D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation

Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian, *Purdue University*

https://www.usenix.org/conference/usenixsecurity24/presentation/zou

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation

Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, Dave (Jing) Tian

Purdue University
{zou116, khan253, wu1377, coolgao, antoniob, daveti}@purdue.edu

Abstract

Decompilers, one of the widely used security tools, transform low-level binary programs back into their high-level source representations, such as C/C++. While state-of-the-art decompilers try to generate more human-readable outputs, for instance, by eliminating `goto` statements in their decompiled code, the correctness of a decompilation process is largely ignored due to the complexity of decompilers, e.g., involving hundreds of heuristic rules. As a result, outputs from decompilers are often not accurate, which affects the effectiveness of downstream security tasks.

In this paper, we propose D-HELIX, a generic decompiler testing framework that can automatically vet the decompilation correctness on the function level. D-HELIX uses RECOMPILER to compile the decompiled code at the functional level. It then uses SYMDIFF to compare the symbolic model of the original binary with the one of the decompiled code, detecting potential errors introduced by the decompilation process. D-HELIX further provides TUNER to help debug the incorrect decompilation via toggling decompilation heuristic rules automatically. We evaluated D-HELIX on Ghidra and angr using 2,004 binaries and object files ending up with 93K decompiled functions in total. D-HELIX detected 4,515 incorrectly decompiled functions, reproduced 8 known bugs, found 17 distinct previously unknown bugs within these two decompilers, and fixed 7 bugs automatically.

1 Introduction

Widely applied in security, decompilation is a reverse engineering procedure that transforms a low-level binary program back to high-level source code representations such as C/C++. For example, Ghidra is such a decompilation tool "developed by NSA's Research Directorate in support of the Cybersecurity mission" [1]. The whole decompilation process can be divided into four stages: disassembly, lifting to Intermediate Representation (IR), higher-level semantics abstraction, and decompiled code generation [11]. Disassembly is the initial process that translates the machine code into assembly code. Then the lifter further converts it into IR (e.g., P-Code [6] or VEX [3]). Based on the IR, the decompiler recovers higher-level semantics (e.g., control flow graphs, variable types, or structures) and finally generates the decompiled code as output. In this paper, we refer *lifter* as the tool that can convert machine code into IR and define *decompiler* as the tool working on IR and finally generating the decompiled code.

Although decompilation techniques are expected to facilitate real-world reverse engineering, many engineers do not trust the output of decompilers. In fact, approximately 69% of reverse engineers (i.e., 11 out of 16) pay less attention to the decompilation output since they think the decompilation process is error-prone and the wrong result will mislead the analysis [50].

The hurdle of impeding security engineers from trusting decompilers, as we observe, is the essential trade-off between code readability and semantic accuracy. During the last two stages of the decompilation, a decompiler has to infer information (e.g., structural or type information) that has been discarded during compilation [60]. To tackle this, modern decompilers involve hundreds of heuristic rules, such as type reconstruction [38] and code structure recovery [12] to compensate for the missing information, and iteratively apply these rules to generate code that is progressively more and more readable. However, as shown in existing work [30], these heuristic rules might be unsound or even incorrect, introducing semantic inaccuracies in the decompiled code. Though security researchers have conducted numerous works [10, 12, 13, 16, 22, 54, 55, 60] to improve decompilers, most of them focus on improving the readability of the output, e.g., introducing heuristics to minimize `goto` statements. Consequently, as we will show later in Table 1, most of the state-of-the-art decompilers neglect the semantic accuracy of decompiled code when devising the heuristic rules. Meanwhile, not surprisingly, current decompilers [4, 5, 9, 29, 48, 51] have no mechanism to test their heuristic rules' soundness and implementation correctness, except basic unit testing. To the best of our knowledge, there is only

[1] https://ghidra-sre.org/

one existing study systematically investigating the correctness of decompilers [30]. However, its methodology has certain limitations, for instance, it fails to detect incorrect structure recovery. Furthermore, it can only identify bugs within the decompiled code, without identifying the root cause of the inaccuracy within decompilers.

In this paper, we propose D-HELIX, *a generic decompiler testing framework that can automatically vet the decompilation correctness.* Compared to the existing study, D-HELIX directly works on the binary code and thus does not require source code, thereby following a more realistic usage of decompilers and facilitating testing with a broader scope. Furthermore, D-HELIX is capable of identifying problematic heuristics within decompilers. Initially, D-HELIX performs a dedicated recompilation procedure to compile the decompiled code at the function level. Afterward, by performing symbolic execution, D-HELIX extracts symbolic models from the original binary and the recompiled decompiled output. These symbolic models are then compared to detect semantic discrepancies between the decompiled code and the original binary. Finally, D-HELIX applies a tuning algorithm to infer which decompiler's heuristic rule could be responsible for the detected decompilation inaccuracy by efficiently exploring the configuration space of the tested decompiler.

We have implemented D-HELIX and applied it to Ghidra [5] and angr [51]. In particular, we used Ghidra and angr to decompile a dataset of 2,004 real-world binaries from Github and obtained 93K decompiled functions in total. D-HELIX detected 4,515 incorrectly decompiled functions, reproduced 8 known bugs, found 17 distinct previously unknown bugs in these two widely used decompilers, and automatically located the root causes of 7 bugs. D-HELIX provides a powerful tool for both decompiler developers and security researchers to detect inaccurate decompilation results and debug decompilation processes.

In summary, our contributions are as follows:

- We design D-HELIX, a fully automatic decompiler testing framework using only binaries as the input, with three components:

  1) RECOMPILER, a best-effort recompiler enabling compilation of decompiled output at the function level.

  2) SYMDIFF, an automatic symbolic model checking tool supporting different symbolic execution engines and enabling the detection of inaccurate decompilation by capturing semantic discrepancies between the original binary code and decompiled outputs.

  3) TUNER, an automatic debugger to find potential culprit heuristic rules that lead to inaccurate decompilation by efficiently and effectively exploring different combinations of the heuristic rules available within a decompiler.

- We fully implement D-HELIX and apply it to Ghidra and angr. Besides reproducing 8 known bugs within these decompilers, D-HELIX found 17 distinct previously unknown bugs, 11 in Ghidra and 6 in angr, revealing intrinsic interactions among different heuristic rules that led to inaccurate decompilation.

We followed the disclosure practices and reported all the found bugs to the affected parties. The source code for D-HELIX is available at: https://github.com/purseclab/D-helix.

2 Motivation

Decompiler	SP	SLoC	Heu	OSS
DREAM [55] DREAM++ [54]	✓	12.9K	9	✓
Foxdec [49]	✓	2,924K	146	✓
Retdec [9]	×	2,437K	46	✓
Ghidra [5]	×	4,258K	151	✓
Reko [48]	×	6,764K	26	✓
angr [1]	×	246.8K	41	✓
Radeco [41]	×	40.5K	18	✓
Rellic [29]	×	25.3K	27	✓
llvm-cbe [24]	×	10.9K	0	✓
Phoenix [12]	✓	–	–	×
rev.ng-c [22]	×	–	–	×
Hex-Rays [4]	×	–	–	×
JEB [7]	×	–	–	×
BinNinja [8]	×	–	–	×

Table 1: Systematization of decompilers and their characteristics. SP = Semantic-Preserving, Heu = Heuristics, and OSS = Open Source Software.

Though decompilers are extensively used for security tasks [13, 15, 44, 50, 52–54], many people do not trust the output of decompilers because of the potential inaccuracy introduced during decompilation. To understand the inaccuracies in decompilers, we demonstrate the following observations.

Observation 1: *Decompilers tend to overlook the importance of ensuring the semantic preservation of their decompiled code.* We surveyed 15 state-of-the-art generic decompilers, including both open-source projects and commercial products. As shown in Table 1, we illustrate the systematization of the decompilers from four perspectives, which are semantic-preserving (SP), source lines of code (SLoC), the number of heuristic rules (Heu), and source code availability (OSS). For the semantic-preserving feature of decompilers, we check whether the terminology (e.g., preserving semantic accuracy) is mentioned as a feature in their papers or product descriptions. For open-source decompilers, we get SLoC from the entire codebase of the respective project and count the number of heuristic rules by exploring the projects ourselves.

We observe that most decompilers leave the semantic accuracy out of consideration, while only three decompilers include different semantic-preserving methods to build their decompilers. For instance, Phoenix [12] introduces semantic-preserving iterative refinements, attempting to ensure the control flow structure is correctly recovered. DREAM/DREAM++ [54,55]

provides semantic-preserving transformations focusing on data flow and control flow path recovery, such as `goto` removal, redundant variable removal, semantic-aware variable naming, etc. FoxDec [49] uses formal methods for semantic-preserving decompilation. However, except formally-verified decompilation, all these works focus on introducing semantic-preserving heuristics to address their corresponding task domains, such as `goto` statement elimination. Additionally, for FoxDec, the binaries it supports are constrained by certain assumptions, such as no global variables and no indirect branching. For the remaining decompilers, we observe that their decompiled outputs contain many semantic inaccuracies, e.g., type recovery errors, structure recovery errors, and function pointer type recovery errors. These inaccuracies arise due to the loss of critical information during the compilation process, that is, compilers translate variables and data structure fields to plain registers and memory locations without any structural or type information. Sometimes, decompilers use special symbols to represent the terms they cannot recover. For instance, Ghidra uses `undefined[16]` to represent the type of a 16-bit variable that it cannot recover. However, not every inaccuracy is labeled with special symbols. As we will show later in Section 6, there exists a certain amount of inaccuracies that can hardly be detected without testing.

To conclude, current decompilers define function semantics in different ways and conduct limited examinations of the decompiled code.

Observation 2: *There lacks a generic methodology, which can soundly examine the decompilers.* While existing works [25, 30–32, 34, 40] propose different methodologies to conduct code equivalence examinations, only one of them [30] focuses on the decompiled code. For the rest, as we will further discuss in Section 8, their methodologies are not sound enough to examine the decompiled code, e.g., these methods cannot detect incorrect recovery of function prototypes in the decompiled code. Meanwhile, the existing decompiled code equivalence examination [30] lacks generality and is not sound. Specifically, it uses Equivalence Modulo Inputs (EMI) testing [27] to mutate the original source code and compare it with the corresponding decompiled code under the same set of pre-defined inputs to find semantic inaccuracies. However, this method is not generic, since it cannot report inaccuracies in binaries without source code. For instance, it relies on Csmith [57] to generate the source code, which is devoid of undefined behavior (e.g., no uninitialized variables). Consequently, this approach cannot detect the semantic inaccuracy of decompilers on binaries containing undefined behavior. Furthermore, its methodology is not sound to thoroughly examine decompilers, as it does not support the decompiled code with C struct and union, thereby failing to detect any incorrect structure recovery bug in decompilers.

Observation 3: *It is hard to debug root causes of semantic inaccuracies in decompilers.* Based on our survey, we note that existing decompilers have a non-trivial codebase and employ a large number of heuristics (e.g., converting the `goto` statement to the while loop) to facilitate decompilation. As shown in the LoC and Heu columns of Table 1, decompilers have code base sizes from 12.9K (i.e., DREAM [55]) to 6.8M (i.e., Reko [48]) lines of code and can have as many as 151 heuristic rules (i.e, Ghidra [5]) applied during the decompilation process. One may think that debugging decompilers is similar to debugging compilers because they both apply heuristics/optimization passes to the code. However, in the case of compilers, developers can rely on the fact that compilers should always generate a correct binary when some optimization passes are applied. In our case, however, this assurance does not hold true. Specifically, because decompilation is a process of recovering the IR to the high-level language, decompilers heavily rely on heuristics to infer essential missing information. Therefore, when some heuristic rules are not applied, decompilers do not ensure semantic preservation in their generated code. In fact, when no heuristic rule is applied, decompilers may totally screw up the decompiled code.

Meanwhile, to correctly infer the missing information (e.g., the data type of a variable), decompilers usually look through the context of the whole binary (e.g., the data-flow of that variable) and iteratively apply heuristic rules to each instruction. For instance, Ghidra iteratively applies all heuristic rules to a single instruction before moving to the next one, while angr iteratively applies one heuristic rule at a time but to all instructions. Hence, it is common for decompilers to apply numerous heuristic rules on a single instruction hundreds of times, which yields thousands of distinct intermediate statuses. As a result, to debug a semantic inaccuracy, decompiler developers need to review these intermediate statuses to identify at which point the inaccuracy begins.

Our Approach: We propose our testing framework, which runs symbolic execution testing on the lifted IR level. Compared with previous work, it is more generic. Specifically, since every heuristic-based decompiler generates the decompiled code from the lifted IR, every piece of decompiled code has its corresponding lifted IR code. Therefore, as our testing framework runs symbolic execution testing on the lifted IR level, it allows any real-world binary as the input to test decompilers. Meanwhile, it is more complete compared with previous work. Since we use the lifted IR as the ground truth to compare with the decompiled code, our approach will not report inaccuracies introduced in the disassembling or lifting phase (i.e., lifter). Moreover, to help debug decompilers, we implement a dedicated heuristic rule tuning system on top of the symbolic execution testing approach that we use.

3 Design Challenges

Section 2 motivates the need for a novel decompiler testing framework, which can not only provide a more generic and sound semantic examination of the decompiled code but also automatically debug the root causes of different semantic in-

accuracies. In this section, we will summarize the challenges we need to solve to achieve our goal.

Challenge 1: *Recompilation.* To check the semantics of the decompiled code, existing work [12, 30] recompiles the decompiled code as a whole into a binary. However, as syntactical errors such as undefined global variables are common among the decompiled code [30], decompilers do not guarantee the recompilability of the code they produce. To address these errors, existing work [12, 30] needs the original source code for help. Specifically, they replace the component containing syntactical errors with their corresponding original source code during the recompilation. Nevertheless, many real-world binaries do not come with corresponding source code. Hence, recompiling the entire decompiled code of close-source binaries is challenging.

Observation: While recompiling decompiled code as a whole is usually challenging, an existing study indicates that recompiling at the function level is usually approachable [39], after proper adjustments.

Solution: To make sure one decompiled function that cannot be recompiled does not affect the evaluation of the other functions in the binary, we recompile each function independently. Specifically, we design RECOMPILER, an *iterative automatic recompiler* (details in Section 4.1) to revise and recompile the decompiled code at the function level.

Challenge 2: *Semantic error detection.* There is no universally accepted definition of function semantics in the existing work [12, 30, 49, 54, 55]. For instance, certain definitions consider two functions as equivalent in semantics if they both pass a unit test [12], whereas an alternative criterion checks equivalence in the sum of instrumented global variables [30]. Meanwhile, codes sharing the same function semantics can have syntactical differences in terms of control flows, data flows, and the used instructions. As a result, determining whether two functions are semantically equivalent is non-trivial.

Observation: Symbolic execution provides a sound approach to verify the semantic equivalences. However, it is known to suffer from poor scalability due to path explosions.

Solution: We define the semantic equivalence of two functions as a subset of observation equivalence focusing on their return values. To achieve this, we formally represent the semantics of each function as a symbolic model (refer to Section 4.2). Meanwhile, to examine the semantic equivalence of two functions, we propose *symbolic differentiator* (SYMDIFF), which first abstracts the semantics of each function into a symbolic model through symbolic execution and then employs an SMT solver to verify their equivalence. By focusing on comparing code at the function-level granularity and generating symbolic models from both binary and decompiled code, our symbolic model differentiator allows us to keep the scalability of the symbolic execution, while detecting bugs in decompilers.

Challenge 3: *Semantic errors debugging.* Modern decompilers typically have a complex codebase and employ a large number of heuristic rules. Due to this complexity, as we will later show in Section 6.4, we notice that even for their developers it is hard to trace and fix bugs in decompilers. For instance, as the time of writing, there are still over 850 issues open in Ghidra Github.

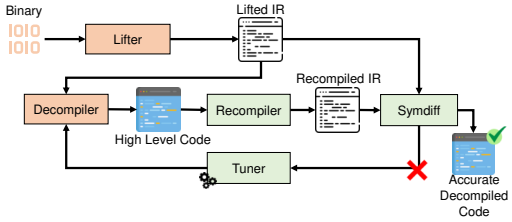

Figure 1: D-HELIX pipeline: D-HELIX extracts symbolic models for *Lifted IR* and *Recompiled IR*, and compares the symbolic models using a constraint solver. For semantically inequivalent models, D-HELIX uses the TUNER to fix the decompiled code.

Observation: Decompilers apply a set of heuristic rules while generating and simplifying the decompiled code. Typically, each heuristic rule has a dedicated goal (e.g., use debug information to help recover the function prototype) and the application of a specific heuristic rule (or a small subset of heuristic rules) is what causes a specific decompilation error, e.g., misordered function arguments due to incorrect debug information analysis.

Solution: To automatically find the root cause of the identified decompilation errors, we propose TUNER, which iteratively toggles decompilers' heuristic rules and compares the newly generated decompiled code with the original binary via the help of SYMDIFF. Since TUNER identifies the problematic heuristic rule(s) fully automatically, it can help decompilers' developers save manual efforts in locating the root cause of a given decompilation error.

4 Design

D-HELIX is a generic decompiler testing framework that can automatically vet the decompilation correctness. Figure 1 shows the pipeline for D-HELIX. D-HELIX takes a binary and a decompiler as its inputs. As a first step, D-HELIX translates the input binary to *Lifted IR* using the *lifter* of the analyzed decompiler. Then, D-HELIX performs the following operations for each function present in the analyzed binary. First, the *Lifted IR* is passed to the tested *decompiler* to generate the decompiled code. Then, RECOMPILER (Section 4.1) compiles the decompiled code to generate *Recompiled IR* with the help of the compiler. Next, D-HELIX checks the semantic equivalence of the *Lifted IR* and the *Recompiled IR* using the SYMDIFF (Section 4.2). If both IRs are equivalent, D-HELIX reports the decompiled code as accurate. Otherwise, D-HELIX uses TUNER to automatically identify the heuristic rule(s) within a decompiler that is(are) responsible for the detected

semantic inaccuracy (Section 4.3).

4.1 RECOMPILER

As mentioned, our approach is based on comparing the IR we obtain by lifting the original binary against the IR we compile from the recompiled decompiled code. To achieve this goal, we need to recompile the decompiled code output by the decompiler. Unfortunately, decompilers often output code that is not even syntactically valid and, for this reason, it cannot be directly recompiled.

A naive approach would just discard any non-recompilable function and consider it as a wrongly decompiled function. However, by doing so, a large amount of functions would be filtered out since decompilers do not guarantee recompilability, and the number of functions we can analyze using our symbolic differentiation approach would be significantly lower, hindering our ability to detect hard-to-find, semantic decompilers' errors.

To recompile as many functions as possible, we use RECOMPILER, a program mutation engine that uses the errors generated by the compiler as feedback to fix syntactic errors from the decompiled code. In particular, RECOMPILER uses an *Iterative Recompilation* process. Firstly, RECOMPILER places the decompiled source code of each function in a separate translation unit (i.e., file). Then, for each translation unit, RECOMPILER modifies the decompiled code iteratively. Specifically, during each iteration, RECOMPILER attempts to compile the code and counters the eventual error logs emitted by the compiler in the following way.

Undefined variables. RECOMPILER first tries to recover their initial values and types from the lifted IR code. Specifically, some decompilers (e.g., Ghidra) store the types and initial values of global variables in the IR code but do not define them in the decompiled code. In this case, RECOMPILER directly defines these variables using the information provided by the lifter. Otherwise, RECOMPILER follows the convention used by the VS compiler (i.e., C4430 [37]) to define these variables with a default type and value, e.g., *int x;*. Note that this approach can potentially introduce semantic inaccuracies into the code, as we will discuss further in Section 6.1.

Syntactic errors. RECOMPILER uses the syntactic error details to update the wrong type to the type expected by the compiler. For instance, RECOMPILER would help remove a pointer, when the compiler reports errors, such as *indirection requires pointer operand*.

RECOMPILER keeps fixing these errors until either the current translation unit is free of errors (i.e., can be recompiled) or a maximum iteration count (by default, 10) is reached. When the maximum iteration count is reached, RECOMPILER terminates and reports the failure to D-HELIX, which will ignore such a function in the later stages of the pipeline. Conversely, for each recompiled translation unit, RECOMPILER generates the corresponding IR for the next stage of D-HELIX pipeline.

4.2 SYMDIFF

To examine function-level semantics of the decompiled code, we design SYMDIFF, a dynamic symbolic execution engine that automatically verifies the semantic equivalence of two input functions. As RECOMPILER generates *Recompiled IR* at the function level, SYMDIFF first uses the function identifiers to map each function from the *Recompiled IR* to the corresponding function in the *Lifted IR* and then test their semantic equivalence using symbolic execution.

As shown in Figure 2, SYMDIFF constitutes two components: Generator and Comparator. Generator constructs a symbolic model of the input IR, whereas Comparator matches two models to determine if they are semantically equivalent. As the design of Generator aims to accommodate various IRs and symbolic engines, this design hides their potential differences.

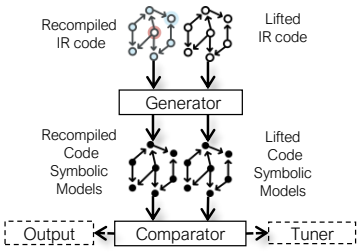

Figure 2: SYMDIFF: Comparator compares symbolic models, generated by Generator.

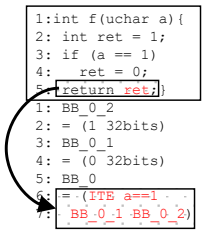

Figure 3: C code and its symbolic model.

4.2.1 Generator

For each input function, Generator generates a symbolic model to represent its function semantics. To do this, we build a symbolic model as a mathematical representation of the modeled function, such that the inputs of the formula are the input arguments of the modeled function and the output of the formula is the return value of the modeled function. For example, Figure 3 shows the symbolic model generated by Generator for a simple C function. The function takes one input argument and returns 0 if the value of the input variable is zero, otherwise, it returns one. Similarly, the corresponding symbolic model takes in one input argument and achieves the same behavior as the C code by using an If-Then-Else (ITE) expression that evaluates to zero if the value of the input variable is zero.

To generate these symbolic expressions, Generator performs symbolic execution on different IRs using different engines, exploring all possible paths leading to the function return instruction(s). More in detail, Generator first considers all functions' arguments as symbolic and then performs symbolic execution on the function through the modified symbolic execution engines. After that, Generator merges symbolic expressions related to different explored paths using appropriate ITE expression, in which the branch conditions (e.g., if condi-

tion) correspond to the path constraints of the merged paths. Finally, as exemplified in Figure 4, Generator creates a symbolic expression representing how the function's return value is computed based on its arguments.

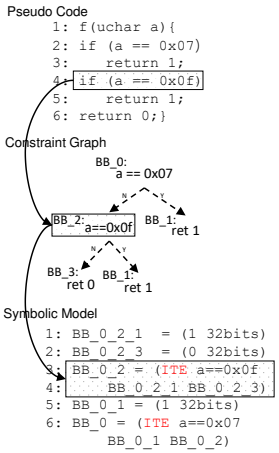

Figure 4: Sample function and its symbolic model.

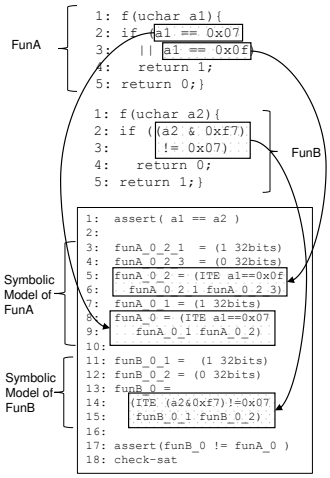

Figure 5: The bottom SMT query is generated by SYMDIFF to compare the two functions at the top.

Noted that to handle scalability problem [26], we designed specific strategies regarding how we handle external function calls, loops, and memory accesses, as we will explain next.

Loop Bounding: To limit the number of paths we need to explore, if a configurable timeout is reached during the symbolic execution of a function (as detailed in Section 5), we limit the number of times we execute each branch instruction. Specifically, if a branch instruction is about to be executed more than 2 times, we stop the symbolic exploration by executing a dummy `return` instruction. This ensures that we execute the loop body of each loop at least once. While this approach can introduce imprecision in our analysis, it is in line with HeapHopper [21] and FIE [18], which show that a single iteration of a loop can yield comparable code coverage to that of a hundred loop iterations.

Memory Model: To achieve a good trade-off between precision and scalability among existing ways to model symbolic memory accesses [42, 46], we follow the approach as outlined below. To universally accommodate the memory layouts and objects, we initialize all the memory to a default value of zero. In the case of a read operation, if a symbolic memory address is used in the read operation, SYMDIFF enumerates all of its possible values, up to a configurable upper bound. On the other hand, when handling a symbolic memory address used in a write operation, SYMDIFF queries an SMT solver to obtain its maximum possible value. To achieve a good trade-off between false positives and high memory usage overhead, for pointer-type input arguments, we make them point to an allocated symbolic space of 256 bytes. For other input arguments, we model their values as 8 symbolic bytes.

External Calls: Since RECOMPILER recompiles each function separately, SYMDIFF executes symbolically on a per-function basis. Hence, when an external function is called, Generator has no source code to directly execute it. A naive solution would be stubbing every function call with a `nop` instruction or with a dummy function returning a constant value. However, in this way, the symbolic execution would lose any information of the relationship between the arguments of a function call and its return value. Hence, SYMDIFF would be unable to detect the decompiler's errors caused by incorrect decompilation of an external function call.

While completely and precisely modeling each possible external function call is unfeasible, as a trade-off, we designed an approximation model in which every encountered external function call is modeled as returning a value that is the sum of the least significant byte of its arguments. Consequently, the resulting symbolic expression, although it does not model precisely the behavior of the external call, still keeps track of the relationship between the arguments of the external call and its return value. Specifically, Generator follows the calling convention recovered by the decompiler to extract the values of the arguments of the called function. Additionally, if an argument is of a pointer type, Generator extracts the content located at the address being pointed to by the pointer, rather than the pointer address itself. Finally, Generator generates the sum of the least significant bytes of the extracted values and signed extends this sum to generate the return value.

4.2.2 Comparator

We use Comparator to test the semantic equivalence of two symbolic models, corresponding to the function from the lifted IR and its recompiled version. To do this, Comparator uses two symbolic models to construct a mathematical formula and queries the SMT solvers whether there exists a path that has different return values, i.e., for the given symbolic models whether there is an input that causes the behavior of the symbolic models to diverge.

Specifically, Comparator first maps the arguments of the two functions from the symbolic models, if such arguments are present. If the mapping fails, Comparator reports semantic inequivalent directly. Otherwise, Comparator sets the corresponding function arguments with the same value, dumps the symbolic models of the two functions, makes an assertion that the return values of the two functions are different, and asks an SMT solver whether the assertion is satisfiable. If the SMT solver finds a path difference in these two symbolic models is satisfiable, Comparator reports two functions are not equivalent. Otherwise, Comparator reports two functions are semantic equivalent. Figure 5 provides an example of this procedure. As shown in line 1 of the SMT query, Comparator first maps two arguments and sets them as equal. After that, it dumps the symbolic models of two functions from line 3 to

line 15. Finally, it asserts return values are not the same and asks the SMT solver to check the satisfiability.

4.3 TUNER

As mentioned in Section 2, decompilers use heuristic rules to recover missing information regarding the original source code. During decompilation, we observed that these rules often lead to semantic inaccuracies. TUNER aims to automatically find a valid combination of rules that achieves semantic equivalence, by iteratively toggling different rules until SYMDIFF reports semantic equivalency.

In general, finding the right combination of rules can be time-consuming because of the large search space. In fact, if a decompiler provides n rules, there are 2^n possible combinations. We call the complete combinations of rules the *Configuration Space* (Γ) of a decompiler. Exhaustively finding the right configuration in the entire configuration space of a decompiler has $\mathcal{O}(2^n)$ complexity. To explore a large configuration space efficiently, we introduce two optimizations:
Binomial Search Optimization: We observe that semantic equivalence often can be achieved by toggling a small number of rules, because some rules are harder to implement correctly than others. Therefore, TUNER divides the configuration space of n rules into smaller subsets and chooses an unordered subset of x rules to toggle from the complete set of rules, i.e., $\binom{n}{x}$. For instance, TUNER creates a subset σ_1 by toggling only one rule at a time, i.e., $\binom{n}{1}$. While exploring the configuration space, TUNER starts from covering subset σ_0 until it reaches σ_n, thus evaluating the complete configuration as: $\bigcup_{i=0}^{n} |\sigma_i| = \Gamma$, where n is the number of rules in a decompiler.
Priority-Based Learning Optimization: During tuning, we observe that some combinations of rules repeatedly rectify semantic inaccuracies. After further research, we find that this phenomenon arises either due to bugs within the implementation of rules or incompatibility among rules. As a result, combinations of rules that have proven effective in the past are more likely to rectify semantic inaccuracies for upcoming input functions. For this reason, TUNER uses a *Pre-Learned Dictionary* to record existing effective combinations of rules, i.e., good configurations. The pre-learned dictionary is a map with configurations as keys and a priority number as a value. The priority of the combination of rules is the number of programs fixed by the combination of rules. While exploring the configuration space, TUNER first tries the configuration from the pre-learned dictionary, with a higher priority.

To apply the *Priority-Based Learning Optimization (PBLO)* and the *Binomial Search Optimization (BSO)*, TUNER uses a work queue to search the configuration space. Specifically, TUNER first inserts the configurations from the pre-learned rule dictionary in this work queue in the priority descending order. Next, it adds configuration to the work queue using the *BSO*. After that, TUNER iteratively consumes this work queue. For each iteration, TUNER dequeues the configuration and uses RECOMPILER and SYMDIFF to verify whether, by using the tested configuration, the decompiled code is semantically equivalent to *Lifted IR*. If the current configuration is able to achieve semantic equivalence between the recompiled and lifted IRs, TUNER increases the priority of the configuration if the current configuration already exists in the pre-learned rule dictionary. Otherwise, the configuration is added to the pre-learned rule map with a priority of one.

5 Implementation

We implement D-HELIX as a modular Python framework. Our implementation is designed to be flexible and allows for the easy integration of existing decompilers through the implementation of abstract interfaces. To better evaluate and understand the bugs found by D-HELIX, we choose to work with two popular and open-source decompilers: angr and Ghidra, by instantiating the abstract interface of these two decompilers. We utilize both decompilers in the headless mode, along with their respective lifters , to lift binary code to IR.
RECOMPILER is implemented as a Python front-end to Clang. RECOMPILER captures errors thrown by Clang and fixes them in the source code. Moreover, for pseudo instructions from different decompilers (e.g., CONCAT from Ghdira) RECOMPILER treats them as special function calls and integrates their implementation with the decompiled code at the function level. Specifically, instead of stubbing these instructions, RECOMPILER first implements them as function calls based on the definitions from the decompiler and then links them as a shared library with the decompiled code.
SYMDIFF is implemented as a Python package to provide an abstract interface to support the plugins of different symbolic engines. Currently, we symbolically execute LLVM IR using prompt [59] and symbolically execute both P-code and Vex IR using angr. Specifically, we run prompt with arguments *–posix-runtime* and *–search=bfs* and angr with arguments *auto_load_libs=False, load_debug_info=True,* and *CFGFast*. To run P-code on angr, the argument *engine=angr.engines.UberEnginePcode* is needed during the initialization.

To compare different IRs, we modify symbolic execution engines (i.e., angr and prompt) to generate function-level symbolic models and compare the generated symbolic models by querying SMT solver. Specifically, to generate the symbolic models for all explored paths, we profile the executions of two instructions (i.e., *return* and *conditional branch*). During the execution of *return* instructions (e.g., *RETURN* in P-code, *Ijk_Ret* in Vex IR, and *ret* in LLVM IR), we dump the constraints of return values following the smtlib format. During the executions of *conditional branch* instructions (e.g., *CBRANCH* in P-code, *Ijk_Boring* in Vex IR, and *br* in LLVM IR), we dump their condition constraints and jump targets using the If-Then-Else (ITE) expression format. Meanwhile, for each engine, we implement the handling for external func-

tion calls, loops, and memory accesses following the design specified in Section 4.2.1. We also implement a configurable timeout for all symbolic execution operations. Specifically, we set the timeout as two minutes by default, considering our server's running speed. Lastly, to compare the symbolic models, SYMDIFF utilizes Z3 [19] as the constraint solver.

TUNER is implemented as a Python package. To accommodate different decompilers, TUNER implements an abstraction layer, a Python class that is used for tuning all the rules within decompilers, to enable the search within the configuration space of any decompiler. The size of the configuration space is dependent on the decompiler as shown in Table 1. For our evaluated decompilers, Ghidra has 151 rules, out of which 124 are private rules, i.e., they are not accessible without modifying the source code. We modified 25 SLoC of Ghidra to expose a public interface for accessing every rule. Similarly, angr provides 41 rules, 11 of which are private. We modified 34 SLoC in angr to expose a public interface for these rules. Regarding the termination strategy, we halt TUNER, when it reaches a 2-week running time limitation, considering our server's running speed. This limit is established on the observations that toggling more than two rules (i.e., $\binom{n}{3}$) in Ghidra during the Binomial Search Optimization stage may introduce additional semantic errors. Meanwhile, we observe that TUNER typically completes toggling two rules within two weeks.

6 Evaluation

We apply D-HELIX to real-world decompilers to investigate the following research questions:

1. How effectively can D-HELIX find semantic inaccuracies between the binary and the decompiled code (Section 6.2)?
2. How effectively can TUNER debug the root causes of the detected semantic inaccuracies between the binary and the decompiled code? (Section 6.3)?

To address these questions, we use Ghidra [5] (version 10.0) and angr [1] (version v9.2.30) as the tested decompilers and evaluate D-HELIX on a 112-core Intel(R) Xeon(R) Gold with 1 TB of physical memory.

Our evaluation program set consists of 2,004 real-world binaries and object files from various open-source projects including binaries used in previous literature [12, 55] (i.e., coreutils and util-linux). Additionally, it contains actual binaries and object files from six Linux-based C language projects, selected from the most popular C language projects on GitHub in November 2021. [2] The selected applications include FFmpeg , skynet, masscan, libuv, curl, and openssl. We compile all projects with GCC (version 11.1.0) and Clang (version 16.0.0) using the default optimization settings specified by the projects. Table 4 in the Appendix shows an overview of our evaluation program set. It includes details on how the binaries were compiled, as well as the complexity of the input binaries at the function level. Using our program set, whose source codes contain 86.93k functions, Ghidra was able to decompile 55.4k functions, whereas angr could decompile 37.6k functions.

[2] We exclude non-Linux binaries from other trending projects, due to the limited support by the used upstream tools.

6.1 Findings

D-HELIX found a total of 25 (17 previously unknown) bugs in the two decompilers (Ghidra and angr). We categorize the bugs based on the semantics incorrectly recovered by the decompilers in 11 categories, listed in Table 2. As shown in the table, both decompilers have difficulties in the recovery of the function prototype, variable type, structure's member, and certain instructions. Moreover, Ghidra struggles with identifying boundaries of no-return functions and recovering some literal values. On the other hand, angr struggles with correctly recovering the CFG of the decompiled program. Note that, as mentioned in Section 4.1, SYMDIFF may detect semantic inaccuracies introduced by the RECOMPILER's approach of assigning default types and values to undefined variables. In Table 2, we categorize these inaccuracies, which arise from the absence of initialization, as missing instructions, i.e., rows #5 and #9.

We have reported all bugs discovered by D-HELIX to the relevant parties. As of this writing, 10 out of 17 previously unknown bugs have been acknowledged and resolved by the developers, while the remaining 7 bugs are still under review.

As shown in Table 2, D-HELIX *found a total of 25 bugs in the two decompilers (Ghidra and angr), and 7 of these bugs have been fixed by the developers.*

6.2 Semantic inaccuracies findings

To completely evaluate the effectiveness of semantic inaccuracies findings, we separate the evaluation into two parts, RECOMPILER and SYMDIFF.

6.2.1 RECOMPILER

To evaluate the effectiveness of RECOMPILER, we compile the decompiled functions in our evaluation program set using RECOMPILER and Clang. As shown in Figure 6, for Ghidra, Clang only compiles 24.9% (13,780) functions, whereas RECOMPILER recompiles 72.4% (40,129) functions. Similarly, for angr, Clang compiles 29.1% (10,953) functions, whereas RECOMPILER compiles 45.0% (16,910) functions. As shown above, RECOMPILER compiles 130.6% more functions than Clang, alleviating the challenges of the recompilation.

#	Category of Bugs	Decompiler	No. of distinct bugs	No. of funcs	Affected program set	Bug example:(GT *vs.* DC)
1	Function prototype recovery	Ghidra	7	63	P_1–P_5,P_7,P_8	`f(int a)` *vs.* `f(void)`
2	Literal value recovery	Ghidra	3	6	P_1,P_7,P_8	`x < 128` *vs.* `x < -128`
3	Type recovery	Ghidra	3	222	P_1–P_4,P_8	`-0x18` *vs.* `&DAT_ffffe8`
4	Structure recovery	Ghidra	1	130	P_1–P_4,P_8	`Struct{...}a; a+0x6` *vs.* `Struct{...}a; a+0x12`
5	Missing instructions	Ghidra	1	18	P_1,P_3,P_4,P_8	`int a = 23;` *vs.* `//a is not declared`
6	Function boundary recovery	Ghidra	1	4	P_3	`{f(); ...}` *vs.* `{f(); return;}`
7	Function prototype recovery	angr	4	132	P_1–P_8	`return a < 26;` *vs.* `return a;`
8	CFG recovery	angr	2	204	P_1–P_8	`while(){if...}` *vs.* `while(){...}`
9	Missing instructions	angr	2	8	P_1–P_3,P_4,P_8	`int a = 23;` *vs.* `//a is not declared`
10	Type recovery	angr	1	47	P_1–P_8	`char v1;` *vs.* `unsigned long long v1;`
11	Structure recovery	angr	1	121	P_1–P_8	`Struct{...}a; a+0x6` *vs.* `Struct{...}a; a+0x12`
Total			25	955	P_1–P_8	

Table 2: Summary of bugs discovered by SYMDIFF. Bugs are categorized into different rows according to their category. The fourth column refers to the number of bugs for each error category. The fifth column represents the number of incorrectly decompiled functions for each error category detected by SYMDIFF. The sixth column shows in which project D-HELIX found these bugs and the number corresponds to the projects listed in Table 4 in the Appendix. The last column shows how decompilers incorrectly decompile the GT (Ground Truth) as the DC (Decompiled Code).

We also analyzed the functions that RECOMPILER failed to compile. As shown in Figure 7, for both decompilers, two most common error types are type recovery and structure recovery. Besides these two error categories, RECOMPILER may fail to recover the function pointer, function prototype, etc. The distribution of errors for both decompilers is roughly the same.

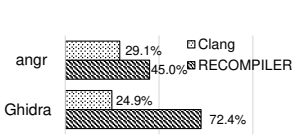

Figure 6: The percentage of functions that can be compiled after using RECOMPILER.

Figure 7: The distribution of different types of errors that cannot be fixed by RECOMPILER.

6.2.2 SYMDIFF

We evaluate the effectiveness of SYMDIFF by utilizing the decompiled functions recompiled by RECOMPILER. Specifically, our evaluation focuses on two metrics: the scalability and the accuracy of SYMDIFF analyses.

Scalability. Our evaluation shows that out of the functions that can be recompiled by RECOMPILER, SYMDIFF can successfully analyze 91.3% (36,628) functions for Ghidra and 93.9% (15,877) functions for angr. To understand the limitations of SYMDIFF, we manually analyze the cases where SYMDIFF fails. Figure 8 summarizes the errors thrown by SYMDIFF. The errors can be categorized into three categories, (1) error from the underlying tools, such as angr, prompt, etc., (2) timeout in the constraint solver due to the complexity of the collected constraints, and (3) unsupported instruction by

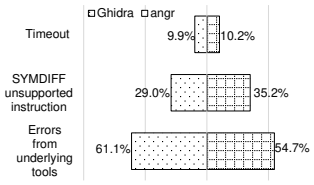

Figure 8: The percentage of decompiled functions that are not fully analyzed by SYMDIFF due to the listed errors.

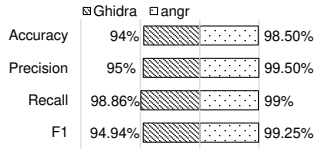

Figure 9: The accuracy, precision, recall, and F1 score of SYMDIFF on the tested decompilers.

SYMDIFF, such as floating point instruction. For the programs decompiled by Ghidra, 55% of the errors were caused by the tools used by our analysis such as angr, prompt, etc. Similarly, for angr, 61% of the errors are related to underlying tools. 29% of the errors are caused by unsupported instructions and 10% of failures are caused by constraint solver timeout.

Accuracy. To evaluate the accuracy of SYMDIFF, we calculate the false positives and false negatives generated by D-HELIX via manually comparing two symbolic models [3]. Since this evaluation requires extensive manual work, for each decompiler, we randomly sample a subset of 200 functions, 100 for examining false positives and 100 for examining false negatives. Figure 9 shows our evaluation result of SYMDIFF on Ghidra and angr, in the field of accuracy, precision, recall, and F1 score.

To understand the limitations of SYMDIFF, we investigate the root cause of false positives and false negatives. Based

[3] Ideally, we should manually examine the differences between IRs, which, however, requires significant human efforts. Hence, we compare the differences at the source code level. And if we observe an inaccuracy in the decompiled code that is not shown in its symbolic model, we claim a false negative.

#	Category	No. bugs	Related Rules	No. funcs	Root Cause
1	Incorrect function prototype recovery	3	DWARF	26	✓
2	Incorrect literal value recovery	1	RuleSubvarSext & RuleIntLessEqual	1	✓
3	Incorrect type recovery	2	Apply Data Archives	33	✓
			X86 Constant Reference Analyze	1	UR
4	Incorrect function prototype recovery	1	Decompiler Parameter ID	11	✓
Total		7		72	

Table 3: Summary of bugs in Ghidra that can be fixed by the TUNER. The last column shows whether the problematic rule found by TUNER is the root cause of this bug. UR illustrates that the bug is still under review.

on our findings, the false positives are all caused by the accuracy trade-offs employed by SYMDIFF to scale symbolic execution, as mentioned in Section 4.2.1. For instance, as shown in Figure 11 in the Appendix, the concretization strategy (e.g., using `malloc` to allocate memory for every symbolized pointer) causes the address of the symbolic pointers between the lifted and high-level source symbolic models varies and generates false positives. The false negatives were all caused by SYMDIFF's dependence on the function prototype recovered by the decompiler. Since the decompiler may fail to recover the correct function's prototype, as shown in Figure 12 in the Appendix, the generated symbolic models may have incorrect external call comparisons.

Hence, D-HELIX *works on the majority of the re-compiled functions (around 92% average of Ghidra and angr) and achieves high F1 scores (around 97% average of Ghidra and angr) in accuracy evaluation, overcoming the challenges of the formalization and semantic inaccuracy detection.*

6.3 Semantic inaccuracy debugging

To completely evaluate semantic inaccuracies debugging of D-HELIX, we separate the evaluation into two aspects, effectiveness (i.e, if TUNER can generate accurate decompiled code for the functions that are reported as inaccurately decompiled by SYMDIFF) and efficiency (i.e, the time required for tuning).

Effectiveness. We use the TUNER to analyze a set of inaccurately decompiled functions reported by SYMDIFF. Since TUNER depends on the heuristic rules provided by the decompiler, we evaluate TUNER for each decompiler separately.

Ghidra. For our program set, SYMDIFF identified 443 semantic inequivalent functions decompiled by Ghidra.

As shown in Table 3, out of these 443 functions, TUNER automatically fixed 16.3% (72) functions. Meanwhile, we identify 7 distinct bugs from these 72 functions. By observing how Ghidra developers fixed the reported bugs, we verified that TUNER correctly identified the root cause (i.e., bugs within the problematic rule) of 6 bugs, accounting for 98.4% (71) of the functions fixed by TUNER. For instance, in the first example of *Incorrect type recovery* (#3), Ghidra maintains a dictionary mapping the function names of well-known libraries (e.g., libcrypto) to their function prototype for faster decompilation. However, the function prototype information could be outdated, resulting in inaccuracies. TUNER fixes this bug by forcing Ghidra to avoid applying this rule (*Apply Data Archives*). For the rest cases that can be fixed by TUNER, we will further explore them in Section 6.4. For the second example of *Incorrect type recovery* (#3)) in Table 3, it is still under review by Ghidra developers as of this writing.

For the remaining 83.7% (371) functions out of 443 semantic inaccuracies, that TUNER could not fix, we manually investigated them. Based on our findings, TUNER fails to fix the inaccuracy if it is triggered by implementation bugs not in the rule but elsewhere, or if toggling all rules implemented by the decompilers is unable to fix the issue. For instance, one of the literal value recovery bugs was triggered by a bug in the code generation system of Ghidra, and hence TUNER was unable to fix it. Similarly, for one of the type recovery issues in Ghidra, TUNER was unable to generate the correct decompiled code, even after trying all permutations of rules related to type recovery. However, for these implementation bugs in Ghidra, TUNER can still help by, for example, ruling out some rule conjunctures, This will be further discussed in the last case of Section 6.4.

angr. For our program set, SYMDIFF identified 4,071 semantic inequivalent functions decompiled by angr. However, TUNER was unable to fix any semantic inequivalence for angr by tuning existing rules. We randomly sample 500 functions and conduct further analysis on them. We found that 30.2% functions have no instruction but a function prototype only (e.g., *int f(){}*), and the inaccuracies in the remaining 69.8% were caused by implementation bugs in angr. For instance, we found that, in angr, the decompilation handler of the `SETcc` instructions was not implemented. However, TUNER can still help, and we will demonstrate how TUNER helps in the second last case of Section 6.4.

D-HELIX *automatically fixes 16.3% detected inaccuracies and successfully identifies the problematic rule for 98% of these inaccuracies. Moreover, for the inaccuracies that cannot be automatically resolved,* TUNER *can still greatly aid developers in tracing their root causes.*

Efficiency. To evaluate the debugging efficiency of semantic inaccuracies, we show how the *Priority-Based Learning Optimization (PBLO)* boosts the efficiency of TUNER by presenting the time reduction achieved through its integration. Since tuning a function consumes considerable computing resources and may easily exceed the 2-week time limit, we select 5 different functions within 5 binaries, which contain the most inaccurate functions reported by SYMDIFF, to demonstrate this comparison. We use TUNER to fix the inaccuracy of these functions and show the time to fix them with and without the *PBLO*. As shown in Figure 10, the dashed bar represents the

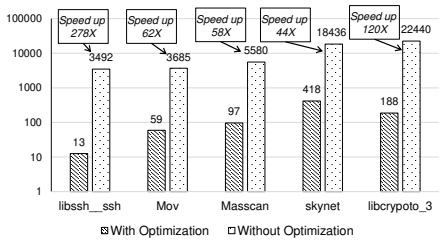

Figure 10: Time consumption in minutes with and without *Priority-Based Learning Optimization (PBLO)*, with annotations indicating the speedup achieved after applying *PBLO*. The y-axis is logarithmically scaled.

```
1://Original code                    1://Decompiled code
2:char temp3;                        2:int iVar1;
3:temp3 = getchar();                 3:iVar1 = getchar();
4:if (temp3 < 128){                  4:if ((char)iVar1<-0x80){
```

Listing 1: Sample code showing 128 incorrectly being decompiled as -128 (-0x80).

time used by TUNER to fix it using the *PBLO*, whereas the dotted bar represents the time used without the *PBLO*. Specifically, in resolving the function in the binary *libcrypoto_3*, *PBLO* accelerates TUNER by 120 times, resulting in a saving of approximately 15 days.

With the help of PBLO, D-HELIX *achieves notable time savings in resolving inaccuracies compared to scenarios without the optimization's assistance.*

6.4 Case studies

In this section, we describe a few interesting case studies highlighting different decompiler bugs discovered by D-HELIX.

6.4.1 Incorrect Literal Value Recovery

Sample Code: Listing 1 provides an example of this semantic inaccuracy that occurred according to D-HELIX. The original code(on the left) checks whether the output received from the `getchar` function is less than 128 (line 4). However, the decompiled code checks if the output from `getchar` is less than -128. As shown in Table 3 case 2, TUNER reveals that this inaccuracy is caused by the application of two specific rules in Ghidra, *RuleIntLessEqual* and *RuleSubvarSext*, and was able to fix this issue by disabling any of them.

Related rule explanation: Internally, Ghidra uses *RuleIntLessEqual* to convert all signed less-than-or-equal-to comparisons with one constant literal operand, e.g. $c <= V$, to a signed less-than comparisons by decrementing the constant literal by one, e.g., $c - 1 < V$. As the value of the constant involved in this step is changed (e.g., c to $c-1$) Ghidra uses another rule, *RuleSubvarSext*, to reduce the width of this constant's representation, if possible (e.g., reduce the width of constant $c-1$ from 32-bit to 8-bit if the value of $c-1$ can fit in an 8-bit sign integer).

Root cause explanation: Upon investigation, we found that the semantic inaccuracy was caused by the following events: ❶ The lifter generates the instruction `128 (0x80) <= iVar1`. ❷ Ghidra applies the rule *RuleIntLessEqual*, which converts `128(0x80) <= iVar1` to `127(0x7f) < iVar1`. ❸ This transformation triggers *RuleSubvarSext*, which changes the width of the constant literal from 32-bit to 8-bit losing signedness information. ❹ Ghidra's readability optimization pass (*ActionPreferComplement*) optimizes the cfg of this `if` branch and reverses the condition from `127(0x7f) < iVar1` to `iVar1 < 128(0x80)`. However, in doing so, the width of the constant literal is not extended from 8 bits back to 32 bits, resulting in a signed char compared to a value -0x80.

```
1:// Original code                  1:// Decompiled code
2:static int                        2:int decode_text_chunk
3: decode_text_chunk                3:  (void){
4: (PNGDecContext *s,
5:  GetByteContext *gb,
6:   int compressed){
```

Listing 2: Sample code shows the decompiled code does not generate any input arguments.

6.4.2 Incorrect Function Prototype Recovery

During our evaluation, we found 194 incorrect function prototype recovery bugs, with 62 of them found in Ghidra and 132 in angr, as listed in Table 2.

Depending on the project, TUNER can fix 47 semantic inequivalence related to three different rules: Debugging With Attributed Record Formats (*DWARF*), Decompiler Parameter ID, and Custom rule. Since the root causes of these inaccurate decompilations are different, we explain them separately as follows.

Debugging With Attributed Record Formats(DWARF):

Sample Code: In Listing 2, the decompiled source code of the function `decode_text_chunk` has the wrong input arguments.

Related rule explanation: Internally, Ghidra uses debug information, stored in the binary in the *DWARF* format, from binary to help recover the function prototype of the decompiled function. However, during the building of the binary, compilers can introduce bugs into the debug information, as shown by previous research [20, 28]. Hence, by disabling the rule *DWARF*, Ghidra will not use the debug information from the binary during decompilation. Disabling this rule allowed TUNER to successfully fix three types of function prototype inaccuracies, as shown in Table 3 case 1, fixing a total of 26 inaccuracies in Ghidra.

Root cause explanation: For these semantic inaccuracies that can be fixed by disabling the rule *DWARF*, our investigation discovered three distinct root causes, outlined as follows:

1) During the compilation of some binaries, compilers erroneously record the debugging information for one function as two distinct debug entities, each providing conflicting details about the function's arguments, e.g., different argument orders. Consequently, when decompiling such a binary, Ghidra's parser in rule *DWARF* struggles to resolve this conflict, leading to the generation of a decompiled function with incorrectly ordered function arguments.

2) For functions with the same name with different arguments (i.e., function overloading), compilers store multiple entities in *DWARF* sections. However, Ghidra may fail to match the correct entity for such a function. Consequently, Ghidra suspends the analysis of this function, which results in its decompiled function lacking arguments, i.e., `void`.

3) The third case arises when one of a function's input arguments is a structure, and this structure is passed from the callee to the caller using multiple registers. In such cases, there could be a mismatch between the number of registers used to pass a function's arguments and the number of function's arguments encoded in the *DWARF* debugging information. Consequently, when Ghidra decompiles this function, the structure will not be decompiled as an input argument but as a list of local variables.

Decompiler Parameter ID:

Sample Code: The decompiled source code has the incorrect function return type, e.g.,`long wvenc_wv_write` is incorrectly decompiled as `int wvenc_wv_write`.

Related rule explanation: In Ghidra, this rule will trigger an analysis that explores the call tree, a tree reduction of the call graph constructed by every function call, to determine the function prototype of the called function in a program. However, Ghidra disables this rule by default. Hence, TUNER automatically enables this analysis and effectively fixes the function prototypes of 11 semantic inaccuracies.

Root cause explanation: Upon investigation, we found Ghidra disables this rule by default since the analysis initiated by it requires a significant amount of time. For instance, during the decompilation of our program set, enabling the rule *Decompiler Parameter ID* results in an average increase of approximately threefold compared to when the rule is disabled.

Custom Rule:

Sample Code: The decompiled source code has the incorrect function return type.

Related rule explanation: In angr, as shown in Section 6.3, TUNER didn't achieve automatic fixes for any inaccuracies by toggling existing rules. Consequently, we developed *Custom rule* to ensure that the function return type matches the return value type. Specifically, we first identify the last return variable in a function and determine whether it is cast to a different type. If so, we set the function return type to the type specified in the cast. If not, we set the function return type to the type of the return variable as declared. As a result, TUNER was able to automatically fix 10 inaccuracies by applying *Custom rule*.

Root cause explanation: Further investigation of the 500 inaccuracies revealed that 121 of the inaccuracies were triggered due to incorrect function return type in the decompiled code. In the remaining 111 cases, where TUNER could not offer a resolution, the underlying problem was traced back to angr's inability to accurately recover the type of the return variable.

```
1:// Original code                  1:// Decompiled code
2:const char *                      2:long argmatch_to_argument(
3:argmatch_to_argument (            3: ...
4: ...                              4: while (true){
5: for (; arglist[i];)              5:  if (*(arglist[i]) != 0){
6:  if (!memcmp (...))              6:   v4 = memcmp(...);
7:   return arglist[i];             7:   return arglist[i];
8: return NULL;                     8:   return 0;
                                    9:  }
                                    10:}
```

Listing 3: Sample code showing how the CFG recovery of a if statement inside for loops is incorrect.

6.4.3 Incorrect CFG Recovery

Sample Code: As shown in Listing 3, the function `argmatch_to_argument` is decompiled with incorrect control flow semantics. Specifically, the check for the return value of the `memcmp` function call at line 6 in the original code (on the left) is not recovered in the decompiled code, resulting in dead code at line 8.

Related rule explanation: Upon reporting this issue, angr developers proposed disabling the rule *EagerReturnsSimplifier* to fix the inaccuracy. In angr, to improve the readability of the code, the rule *EagerReturnsSimplifier* adds additional return statements to the decompiled code, if the number of the "in edges" for the return node (i.e., in-degree of the return site) is less than a specified threshold.

Root cause explanation: Upon further investigation, we discovered that the inaccurate decompilation is caused by a bug in `SequenceWalker`, one of the core libraries used to traverse graphs by angr. Specifically, for each decompiled function, angr constructs a corresponding abstract syntax tree (AST). When angr modifies the CFG (e.g., applies *EagerReturnsSimplifier*), angr calls `SequenceWalker` to traverse the graph and modify nodes, e.g., insert additional return statements on the AST. However, a bug within the `SequenceWalker` causes the incorrect insertion of the node (e.g., return statement) into the AST. Consequently, the parent node (e.g., for loop node) fails to correctly link with the inserted node (e.g., return statement node) on the AST, which causes the return statement at line 8 in Listing 3 being decompiled as dead code.

```
1:// Original code              1:// Decompiled code
2:bool is_fatal(uint_8 code)    2:bool is_fatal(uint_8 code)
3: ...                          3: ...
4: if((1LL << (code & 0x3f)     4: if((1 << (code & 0x3f)
5:  & 0x1000000000000U)!= 0)    5:  & 0x1000000000000U)!= 0)
6:  ...                         6:  ...
7:  if(code < 0xff)             7:  if(code < -1)
```

Listing 4: Sample code shows the decompiled code does not generate correct constants.

6.4.4 Incorrect Constant Recovery

Sample Code: Listing 4 shows two incorrect constant recoveries in the decompiled code. Specifically, in line 4, the 64-bit constant (i.e., `1LL`) in the original code (on the left) is incorrectly decompiled as an `int` constant, i.e., `1`. Consequently, when `code` is `48`, the condition in the original code is true whereas the condition in the decompiled code is false, since the shift operation at line 4 of the decompiled code will be an undefined behavior. Moreover, in line 7, `0xff` in the original code is incorrectly decompiled as `-1`. Since the constant `-1` is treated as `int` type in C99 standard, as a result, the condition in line 7 of the decompiled code will always be false.

Root cause explanation: In Ghidra, constants are treated similarly to global variables, which means rules will be applied to infer their types (both their signedness and their sizes). However, since TUNER reports that no rule is related to these bugs, we infer the type information of these constants has been correctly recovered after the third stage of decompilation, i.e., higher-level semantic abstraction. Hence, we locate the varnodes that store the above two constants and find that their types have been correctly recovered by Ghidra. Since these two constants are correctly handled before the code generation, we, therefore, straightforwardly examine the code generation system of Ghidra and find the constant printing is not appropriately handled. Specifically, Ghidra cannot correctly print some 1-byte constants with edge values (e.g., 0x7f and 0xff) and constants with integer suffixes. We reported this issue within the code generation system to the Ghidra team. As of this writing, these issues have been resolved and included in Ghidra's 10.2 version release by the Ghidra team.

Takeaway: As we can see in the first two cases, D-HELIX can automatically identify the root causes of decompiler bugs and generate accurate decompiled code. Moreover, as shown in the last two cases, even when it's difficult to identify the exact root causes, TUNER proves valuable for decompiler developers during the debugging process, e.g., helps developers rule out some rule conjunctures. Hence, D-HELIX greatly simplifies the work for decompiler developers, saving them a significant amount of time.

7 Discussion and Limitation

Large binaries. As mentioned in Section 6, due to performance constraints, we decided to remove binaries larger than 30 MiB from our program set. In fact, experimentally, we have found that binaries larger than this size require multiple days to be fully analyzed. To this end, we plan to add optimizations such as constraint strength reduction, constraint independence, and constraint caching [14], to alleviate the runtime of SYMDIFF and D-HELIX in general.

Decompiler syntactic quirks. Currently, RECOMPILER can only tackle compilation issues in standard C code as mentioned in Section 6.2. However, some decompilers have their own syntactical quirks to emit code when their decompilation analyses cannot converge. For instance, when Ghidra cannot correctly resolve indirect addresses, it uses the notion of partially resolved address, as shown in this expression: "$var1$._x_y_ = $var2$". This expression means that only y bytes starting with offset x in $var1$ should become equal to $var2$. Currently, we only fix this syntax when $var1$ is an array. We plan to add support for such decompiler-specific syntactic quirks to increase the recompilation rate in future work.

Floating point operations. Currently, SYMDIFF lacks the support for parsing constraints related to floating point operations. However, during our evaluation, we observed that both Ghidra and angr struggle with recognizing floating point operations. Hence, this seems a promising avenue to find semantic bugs in decompilers. We will tackle this problem in our future work.

Memory model in SYMDIFF. D-HELIX currently does not model global variables and considers them as constant values (zero, in case they are not initialized). Additionally, although D-HELIX tries to make the memory model consistent between different symbolic execution engines, there are still some inconsistencies uncovered. For example, in angr, the value of `**param_1` would be set to zero while in prompt it would be set to a symbolic symbol such as `i8`. In future work, we will explore more accurate and consistent memory models.

Supporting a new decompiler. To evaluate a new decompiler with an IR that is not supported by the existing symbolic engine, such as FoxDec [49], D-HELIX requires extra engineering efforts to modify the symbolic execution engine to support a new IR. Specifically, this adaptation involves two steps. Firstly, the existing symbolic execution engine (e.g., angr) must accurately partition the new IR code (e.g., P-code code segment) into basic blocks, following the basic block convention of the supported IR (e.g., Vex IRSB). After that, the existing symbolic execution engine needs to properly handle each new IR instruction within each basic block by translating the new IRs to the supported IRs, thus enabling the use of existing handlers for the supported IRs on the new instructions, e.g., angr translates *cbranch* as *Ijk_Boring* to handle the conditional branch instruction. Overall, we estimate that supporting a new IR on an existing symbolic execution engine (e.g., angr) would require at least 40 days since it took angr

developers 41 days to support P-code [2].

8 Related Work

Existing work recompiles the incomplete C code using different techniques, such as unification-based heuristics [35, 36] and Large Language Models (LLMs) [23]. PsycheC [35, 36] uses unification-based heuristics to generate a well-typed program from an incomplete code. To achieve this, it deduces the missing types of terms by considering their context within the program's abstract syntax tree. However, this approach relies on the data-flow information within the incomplete code. Consequently, for variables in the decompiled code that remain undefined due to insufficient information, this method cannot provide a solution. Additionally, recent research [23] has begun to employ LLMs for the recompilation process. However, as its guiding criteria, i.e., reward function, relies solely on compiler error feedback and code similarity, this approach does not provide a guarantee of semantic preservation in its generated code. Hence, in D-HELIX, we design our own RECOMPILER to improve the recompilation rate, while maintaining semantic preservation as much as possible.

Using symbolic execution for equivalence checking has been explored in other domains [25, 31, 32, 40]. Alive [32] and Alive2 [31] use SMT to conduct code equivalence checks for LLVM IR, focusing on the bugs in LLVM optimizations when handling undefined behaviors. For each function, they encode the input arguments with three types: `undef`, `poison`, or `well-defined`, and determine whether the return value types from the optimized code and the source code are consistent. Since other function aspects (e.g., function prototypes, variable types, and constants) are not considered, their approach cannot detect common decompiler errors, such as the incorrect recovery of function prototypes detected by D-HELIX. MeanDiff [25], a testing tool for binary lifters, uses the symbolic formula to check the semantic equivalence between different IR representations obtained from a single binary instruction. As such, this technique is specifically designed to handle individual binary instruction rather than an entire function. Overall, D-HELIX brings symbolic differentiation into decompilers for the first time, and uses it to vet the whole decompilation process for semantic bugs.

A few research tunes optimizations in compilers for different purposes, such as binary diffing [45, 47] and problematic optimization identification [56]. For instance, Cornucopia [47] generates binaries by iteratively learning from the mutation of compiler optimizations. BinTuner [45] searches near-optimal optimization sequences to generate binaries that maximize the code differences. ODFL [56] differentiates finer-grained optimization to pinpoint problematic optimizations in compilers. However, since the purposes of optimizations in compilers and decompilers differ fundamentally, these approaches do not work properly for decompilers. Hence, in D-HELIX, we build our own TUNER to accurately and efficiently identify the problematic heuristic.

Equivalence checking [17, 33, 43, 58] at the source code level has been explored for different domains. DiffKemp [33] first applies semantics-preserving code transformations between different code patterns for refactoring, then maps variables between the source code and the refactored code, and finally checks the semantic differences between different versions of large-scale C projects by examining the memory states. Churchill et al. [17] first use the program alignment automaton to predict the behaviors of programs, then follow the prediction to align the compared source code, and finally verify the equivalence between different optimized binaries through the memory states. However, these two tools rely on source code availability. Hence, they could not be directly applied to IR code. UC-KLEE [43] automatically synthesizes inputs and runs symbolic execution on the LLVM IR to verify code equivalence between different implementations of the standardized interface with lazy initialization. Similarly, SEC [58] runs symbolic execution on Register Transfer Level (RTL) IRs and compares the terminal states between binaries with different optimizations. Nevertheless, both UC-KLEE and SEC operate on the same IRs, while D-HELIX faces the challenge of comparing different IRs.

9 Conclusion

In this paper, we design D-HELIX, a generic decompiler testing framework that can automatically vet the decompilation correctness. We have fully implemented D-HELIX and applied it to Ghidra and angr. D-HELIX managed to find 17 previously unknown decompilation bugs in Ghidra and angr and helped fix 7 of them automatically.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported in part by NSF under grant NSF CNS-2145744, the Office of Naval Research (ONR) under grant N00014-23-1-2157, and the Defense Advanced Research Projects Agency (DARPA) under contract number N6600120C4031. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or any agency thereof.

References

[1] angr. angr decompiler. https://docs.angr.io/appendix/changelog}angr-8.19.2.4.

[2] angr. P-code. https://github.com/angr/angr/pull/2328.

[3] angr. Vex. https://github.com/angr/vex/blob/master/pub/libvex_ir.h.

[4] Hex-Rays SA. Hex rays decompiler. https://hex-rays.com/decompiler/.

[5] National Security Agency. Ghidra. https://ghidra-sre.org/.

[6] National Security Agency. Pcode. https://ghidra.re/ghidra_docs/api/ghidra/program/model/pcode/PcodeOp.html.

[7] PNF Software. Jeb decompiler. https://www.pnfsoftware.com/.

[8] VECTOR 35. Binary ninja. https://binary.ninja/.

[9] Avast Software. RetDec: A retargetable machine-code decompiler. https://retdec.com/.

[10] 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 *33st USENIX Security Symposium (USENIX Security 24)*, 2024.

[11] Marcus Botacin, Lucas Galante, Paulo de Geus, and André Grégio. Revenge is a dish served cold: Debug-oriented malware decompilation and reassembly. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, New York, NY, USA, 2020. Association for Computing Machinery.

[12] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using {Semantics-Preserving} structural analysis and iterative {Control-Flow} structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, 2013.

[13] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. Decomperson: How humans decompile and what we can learn from it. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2765–2782, Boston, MA, August 2022. USENIX Association.

[14] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[15] Lei Cen, Christoher S. Gates, Luo Si, and Ninghui Li. A probabilistic discriminative model for android malware detection with decompiled source code. *IEEE Transactions on Dependable and Secure Computing*, 12(4):400–412, 2015.

[16] 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 *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, Boston, MA, August 2022. USENIX Association.

[17] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery.

[18] Drew Davidson, Benjamin Moench, Somesh Jha, and Thomas Ristenpart. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, page 463–478, USA, 2013. USENIX Association.

[19] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[20] Giuseppe Antonio Di Luna, Davide Italiano, Luca Massarelli, Sebastian Österlund, Cristiano Giuffrida, and Leonardo Querzoni. Who's debugging the debuggers? exposing debug information bugs in optimized binaries. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 1034–1045, New York, NY, USA, 2021. Association for Computing Machinery.

[21] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. HeapHopper: Bringing Bounded Model Checking to Heap Implementation Security. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 99–116, Baltimore, MD, August 2018. USENIX Association.

[22] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. A comb for decompiled C code. In Hung-Min Sun, Shiuh-Pyng Shieh, Guofei Gu, and Giuseppe Ateniese, editors, *ASIA CCS '20: The*

15th ACM Asia Conference on Computer and Communications Security, Taipei, Taiwan, October 5-9, 2020, pages 637–651. ACM, 2020.

[23] Abhinav Jain, Chima Adiole, Swarat Chaudhuri, Thomas Reps, and Chris Jermaine. Tuning models of code with compiler-generated reinforcement learning feedback, 2023.

[24] JuliaComputingOSS. Llvm-cbe. https://github.com/JuliaComputingOSS/llvm-cbe.

[25] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, Jonghyup Lee, and Sang Kil Cha. Testing intermediate representations for binary analysis. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364, 2017.

[26] Saparya Krishnamoorthy, Michael S. Hsiao, and Loganathan Lingappan. Tackling the path explosion problem in symbolic execution-driven test generation for programs. In *2010 19th IEEE Asian Test Symposium*, pages 59–64, 2010.

[27] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6):216–226, 2014.

[28] Yuanbo Li, Shuo Ding, Qirun Zhang, and Davide Italiano. Debug information validation for optimized code. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1052–1065, New York, NY, USA, 2020. Association for Computing Machinery.

[29] lifting bits. Rellic.

[30] Zhibo Liu and Shuai Wang. How far we have come: testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487, 2020.

[31] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: Bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 65–79, New York, NY, USA, 2021. Association for Computing Machinery.

[32] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, page 22–32, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Viktor Malík and Tomáš Vojnar. Automatically checking semantic equivalence between versions of large-scale c projects. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 329–339, 2021.

[34] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[35] Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. Inference of static semantics for incomplete c programs. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.

[36] Leandro T. C. Melo, Rodrigo G. Ribeiro, Breno C. F. Guimarães, and Fernando Magno Quintão Pereira. Type inference for c: Applications to the static analysis of incomplete programs. *ACM Trans. Program. Lang. Syst.*, 42(3), nov 2020.

[37] microsoft. compiler-warning-c4430. https://learn.microsoft.com/en-us/cpp/error-messages/compiler-warnings/compiler-warning-c4430.

[38] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *European Symposium on Programming*, pages 208–223. Springer, 1999.

[39] George C Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. *CC*, 2:213–228, 2002.

[40] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. N-version disassembly: Differential testing of x86 disassemblers. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, page 265–274, New York, NY, USA, 2010. Association for Computing Machinery.

[41] radeco. radareorg. https://github.com/radareorg/radeco.

[42] David A Ramos and Dawson Engler. Under-constrained symbolic execution: Correctness checking for real code. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 49–64, 2015.

[43] David A. Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 669–685, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[44] Pemma Reiter, Hui Jun Tay, Westley Weimer, Adam Doupé, Ruoyu Wang, and Stephanie Forrest. Automatically mitigating vulnerabilities in x86 binary programs via partially recompilable decompilation. *arXiv preprint arXiv:2202.12336*, 2022.

[45] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. Unleashing the hidden power of compiler optimization on binary code difference: An empirical study. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 142–157, New York, NY, USA, 2021. Association for Computing Machinery.

[46] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 317–331. IEEE Computer Society, 2010.

[47] Vidush Singhal, Akul Abhilash Pillai, Charitha Saumya, Milind Kulkarni, and Aravind Machiry. Cornucopia: A framework for feedback guided generation of binaries. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[48] Uxmal. Reko. https://github.com/uxmal/reko.

[49] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound C code decompilation for a subset of x86-64 binaries. In Frank S. de Boer and Antonio Cerone, editors, *Software Engineering and Formal Methods - 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14-18, 2020, Proceedings*, volume 12310 of *Lecture Notes in Computer Science*, pages 247–264. Springer, 2020.

[50] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle L. Mazurek. An observational investigation of reverse Engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892. USENIX Association, August 2020.

[51] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 8–9, 2017.

[52] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. {LIGHTBLUE}: Automatic {Profile-Aware} debloating of bluetooth stacks. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 339–356, 2021.

[53] Ruoyu Wu, Taegyu Kim, Dave (Jing) Tian, Antonio Bianchi, and Dongyan Xu. DnD: A Cross-Architecture deep neural network decompiler. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2135–2152, Boston, MA, August 2022. USENIX Association.

[54] 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, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 158–177. IEEE Computer Society, 2016.

[55] 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 *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[56] Jing Yang, Yibiao Yang, Maolin Sun, Ming Wen, Yuming Zhou, and Hai Jin. Isolating compiler optimization faults via differentiating finer-grained options. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 481–491, 2022.

[57] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283–294, New York, NY, USA, 2011. Association for Computing Machinery.

[58] Zhenkun Yang, Kecheng Hao, Kai Cong, Sandip Ray, and Fei Xie. Equivalence checking for compiler transformations in behavioral synthesis. In *2013 IEEE 31st International Conference on Computer Design (ICCD)*, pages 491–494, 2013.

[59] Tuba Yavuz and Ken (Yihang) Bai. Analyzing system software components using api model guided symbolic execution. *Journal of Automated Software Engineering*, 2020.

[60] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wenchuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832, 2021.

#	Project name	Archt.	Version	Comp. Optim.	No. of bins&objs	No. of funcs (K)	Avg. binary size (KB)	Avg. locs in funcs	Avg. No. of return stmts in funcs	Pct. of funcs access structure variable	Pct. of funcs contains multi-return stmts	Pct. of funcs contains one pointer
P_1	coreutils	x86_64 AArch64	v9.0	O2	212	14.62	230.18	45.5	1.74	1.68%	32.68%	56.23%
P_2	util-linux	x86_64	v2.37.2	O2	68	4.48	118.56	28.44	1.84	14.55%	50.40%	63.02%
P_3	ffmpeg	x86_64	n4.4.1	O3	1715	42.39	155.55	24.3	2.59	39.63%	49.85%	80.85%
P_4	skynet	x86_64	1.5.0	O2	1	3.58	10,939.0	19.6	2.41	55.76%	57.79%	73.52%
P_5	masscan	x86_64	v1.3.2	O2	1	0.86	2,476.6	40.6	2.13	45.61%	56.67%	76.67%
P_6	libuv	x86_64	v1.42.0	O0	3	2.78	687.79	20.8	5.73	7.55%	50.74%	44.72%
P_7	curl	x86_64	7.80.0	O0	2	3.41	513.09	41.0	1.29	0.65%	24.60%	48.22%
P_8	openssl	x86_64	3.0.0	O3	2	14.67	2066.9	28.6	2.01	9.85%	46.19%	84.79%
Total					2,004	86.93	167.07	29.32	2.37	19.73%	44.38%	69.83%

Table 4: Overview of our program set. The first eight columns show the info of the program set at the binary level, e.g., the number of binaries in each project [4] and how are they compiled. The rest columns show the complexity of the program set at the function level (e.g., the percentage of functions) that contains multiple return statements, pointer variables, or structure variables.

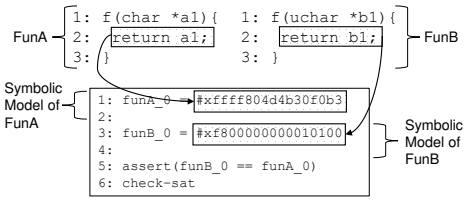

Figure 11: Sample code of a false positive of SYMDIFF.

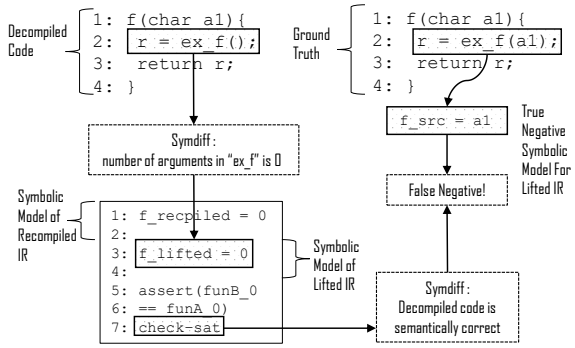

Figure 12: Sample Code of a false negative of SYMDIFF.

A Description of Program Set

Table 4 shows an overview of our program set. Specifically, `Archt.` refers to the architecture of the binary, `Comp. Optim.` refers to the compiler optimization levels of the binary, `No. of bins&objs` refers to the number of binaries and object files in this project, `Avg. size` refers to the average size of binary and object files, and `Pct. of funcs access structure variable` refers to the percentage of functions in this project that contain structure variables. Note that, since some binaries generated by the "ffmpeg" project are more than 30 MiB, directly applying D-HELIX on these large-size binaries significantly increases the running time, e.g., on average, it took SYMDIFF 14 days to finish the symbolic execution on one of the "ffmpeg" binary with size 100 MiB. Hence, we include the object files from "ffmpeg" project to test.