testing.

5.4.2 Causes of Compilation-related Defects

We further examine the influence of different compilers, executable formats, and optimization flags on decompilation defects. Table 8 details the compilation parameters for all of BIN2WRONG's 48 uncovered decompiler bugs.

	Binary Formats & Associated Compiler(s)						# Opts		
	ELF				PE	Mach-O			
	CLANG	GCC	TCC	ICX	MSVC	APPLE	0	1	2+
Angr	9	7	7	9	4	9	9	0	0
Bin.Ninja	8	7	4	8	4	8	8	1	2
Reko	2	2	2	2	2	2	2	0	0
R2Ghidra	4	4	5	4	3	4	6	0	0
Relyze	7	5	5	6	3	0	7	0	0
RetDec	9	7	6	7	5	8	10	0	1
Rev.Ng	2	2	2	2	2	0	2	0	0

Table 8: High-level summary of the compilation parameters for all BIN2WRONG-discovered semantic bugs.

Optimization-dependent Bugs. Although most bugs are not directly caused by compiler optimizations, we observe that various optimizations are key to exposing intricate edge cases. We find three such bugs (2×Binary Ninja, 1×RetDec) from combinations of **two or more** optimizations, where decompilers produced only partial code and missed significant logic. Additionally, one bug, triggered only by the `-O1` optimization level, led to errors in loop logic recovery (in Binary Ninja). We detail the latter example below, shown in Figure 5.

```
1  while (v0!=11) { v0++ };
```
(a) Unoptimized code.

```
1  while (v0!=0xFFF5) { v0-- };
```
(b) Optimized & decompiled.

Figure 5: Binary Ninja bug stemming from the `-O1` optimization negating the loop constraint, causing divergent `v0` values in the original and decompiled programs post-loop-execution.

Case Study: Optimized Loop Recovery. Shown in Figure 5 is a `while` loop with counter incremented on each iteration, breaking only on count `11`. When compiled with `-O0`, the binary closely resembles this original structure; however, with `-O1`, the compiler replaces `11` with `-11` and the increment operation with a *decrement*. As negative numbers are stored in two's complement, Binary Ninja incorrectly recovers only this form, deviating from the original loop's semantics. BIN2WRONG caught this via the different `v0` values seen in the original and decompiled programs *post*-loop-execution (i.e., `11` with `-11`, respectively). This issue reiterates the importance of considering **diverse optimizations**—testing decompilers' ability to handle *each* effectively.

Compiler- and Format-dependent Bugs. We identify **eight** bugs uniquely stemming from a *single* executable format (ELF). However, these bugs are triggered by different compilers: four occur only in GCC-compiled binaries, one appears solely in TCC-compiled binaries, and the remaining span two or more compilers. Interestingly, we see the vast majority of bugs span *multiple*—but not all—executable formats and compilers: 14 appear in compilers *other* than GCC, 7 in ones other than Clang, 17 in ones other than TCC, 24 in ones other than MSVC, 10 in ones other than ICX, and 10 in ones other than AppleClang. Figure 6 shows an example bug that eludes PE binaries yet appears in both ELF *and* Mach-O formats.

```
1  mov r8,  z
2  mov rdx, y
3  mov rcx, x
4  func(x, y, z);
```
(a) PE convention & code.

```
1  mov rdi, x
2  mov rsi, y
3  mov rdx, z
4  func(z, y, x);
```
(b) Mach-O & ELF decompiled.

Figure 6: Call convention bug in Reko on Mach-O & ELF binaries (right) versus the analogous PE form (left).

Case Study: Calling Convention Recovery. In PE binaries, Microsoft's x64 conventions enforces that the first, second, and third integer arguments of a function are passed via registers `rcx`, `rdx`, and `r8`, respectively. Yet for ELF and Mach-O, AMD64's System V ABI instead passes these via `rdi`, `rsi`, and `rdx`, respectively. Shown in Figure 6, Reko's call recovery succeeds on PE binaries, yet *fails* on *non-PE* binaries' conventions, resulting in incorrect argument ordering. Ensuring decompilers' accuracy with these subtle differences reinforces the need to explore **both *ELF* and *non-ELF*** formats.

```
1  0x100 → 0x200
2  0x200 → 0x4142
3  mov rsi, [0x100]
4  char *s ="AB";
```
(a) Clang convention (one-level indirection) and original code.

```
1  0x100 → 0x150
2  0x150 → 0x200
3  0x200 → 0x4142
4  mov rax, [0x100]
5  mov rsi, [rax]
6  int64_t s =0x200;
```
(b) TCC convention (two-level indirection) and decompiled code.

Figure 7: TCC-specific string literal recovery bug in Reko.

Case Study: Compiler-specific String Recovery. Figure 7 depicts a compiler- *and* source-related bug in Reko. While most compilers store the string's direct memory address, allowing retrieval with a single dereference, TCC introduces an

extra indirection—a pointer to an address holding the actual location of the string, requiring *two* dereferences to access it. This approach presents challenges for decompilers, as seen in Reko, which retrieves only the *first* level of the memory address, leaving the second dereference unresolved. As a result, Reko fails to recognize the additional indirection, obscuring the true content of the string literal—and the program's intended semantics. Such defects demonstrate how binary semantics are uniquely influenced by **variations in both source *and* compilation**, highlighting BIN2WRONG's strengths over prior approaches that largely overlook such diversity.

6 Discussion

Below we discuss potential limitations of this work and our prototype decompiler fuzzer, BIN2WRONG.

6.1 Responses from Decompiler Developers

Developers of Angr, Binary Ninja, R2Ghidra, Reko, and Rev.Ng promptly **confirmed all of BIN2WRONG's uncovered issues as valid** decompilation bugs. Unfortunately, our attempts to contact developers of RetDec and Relyze went unanswered. Fortunately, most developers responded and quickly resolved many of our reported bugs, underscoring the importance of these fixes on the reliability of their tools.

We see that Binary Ninja is the only decompiler whose developers disclose severity scores of publicly-reported bugs, classifying eight of BIN2WRONG's found defects as follows: three *low* severity, four as *medium*, and one as *high* severity. These errors spanned *all three* of Binary Ninja's internal stages: one in its medium-level intermediate language (IL) generation, two in high-level IL, and five in final C code generation. BIN2WRONG's discovered high-severity bug (Figure 4) drew significant attention in particular, **prompting a total overhaul** of Binary Ninja's core control-flow structuring techniques. This bug was later featured in a blog post detailing the efforts taken to improve the soundness of Binary Ninja's decompilation—**further highlighting BIN2WRONG's real-world impacts** in uncovering critical semantic errors in even mainstream, commercially-sold decompilers.[1]

6.2 Reliance on Recompilation

BIN2WRONG's differential testing (§ 4.2.2) relies on *recompiling* decompiled code, with minor patching applied to fix common decompiler syntax errors. As BIN2WRONG's post-patch recompilation rate averages 47.8%—in-line with D-Helix's own 45% post-patching recompilation rate for Angr [142]—many syntax errors remain unresolvable by our current patches. We observe **84.03%** result from undeclared identifiers, **7.51%** from incompatible type conversions (e.g., pointers → integers), **6.24%** from extraneous arguments, **1.49%** from extraneous symbols (e.g., code comments), and **0.52%** from patch conflicts (e.g., BIN2WRONG adding `extern` within expressions). While more advanced program repair will improve recompilation, we posit its overhead will greatly reduces BIN2WRONG's total throughput. Thus, as research suggests *faster* fuzzing is better [63], we prioritize speed over 100% recompilation. Interestingly, the remaining **0.21%** of issues stem from *compiler* errors, typically with Clang or ICX crashing under certain optimization sets. Given their infrequency, we consider identifying and culling these specific conflicts outside our current scope.

6.3 Other Potential Binary Dimensions

Binary obfuscation—such as control-flow flattening, opaque predicates, and virtual-machine-based obfuscation—are often employed to hinder decompilers, typically to protect intellectual property or complicate reverse engineering [23, 107]. BIN2WRONG currently deliberately excludes such techniques, as contemporary decompilers tend to delegate deobfuscation tasks to specialized tools and plugins [35, 109]. However, BIN2WRONG easily supports incorporating obfuscation with minor adjustments, though we leave the exploration and requisite reevaluation of this to future work.

While we also considered linkers in BIN2WRONG's mutation space, our investigation found no prior decompiler bugs where the linker was the true root cause of semantic recovery errors. All linker-related issues we found were limited to syntactic or format-specific parsing failures—e.g., malformed headers or symbol resolution issues that simply prevented the binary from being fully analyzed at all (e.g., [64, 67, 83]). We believe this is unsurprising, as linkers must conform to OS-level specifications (e.g., System V ABI), and thus typically do not touch control/data semantics. As such, we scoped linking out of BIN2WRONG's design, though we anticipate this could be explored in future work focusing on malformed binaries or toolchain robustness.

6.4 Search Space and Bug Discovery Cost

While BIN2WRONG's combined source/compiler/format/optimization search space currently spans 6 compilers, 3 formats, 5,183 optimizations, and dozens of source constructs—easily amounting to trillions of unique binary generation configurations—BIN2WRONG's goal is not to exhaustively search this space, but to sample it broadly and systematically. Inspired by mutation-based fuzzers like AFL++ [29], we find that even simple, random mutations across multiple dimensions are highly effective at uncovering semantic bugs missed by prior tools limited to one or two axes (Table 2). BIN2WRONG's novelty thus lies in unifying diverse mutation dimensions into a single framework—showing that breadth alone, when systematically applied, surfaces impactful bugs that dimension-isolated fuzzers cannot reach. While we envision opportunities to more directly target distinct elements

[1] https://binary.ninja/2024/06/19/restructuring-the-decompiler.html

of this search space that hinder decompilation (e.g., specific source and optimization combinations guaranteed to produce interesting machine code), we leave this to future work.

7 Related Work

Below we discuss recent related research in the areas of binary decompilation, as well as the use of program generation and mutation in other applications of automated testing.

7.1 Recent Advancements in Decompilation

Several works are leveraging advances in machine learning to enhance the recovery of variable *type* information: Coda [30] employs neural networks, while Slade [5] and ReSym [135] utilize large language models. Machine learning is also seeing use in improving recovery of variable *names*, further boosting the human readability of decompiled code. DIRE [51], DIRTY [21], and TYGR [141] all utilize neural networks, whereas VarBERT [79] incorporates additional pre-training on human-written source code to produce more intuitive variable names. Many efforts are also enhancing restructuring and optimization of decompiler-generated *code*—particularly for `gotos`: Phoenix [18], Rev.Ng [24,33], and DREAM [137] aggressively cull `gotos`, while SAILR [15] instead introduces compiler-specific heuristics for eliminating only *spurious* ones. Though these recent advancements largely exist as separate from mainstream decompiler platforms, we expect that future efforts will bridge the gap between academically-sourced decompilation enhancements and industrial decompilers. We foresee BIN2WRONG playing a key role in enabling automated semantic testing of decompilers—and accelerating discovery of their bugs—as these platforms continue to evolve with increasing sophistication each year.

7.2 Program Generation and Mutation

A cornucopia of program generation and mutation techniques have emerged from the field of compiler fuzzing: Csmith [139], Rustsmith [88], and YARPGen [55] randomly generate grammar-conforming programs, while GrayC [27] mutates existing ones via Clang's AST API. More recently, Whitefox [138], GoFuzz [32], and Fuzz4All [134] instead use LLMs to explore more diverse source code constructs. As BIN2WRONG builds upon the program generation and mutation popularized by compiler fuzzing, we expect future synergistic approaches to extend BIN2WRONG's to other compiled languages. For instance, incorporating other language-specific AST APIs will help BIN2WRONG test emergent decompilers targeting non-C code, such as Go [41] and Rust [59].

7.3 Grey-box Fuzzers

BIN2WRONG's overall design inherits many design advantages from its core grey-box fuzzer, AFL++ [29]—which itself is one of many mainstream general-purpose fuzzers seeing large-scale adoption today [31]. Yet, standard grey-box fuzzers are not built to find semantic bugs such as decompilation recovery errors, and instead only look for crashing signals (e.g, `SIGSEGV`). While semantic bug-finding is not a new concept [55, 139], to our knowledge, BIN2WRONG is the first system to extend AFL++ with this capability in fuzzing decompilers. We anticipate these components of BIN2WRONG could likely be repurposed for similar semantics-related bug domains (e.g., compiler fuzzing, transpiler fuzzing).

8 Conclusion

Current decompiler testing approaches broadly overlook how source code, compilers, optimizations, executable formats, *and combinations thereof* collectively shape binary executable code. In this work, we show that giving equal consideration to these factors yields significantly more diverse binaries, enabling far more effective testing of decompilers' correctness. Beyond attaining the highest-overall code coverage across seven free and industrial decompilers, our prototype BIN2WRONG reveals the most semantic errors of all decompiler testing approaches—the vast majority of which remain undetectable by prior state-of-the-art methods. At the time of writing, **30 out of 48 decompiler errors found by BIN2WRONG are confirmed by developers**, with several prompting major changes in real-world decompilers.

By embracing principles that make conventional mutation-based application fuzzing successful—namely, maximizing testcase diversity through pursuing as broad of mutation as possible—we show that complex decompiler semantic errors are easily discovered in as little as 24 hours. We thus envision an era where holistic, automated techniques help improve the accuracy of today's many critical binary analysis tools—facilitating faster and more reliable completion of the downstream tasks that rely on these tools' precision.

Acknowledgments

We thank our reviewers for helping us improve the paper. This material is based upon work supported by the National Science Foundation under Grant No. 2419798.

References

[1] Sergi Alvarez. Radare2: Libre Reversing Framework for Unix Geeks. `https://github.com/radareorg/radare2`, 2024.

[2] Bob Amstadt, Eric Youngdale, and Alexandre Julliard. Wine: Run Windows applications on Linux, BSD, Solaris and macOS. `https://www.winehq.org/`, 2024.

[3] Quynh Nguyen Anh. Capstone: Next Generation Disassembly Framework. https://www.capstone-engine.org/, 2024.

[4] Apple. Apple xcode: C++ language support. https://developer.apple.com/xcode/cpp/, 2024.

[5] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F.P. O'Boyle. SLaDe: A Portable Small Language Model Decompiler for Optimized Assembly. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2024.

[6] Avast. Retdec: A retargetable machine-code decompiler based on LLVM. https://github.com/avast/retdec, 2024.

[7] Avast Software. RetDec issue #293: Code with fdivr decompiled wrong. https://github.com/avast/retdec/issues/293, 2018.

[8] Avast Software. RetDec issue #375: Incorrect decompile. https://github.com/avast/retdec/issues/375, 2018.

[9] Avast Software. RetDec issue #83: Avoid creating an empty if block. https://github.com/avast/retdec/issues/83, 2018.

[10] Avast Software. RetDec issue #669: Switch-cases wrongly deleted, indirect jump not reconstructed correctly. https://github.com/avast/retdec/issues/669, 2019.

[11] Avast Software. RetDec issue #673: label lab_generated_0 not declared and goto referencing it in decompile inner worlds. https://github.com/avast/retdec/issues/673, 2019.

[12] Avast Software. RetDec issue #717: Constants error on simpilify load instruction. https://github.com/avast/retdec/issues/717, 2020.

[13] Avast Software. RetDec issue #724: Incorrect translation:assembly to c. https://github.com/avast/retdec/issues/724, 2020.

[14] Avast Software. RetDec: Vtable-handling heuristics. https://github.com/avast/retdec/blob/master/src/rtti-finder/vtable/vtable_finder.cpp, 2024.

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

[16] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, ATC, 2005.

[17] Fabrice Bellard. TCC: the Tiny C Compiler. https://bellard.org/tcc/, 2024.

[18] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 Decompilation using {Semantics-Preserving} Structural Analysis and Iterative {Control-Flow} Structuring. In *USENIX Security Symposium*, SEC, 2013.

[19] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Dsmith artifact. 2024.

[20] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the Effectiveness of Decompilers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2024.

[21] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting Decompiler Output with Learned Variable Names and Types. In *USENIX Security Symposium*, SEC, 2022.

[22] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[23] Christian Collberg. The Tigress C Obfuscator. https://tigress.wtf/introduction.html, 2024.

[24] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: a Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *International Conference on Compiler Construction*, CC, 2017.

[25] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. Rev.Ng: Mach-O module. https://github.com/revng/revng/blob/develop/lib/Model/Importer/Binary/MachOImporter.cpp, 2024.

[26] Luboš Doležel. Darling: a translation layer that lets you run macOS software on Linux. https://www.darlinghq.org/, 2024.

[27] Karine Even-Mendoza, Arindam Sharma, Alastair F. Donaldson, and Cristian Cadar. GrayC: Greybox Fuzzing of Compilers and Analysers for C. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[28] Andrea Fioraldi. AFL-QEMU-Cov: Measure basic blocks coverage of all testcases in the afl queue

using a patched QEMU. https://github.com/andreafioraldi/afl-qemu-cov, 2024.

[29] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining Incremental Steps of Fuzzing Research. In *USENIX Workshop on Offensive Technologies*, WOOT, 2020.

[30] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An End-to-End Neural Program Decompiler. In *Advances in Neural Information Processing Systems*, NeurIPS, 2019.

[31] Google. ClusterFuzz, 2018.

[32] Qiuhan Gu. LLM-Based Code Generation Method for Golang Compiler Testing. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2023.

[33] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A Comb for Decompiled C Code. In *ACM Asia Conference on Computer and Communications Security*, ASIACCS, 2020.

[34] HyungSeok Han, JeongOh Kyea, Yonghwi Jin, Jinoh Kang, Brian Pak, and Insu Yun. QueryX: Symbolic Query on Decompiled Code for Finding Bugs in COTS Binaries. In *IEEE Symposium on Security and Privacy*, Oakland, 2023.

[35] Hex-Rays. gooMBA: a Hex-Rays Decompiler plugin that simplifies Mixed Boolean-Arithmetic (MBA) expressions. https://github.com/HexRaysSA/goomba, 2024.

[36] Hex-Rays. IDA Pro: A Powerful Disassembler, Decompiler and a Versatile Debugger. https://hex-rays.com/ida-pro, 2024.

[37] Harshada Hole. Microsoft: Discover Hidden Insights with Profiler's External Code Decompilation. https://devblogs.microsoft.com/visualstudio/discover-hidden-insights-with-profilers-external-code-decompilation/, 2024.

[38] Intel. oneAPI DPC++/C++ Compiler: a Standards-based, Cross-architecture Compiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/dpc-compiler.html, 2024.

[39] Intel Corporation. The X86 Encoder Decoder. https://intelxed.github.io, 2024.

[40] Internet Archive. Windows XP Unofficial Service Pack 4. https://archive.org/details/xp-unofficial-sp4-jan2022_20220113, 2016.

[41] Joakim Kennedy. Go Reverse Engineering Tool Kit. https://github.com/goretk, 2024.

[42] Soomin Kim, Markus Faerevaag, Minkyu Jung, Seungll Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. Testing Intermediate Representations for Binary Analysis. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2017.

[43] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating Fuzz Testing. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018.

[44] John Källén. Reko issue #529: Short circuit expressions not detected. https://github.com/uxmal/reko/issues/529, 2018.

[45] John Källén. Reko issue #543: Should encode chars >0x7f correctly in string literals. https://github.com/uxmal/reko/issues/543, 2018.

[46] John Källén. Reko issue #1114: Multiple issues in andn. https://github.com/uxmal/reko/issues/1114, 2021.

[47] John Källén. Reko issue #1149: Wrong decompilation of loop start. https://github.com/uxmal/reko/issues/1149, 2022.

[48] John Källén. Reko issue #1152: Wrong decompilation of movi/mvni. https://github.com/uxmal/reko/issues/1152, 2022.

[49] John Källén. Reko: a general purpose decompiler. https://github.com/uxmal/reko, 2024.

[50] John Källén. Reko: Control structure heuristics. https://github.com/uxmal/reko/tree/master/src/Decompiler/Structure, 2024.

[51] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A Neural Approach to Decompiled Identifier Naming. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2019.

[52] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *IEEE/ACM International Symposium on Code Generation and Optimization*, CGO, 2004.

[53] Vladimir I Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet Physics Doklady*, 10(8), 1966.

[54] Zhibo Liu and Shuai Wang. How Far We Have Come: Testing Decompilation Correctness of C Decompilers. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2020.

[55] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random Testing for C and C++ Compilers with YARPGen. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2020.

[56] LLVM. libClang. https://clang.llvm.org/docs/LibClang.html, 2024.

[57] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security Symposium*, SEC, 2019.

[58] POSIX Programmer's Manual. Truncf manpage. https://man7.org/linux/man-pages/man3/truncf.3p.html, 2024.

[59] Dhruv Maroo. GhidRust: Rust decompiler plugin for Ghidra. https://github.com/DMaroo/GhidRust, 2024.

[60] Microsoft. MSVC: Inlining Decisions in Visual Studio. https://devblogs.microsoft.com/cppblog/inlining-decisions-in-visual-studio/, 2024.

[61] Microsoft. Visual Studio Code: Develop C and C++ applications. https://visualstudio.microsoft.com/vs/features/cplusplus/, 2024.

[62] Eugene W. Myers. AnO(ND) Difference Algorithm and its Variations. *Algorithmica*, 1(1), 1986.

[63] Stefan Nagy and Matthew Hicks. Full-speed Fuzzing: Reducing Fuzzing Overhead through Coverage-guided Tracing. In *IEEE Symposium on Security and Privacy*, Oakland, 2019.

[64] National Security Agency. Ghidra issue #1054: Does not detect dos executables. https://github.com/NationalSecurityAgency/ghidra/issues/1054, 2019.

[65] National Security Agency. Ghidra issue #288: Invalid decompilation of cmpxchg instruction on x86_64. https://github.com/NationalSecurityAgency/ghidra/issues/288, 2019.

[66] National Security Agency. Ghidra issue #778: Decompiled f64 literals lack precision (e.g. for 1/4294967296.0). https://github.com/NationalSecurityAgency/ghidra/issues/778, 2019.

[67] National Security Agency. Ghidra issue #1054: 9.2: createinitializedblock() no longer supports memory region names with spaces. https://github.com/NationalSecurityAgency/ghidra/issues/1054, 2020.

[68] National Security Agency. Ghidra issue #3146: Mach-o library calls not handled properly. https://github.com/NationalSecurityAgency/ghidra/issues/3146, 2021.

[69] National Security Agency. Ghidra issue #3191: Incorrect packing in structures with unions. https://github.com/NationalSecurityAgency/ghidra/issues/3191, 2021.

[70] National Security Agency. Ghidra issue #4708: Ghidra doesn't handle floats in the msvc x86-64 calling convention properly. https://github.com/NationalSecurityAgency/ghidra/issues/4708, 2022.

[71] National Security Agency. Ghidra issue #4983: Improving work with function arguments. https://github.com/NationalSecurityAgency/ghidra/issues/4983, 2023.

[72] National Security Agency. Ghidra issue #5900: Incorrect sizing of data type in assignment leading to bad c output. https://github.com/NationalSecurityAgency/ghidra/issues/5900, 2023.

[73] National Security Agency. Ghidra issue #6119: Decompiler: double casting of variables. https://github.com/NationalSecurityAgency/ghidra/issues/6119, 2024.

[74] National Security Agency. Ghidra issue #6488: Missing and incorrrect variable assignments resulting from function calls. https://github.com/NationalSecurityAgency/ghidra/issues/6488, 2024.

[75] National Security Agency. Ghidra issue #6528: Decompiler erroneously swapping operators without changing the operand in floating-point comparisons where data is marked constant (x86). https://github.com/NationalSecurityAgency/ghidra/issues/6528, 2024.

[76] National Security Agency. Ghidra issue #6648: Sequence of rule applied affects the size ouput. https://github.com/NationalSecurityAgency/ghidra/issues/6648, 2024.

[77] National Security Agency. Ghidra issue #6708: [floats] truncated precision in decompiler view. https://github.com/NationalSecurityAgency/ghidra/issues/6708, 2024.

[78] National Security Agency. Ghidra Software Reverse Engineering Framework. https://ghidra-sre.org/, 2024.

[79] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. len or index or count, anything but v1": Predicting Variable Names in Decompilation Output with Transfer Learning. In *IEEE Symposium on Security and Privacy*, Oakland, 2024.

[80] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. SoK: All You Ever Wanted to Know About x86/x64 Binary Disassembly But Were Afraid to Ask. In *IEEE Symposium on Security and Privacy*, Oakland, 2021.

[81] PNF Software. JEB3: Reverse Engineering for Professionals. https://www.pnfsoftware.com/, 2024.

[82] radareorg. Radare2 issue #4422: Incorrect value of rax after xor eax,eax. https://github.com/radareorg/radare2/issues/4422, 2016.

[83] radareorg. Radare2 issue #9420: Some combo of a linkerscript, minimal sections copied, and a .eh_frame section cause r2 to see file as only 0xff bytes. https://github.com/radareorg/radare2/issues/9420, 2018.

[84] radareorg. Radare2 issue #17036: Incorrect detection of jump table's ending causing incorrect disassemblies in middle of opcodes (x86/64). https://github.com/radareorg/radare2/issues/17036, 2020.

[85] Relyze Software Limited. Relyze: Interactive Software Reverse Engineering. https://www.relyze.com/, 2024.

[86] Michael Rodler, David Paaßen, Wenting Li, Lukas Bernhard, Thorsten Holz, Ghassan Karame, and Lucas Davi. EF/CF: High Performance Smart Contract Fuzzing for Exploit Generation. In *IEEE European Symposium on Security and Privacy*, EuroS&P, 2023.

[87] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference*, ATC, 2012.

[88] Mayank Sharma, Pingshi Yu, and Alastair F Donaldson. Rustsmith: Random Differential Compiler Testing for Rust. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA, 2023.

[89] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. Cornucopia: A Framework for Feedback Guided Generation of Binaries. In *IEEE/ACM International Conference on Automated Software Engineering*, ASE, 2022.

[90] Robert Tarjan. Testing Flow Graph Reducibility. In *ACM Symposium on Theory of Computing*, STOC, 1973.

[91] The angr Project. Angr issue #2454: Inconsistent call graphs when resolving indirect calls. https://github.com/angr/angr/issues/2454, 2021.

[92] The angr Project. Angr issue #2911: Decompilation is not valid when there is double quote in string. https://github.com/angr/angr/issues/2911, 2021.

[93] The angr Project. Angr issue #2914: x86: Calling convention recovery is broken when multiple sizes of the same register are used. https://github.com/angr/angr/issues/2914, 2021.

[94] The angr Project. Angr issue #2915: x86: decompilation is missing exit condition of rep. https://github.com/angr/angr/issues/2915, 2021.

[95] The angr Project. Angr issue #3271: Ail optimization may incorrectly collapse converts. https://github.com/angr/angr/issues/3271, 2022.

[96] The angr Project. Angr issue #3377: test_decompilation_excessive_condition_removal is failing. https://github.com/angr/angr/issues/3377, 2022.

[97] The angr Project. Angr issue #3432: Return value type becomes int rather than long. https://github.com/angr/angr/issues/3432, 2022.

[98] The angr Project. Angr issue #3512: Failed switch-case identification. https://github.com/angr/angr/issues/3512, 2022.

[99] The angr Project. Angr issue #3702: Cfg incorrectly recovered? https://github.com/angr/angr/issues/3702, 2022.

[100] The angr Project. Angr issue #3703: Calling convention recovery. https://github.com/angr/angr/issues/3703, 2022.

[101] The angr Project. Angr issue #3737: Decompiler bug: incorrect arguments for xstrtol function call. https://github.com/angr/angr/issues/3737, 2023.

[102] The angr Project. Angr issue #3992: Calling convention gives wrong registers for arguments. https://github.com/angr/angr/issues/3992, 2023.

[103] The angr Project. Angr issue #4082: Incorrect loop ending under phonenix. https://github.com/angr/angr/issues/4082, 2023.

[104] The angr Project. Angr issue #4358: Incorrect goto edges reported from region simplifier. https://github.com/angr/angr/issues/4358, 2023.

[105] The angr Project. Angr issue #4420: Decompiler test test_decompiling_incorrect_duplication_chcon_main should have less gotos. https://github.com/angr/angr/issues/4420, 2024.

[106] The angr Project. Angr issue #4573: Expression incorrectly moved to lhs and not eliminated. https://github.com/angr/angr/issues/4573, 2024.

[107] Romain Thomas. Open-Obfuscator: O-MVLL & dProtect. https://obfuscator.re/, 2024.

[108] Vector 35. Binary Ninja: An interactive decompiler, disassembler, debugger, and binary analysis platform built by reverse engineers, for reverse engineers. https://binary.ninja/, 2024.

[109] Vector 35. Opaque Predicate Patcher. https://github.com/Vector35/OpaquePredicatePatcher, 2024.

[110] Vector35. Binary Ninja issue #1074: Analysis seems to ignore tailcalls in doing possible value analysis. https://github.com/Vector35/binaryninja-api/issues/1074, 2018.

[111] Vector35. Binary Ninja issue #1434: Incorrect jump table values. https://github.com/Vector35/binaryninja-api/issues/1434, 2019.

[112] Vector35. Binary Ninja issue #1581: Hlil incorrect control flow. https://github.com/Vector35/binaryninja-api/issues/1581, 2020.

[113] Vector35. Binary Ninja issue #1627: Sort hlil cases in switch statements. https://github.com/Vector35/binaryninja-api/issues/1627, 2020.

[114] Vector35. Binary Ninja issue #1789: Hlil confusing/incorrect if statement conditions. https://github.com/Vector35/binaryninja-api/issues/1789, 2020.

[115] Vector35. Binary Ninja issue #1791: Lookup table calculation fails. https://github.com/Vector35/binaryninja-api/issues/1791, 2020.

[116] Vector35. Binary Ninja issue #2824: continue statement wrongly optimized out during optimizations/lifting within a jump table. https://github.com/Vector35/binaryninja-api/issues/2824, 2022.

[117] Vector35. Binary Ninja issue #3186: Lifting llil->mlil shift value lifted incorrectly. https://github.com/Vector35/binaryninja-api/issues/3186, 2022.

[118] Vector35. Binary Ninja issue #3191: Shortcomings of bn's handling of float/vector values in the windows x64 calling convention. https://github.com/Vector35/binaryninja-api/issues/3191, 2022.

[119] Vector35. Binary Ninja issue #3719: Binja does not define second string when comparing. https://github.com/Vector35/binaryninja-api/issues/3719, 2022.

[120] Vector35. Binary Ninja issue #4110: Problems with decompiler splitting a 64-bit store into stores of two 32-bit floats. https://github.com/Vector35/binaryninja-api/issues/4110, 2023.

[121] Vector35. Binary Ninja issue #4252: Hlil generating constant if-condition. https://github.com/Vector35/binaryninja-api/issues/4252, 2023.

[122] Vector35. Binary Ninja issue #4368: Function ... hlil has invalid goto targets. https://github.com/Vector35/binaryninja-api/issues/4368, 2023.

[123] Vector35. Binary Ninja issue #4692: Don't display integer arguments as strings. https://github.com/Vector35/binaryninja-api/issues/4692, 2023.

[124] Vector35. Binary Ninja issue #4800: Wrong cast. https://github.com/Vector35/binaryninja-api/issues/4800, 2023.

[125] Vector35. Binary Ninja issue #3081: Placeholder title. https://github.com/Vector35/binaryninja-api/issues/3081, 2024.

[126] Vector35. Binary Ninja issue #4223: Placeholder title. https://github.com/Vector35/binaryninja-api/issues/4223, 2024.

[127] Vector35. Binary Ninja issue #4971: Function is called without a parameter even though its type requires one. https://github.com/Vector35/binaryninja-api/issues/4971, 2024.

[128] Vector35. Binary Ninja issue #4977: Incorrect main function detection in pe, pdb causes duplicate. https://github.com/Vector35/binaryninja-api/issues/4977, 2024.

[129] Vector35. Binary Ninja issue #5009: Cannot figure out the parameter of the call if the parameters are moved into the stack rather than pushed. https://github.com/Vector35/binaryninja-api/issues/5009, 2024.

[130] Vector35. Binary Ninja issue #5030: Inconsistent display of bool values in conditional statements. https://github.com/Vector35/binaryninja-api/issues/5030, 2024.

[131] Vector35. Binary Ninja issue #5650: No-return not being treated as such if the call instruciton has unresolved stack usage. https://github.com/Vector35/binaryninja-api/issues/5650, 2024.

[132] Fish Wang and Yan Shoshitaishvili. Angr - The Next Generation of Binary Analysis. In *IEEE Cybersecurity Development*, SecDev, 2017.

[133] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One Fuzzing Strategy to Rule Them All. In *International Conference on Software Engineering*, ICSE, 2022.

[134] Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4All: Universal Fuzzing with Large Language Models. In *IEEE/ACM International Conference on Software Engineering*, ICSE, 2024.

[135] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries. In *ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2024.

[136] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. In *IEEE Symposium on Security and Privacy*, Oakland, 2016.

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

[138] Chenyuan Yang, Yinlin Deng, Runyu Lu, Jiayi Yao, Jiawei Liu, Reyhaneh Jabbarvand, and Lingming Zhang. WhiteFox: White-Box Compiler Fuzzing Empowered by Large Language Models. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA, 2024.

[139] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI, 2011.

[140] Yupeng Yang, Yongheng Chen, Rui Zhong, Jizhou Chen, and Wenke Lee. Towards Generic Database Management System Fuzzing. In *USENIX Security Symposium*, SEC, 2024.

[141] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, et al. {TYGR}: Type Inference on Stripped Binaries using Graph Neural Networks. In *USENIX Security Symposium*, SEC, 2024.

[142] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave Jing Tian. {D-Helix}: A Generic Decompiler Testing Framework Using Symbolic Differentiation. In *USENIX Security Symposium*, SEC, 2024.

[143] Zynamics. Bindiff: a comparison tool for binary files. https://www.zynamics.com/bindiff.html, 2024.