How Far We Have Come: Testing Decompilation Correctness of C Decompilers

Zhibo Liu
The Hong Kong University of Science and Technology
Hong Kong, China
zliudc@connect.ust.hk

Shuai Wang*
The Hong Kong University of Science and Technology
Hong Kong, China
shuaiw@cse.ust.hk

ABSTRACT

A C decompiler converts an executable (the output from a C compiler) into source code. The recovered C source code, once recompiled, will produce an executable with the same functionality as the original executable. With over twenty years of development, C decompilers have been widely used in production to support reverse engineering applications, including legacy software migration, security retrofitting, software comprehension, and to act as the first step in launching adversarial software exploitations. As the *paramount component* and the *trust base* in numerous cybersecurity tasks, C decompilers have enabled the analysis of malware, ransomware, and promoted cybersecurity professionals' understanding of vulnerabilities in real-world systems.

In contrast to this flourishing market, our observation is that in academia, outputs of C decompilers (i.e., recovered C source code) are still *not* extensively used. Instead, the intermediate representations are often more desired for usage when developing applications such as binary security retrofitting. We acknowledge that such conservative approaches in academia are a result of widespread and pessimistic views on the decompilation correctness. However, in conventional software engineering and security research, how much of a problem is, for instance, reusing a piece of simple legacy code by taking the output of modern C decompilers?

In this work, we test decompilation correctness to present an up-to-date understanding regarding modern C decompilers. We detected a total of 1,423 inputs that can trigger decompilation errors from four popular decompilers, and with extensive manual effort, we identified 13 bugs in two open-source decompilers. Our findings show that the overly pessimistic view of decompilation correctness leads researchers to underestimate the potential of modern decompilers; the state-of-the-art decompilers certainly care about the functional correctness, and they are making promising progress. However, some tasks that have been studied for years in academia, such as type inference and optimization, still impede C decompilers from generating quality outputs more than is reflected in the literature. These issues rarely receive enough attention and can lead to great confusion that misleads users.

*Corresponding Author

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397370

CCS CONCEPTS

• **Software and its engineering → Software reverse engineering**; *Software testing and debugging*.

KEYWORDS

Software Testing, Reverse Engineering, Decompiler

ACM Reference Format:
Zhibo Liu and Shuai Wang. 2020. 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 (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3395363.3397370

1 INTRODUCTION

A software decompiler recovers program source code by examining and translating executable files. C decompilers are among the most fundamental reverse engineering tools for software re-engineering missions [18, 35, 63], and they have also laid a solid foundation for many cybersecurity applications, including malware analysis and off-the-shelf software security hardening [22, 25, 26]. To date, many C decompilers exist on the market, including commercial tools that cost several thousands of US dollars and also free versions actively maintained by the open-source community. Remarkable commercial decompilers on the market, including IDA-Pro [34] and JEB3 [52], are "must-have" gadgets for reverse engineers and security analysts despite their high prices. Free decompilers maintained by the open-source community, such as RetDec [38] and Radare2 [2], have also started to challenge the dominance of their commercial competitors. Recently, the National Security Agency (NSA) has also released its decompiler framework, Ghidra [48, 50], with the aim of "training the next generation of cybersecurity defenders."

Despite being the core component of most reverse engineering tasks in industry, our observation is that C decompilers, particularly their final-stage outputs (i.e., the recovered C code), are *not* extensively used in academia. While C decompilers are frequently employed in academia as the basis of analyzing legacy software, such as malware clustering, firmware analysis, and security retrofitting [18, 22, 25, 26, 35, 63, 67], the recovered intermediate information is usually preferred in conducting research rather than decompiled C code. For instance, to fix a security flaw in an executable, the convention is to perform binary patching and to edit the executable file after reverse engineering program layouts and locating the issue [27], even though the straightforward way is to decompile the executable, instrument the decompiled code, and recompile the hardened code into a new executable file.

We interpret such *conservative* approaches as being due to the widespread and potentially pessimistic stance on decompiled C

code; one might expect that the decompiled C code is primarily designed as presentable high-level descriptions of input executables, instead of being directly used as a conventional C program. However, recent several years of progressive development on "functionality-preserving" disassembling and C style control structure recovery [17, 31, 47, 64, 65, 67] illustrates that functionality-preserving decompilation of unobfuscated and not-highly-optimized executable is primarily a matter of engineering effort (although a certain amount of readability is sacrificed). Indeed, some popular reverse engineering frameworks are tentatively implementing such techniques to guarantee decompilation correctness in the first place [28, 55]. Overall, we argue that the research community lacks to incorporate an *up-to-date* understanding of de facto C decompilers and the correctness of their outputs, which may impede reaching the full potential of modern C decompilers in conducting research. Thus, this work aims to study C decompilers in a realistic setting and to more clearly delineate the decompilation correctness of modern C decompilers.

In this research, we perform systematic testing to reflect the decompilation correctness of C decompilers, which is certainly not fully understood and can lead to a controversial view between industry hackers and academic researchers. In particular, we target x86 to C decompilers whose inputs are x86 executable files, with such decompilation being deemed as highly challenging and popular. We aim to answer the following important research questions: **RQ1**: *how difficult is it to recompile the outputs of modern C decompilers?*; **RQ2**: *what are the characteristics of typical decompilation defects?*; and **RQ3**: *what insights can we deduce from analyzing the decompilation defects?* Our findings can be adopted to promote the development of decompilers and to serve as guidelines for users to avoid potential pitfalls.

This work is the first to conduct a comprehensive study targeting C decompilers. We employ Equivalent Modulo Inputs (EMI) testing, a random testing technique that has achieved major success in revealing compiler bugs, in this new setting [19, 40, 60]. We also organize a group of security analysts to carry out extensive manual inspection on findings yielded by EMI testing (about 530 man-hours in total). From a total of 10,707 programs used for this study, we found 1,423 programs exposing decompilation errors from four widely-used decompilers, two of which (IDA-Pro [34] and JEB3 [52]) are popular commercial tools, and the other two (RetDec [38] and Radare2/Ghidra [50]) are actively developed and maintained by the community and by NSA. Manual inspection on the open-source decompilers (RetDec and Radare2/Ghidra) detects 13 buggy code fragments that incur the decompilation errors we found. In sum, this research makes the following contributions:

- To study C decompilers in a realistic setting and to delineate their up-to-date capabilities, we introduce and advocate a new focus, conducting comprehensive and large-scale testing on C decompilers. Findings obtained in this study will guide future research that aims to use and improve decompilers.
- We reuse well-established compiler testing techniques in this new setting and form a productive workflow to reveal potential decompiler bugs. From two de facto commercial decompilers and two popular free decompilers, we successfully found 1,423 programs causing decompilation errors. We have reported all the findings to the decompiler developers, and by the time of writing, typical defects have been promptly confirmed by commercial decompiler vendors and to be fixed. We manually confirmed 13 bugs which caused *all* errors found in outputs of free decompilers.
- We made various observations and obtained inspiring findings regarding modern decompilers. We show that the overly pessimistic stance on the recompilability leads researchers to *underestimate* the potential of C decompilers; modern decompilers are making encouraging progress to enhance quality of their outputs. In contrast, subtler issues that lead to erroneous and unreadable decompiled outputs, including the type inference failure and over optimization, frequently exist and do not receive enough attention.
- We have released all the findings and our tool for decompiler testing to facilitate further research [8, 9]. Our artifact has passed the ISSTA Artifact Evaluation check and been awarded the Functional badge. Other decompilers can be tested with our tool following the same procedure.

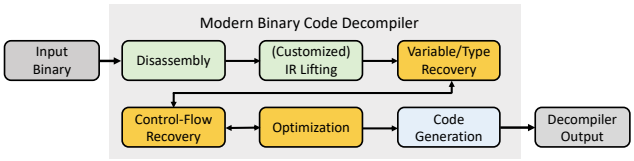

Figure 1: The workflow of C decompilers. We use different colors to differentiate decompilation stages. The key focus is the middle stage (modules in orange) where variables, types, and high-level control structures are recovered.

2 BACKGROUND OF RESEARCH

2.1 Pipeline of C Decompilers

Fig. 1 depicts a high-level overview of modern C decompilers. Multiple stages are involved, and the output of one stage is the input of the next stage.

Front End: Disassembling. Input executables will first be fed into the disassembling module to translate binary code into assembly instructions. The data sections within each executable will also be identified for further usage. Existing research of disassembling has been shown to work very well in practice and the proposed methods can smoothly disassemble large-size binary executables [13, 39, 65]. Nevertheless, the disassembled outputs (i.e., assembly code) are seldom used for analysis; a modern decompiler will first lift assembly instructions into an intermediate representation (IR), which is deemed as more analysis-friendly [16, 59, 61].

Middle Stage: High-Level Program Recovery. The ultimate goal of a C decompiler is to convert input executable into high-level source code. Therefore, given the lifted IR code, the central focus is to recover variables, types, and high-level program control flow from low-level IR code. To recover program variables, some tools already proposed and implemented needed static analysis and inference techniques [10, 12, 13, 30, 54]. To recover variable types, constraint-based type inference systems are typically formulated [42, 49].

To recover C-style control structures, modern decompilers implement a set of structure templates and search to determine whether an IR code region matches the predefined patterns. Some advanced techniques enable an iterative refinement to polish the recovered structure. To date, techniques have been designed to guarantee the structure recovery correctness and also to improve readability [17, 67]. In addition, modern decompilers usually design optimizations to polish the lifted IR code, including dead code elimination and untiling [17, 21, 38]. Also, reverse engineering of C executable files might encounter "chicken and egg" problems (e.g., data flow analysis relies on precise output of control flow analysis, and vice versa). Indeed, modules within the middle stage can be invoked back and forth for iterations; the output of one module is used to promote the analysis of other modules [38].

Back End: Code Generation. The final stage translates IR statements into C statements and outputs source code. As the output of the middle stage, the IR code is already close to being source code. Translations are mostly mundane (but could still contain bugs, as will show in Sec. 5.1) by concretizing code generation templates.

2.2 Equivalence Modulo Inputs (EMI) Testing

We briefly introduce a well-established testing technique that has achieved prominent success in testing compilers, the Equivalence Modulo Inputs (EMI) testing [40]. Soon, we will show that EMI testing can be used to test decompilers in a highly effective manner.

Given a program p and its legitimate input space as $\mathsf{dom}(p)$, the output of p for an input $i \in \mathsf{dom}(p)$ is denoted as $[\![p]\!](i)$. Therefore, two programs p and q are defined as equivalent modulo inputs (EMI) in case $\forall i \in I\ [\![p]\!](i) = [\![q]\!](i)$ *where* $I \subseteq \mathsf{dom}(p) \cap \mathsf{dom}(q)$. Here, q is referred as the EMI variants of p. The main benefit of EMI testing is the functionality-preservation of q w.r.t. inputs $i \in I$; therefore, this method does not require a reference implementation and it provides an explicit testing oracle such that any EMI variants q must behave identically compared with p. The EMI equivalent property alleviates the notion of "program equivalence." It consequently provides a viable way to produce programs for testing compilers. To extend the above formulation in testing decompilers, we start by generating q, which is a mutated program of p (mutation strategies are introduced in the next paragraph). Then, we decompile the executable file compiled from q and generate the decompiled source code q^*. Obviously, q^* is an EMI variant of p and we use the above oracle for testing.

The first EMI implementation [40] generates a mutated program q by profiling p with inputs $i \in I$ and deleting or inserting additional statements within uncovered code blocks w.r.t. to any input $i \in I$. Since any modified code region in q is not executed w.r.t. I, q is naturally an EMI variant of p. Some improvements insert code into the live code region; the inserted code is guarded by opaque predicates that are always evaluated as false during runtime [60]. Some statistical methods are also adopted to optimize the selection of statements for mutations [41]. We employ all three EMI mutation strategies in our new setting (see Sec. 4.1).

3 MOTIVATION

In this section, we show the research topic is *central* and *timely*, by shedding light on potential mismatches between traditional conservative stance of using decompiled C code and the progressive development of decompilation techniques. We also discuss the pitfalls of leveraging IRs for binary code analysis to advocate the development of decompilation techniques.

3.1 Research Using C Decompilers

C decompilers are one of the most critical reverse engineering tools that enable various cybersecurity and software re-engineering missions. To date, reusing decompiled x86 and ARM binary code has been a widespread practice, and industry hackers have successfully decompiled and reused complex real-world legacy software, such as video games [1, 3] and complex firmware [4].

In contrast, while decompilers are also extensively used in academia (searching a popular decompiler, "IDA Pro", in Google Scholar returns 1,370 records since 2015), reusing decompiled C code is not a common approach. Overall, the major application scope of C decompilers in academia includes code comprehension (e.g., similarity analysis) [18, 25, 26], code reuse [27, 37], and various security hardening and vulnerability detection applications [22, 32, 35, 37, 63]. However, our observation is that instead of directly taking final-stage decompiled outputs, intermediate information or representations are usually extracted and leveraged in the conducted research. For instance, instead of directly reusing the decompiled C programs, binary code reuse tasks would be launched with executable patching or replication after reflecting the code layouts and fragments with decompilers [27]. Note that binary patching and replication are usually error-prone and incur a very high cost.

We attribute this conservative and potentially mismatched usage of decompilers to the generally pessimistic views of the decompiled programs: it is traditionally believed that the recovered C code cannot be used for recompilation (unlike conventional C code). Nevertheless, within recent years, promising lines of research have proposed fool-proof techniques for binary disassembling and decompilation [17, 31, 64–67]. As a result, *functionality-preserving* disassembling of non-obfuscated and not-highly-optimized binary code becomes practical, and it could be accurate to assume recovering functionality-preserving C code becomes a matter of engineering effort, although a certain amount of readability of the generated code is sacrificed, e.g., due to the usage of inline assembly or springboard [29, 31]. Indeed, some popular binary analysis frameworks have implemented the proposed techniques [55]: the decompiled output can be smoothly recompiled into an executable that preserves the original semantics [28]. Overall, existing research has shown a practical need to advocate the functionality-preserving "recompilability" as a critical design goal of C decompilers. Given the encouraging development of functionality-preserving reverse engineering, we see this as the perfect time to launch systematic testing of C decompilers, as a means to demystify their full capability.

3.2 Analysis and Instrumentation with IR

At present, a decompiler framework often provides an IR lifted from the assembly instructions in support of static analysis and instrumentation. Given the widespread adoption of decompiler IRs for analysis and instrumentation, we argue that using an IR, although not obvious, *does not* primarily eliminate the need for promoting decompilation. In addition to the self-evident reason

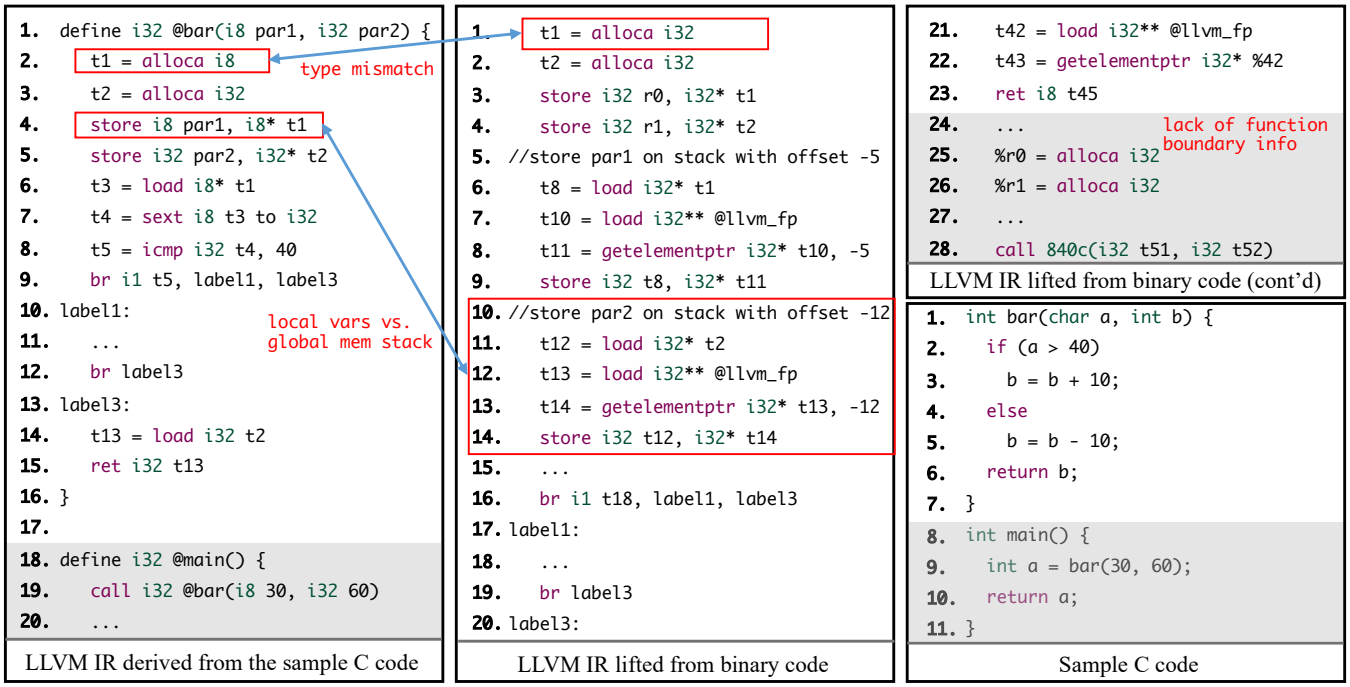

Figure 2: Comparison between LLVM IR generated from sample C code with IR lifted from binary code compiled from the C code. The sample C code is presented on the lower right corner. LLVM IR code is simplified due to the limited space and the `main` function is backgrounded with `grey` for better readability.

that many infrastructures and algorithms (e.g., a symbolic execution engine) need to be reimplemented when analyzing customized IRs, performing static analysis and instrumentation regarding a unified IR, e.g., LLVM IR, suffers from similar challenges in variable and type recovery as that encountered in decompilation.

While lifting assembly code into LLVM IR is mostly mundane, the transformed IR lacks high-level expressiveness, and this absence can impede many standard dataflow analyses and symbolic reasoning facilities [10]. In Fig. 2, we study and present a simple example, where a toy C program was compiled into an LLVM IR and further compiled into an executable. We then disassembled the executable into assembly instructions and transformed the instructions into LLVM IR statements. Comparing the two pieces of IR code, we find that much of the high-level program information is missing in IR derived from low-level code, which, as elaborated in [10, 30], will hinder the adoption of many source code-level static analyses due to the missing of program high-level information. Clearly, using IR *does not* primarily eliminate the need for promoting decompilation techniques, since they face the same hurdle regarding type, (local) variable, and control structure recovery.[1] In other words, reverse engineering of high-level data representations are *unavoidable* to facilitate the same amount of analysis expressiveness (although recovery of high-level control structures may not always be needed).

[1]We are certainly aware of some LLVM IR lifters developed by the reverse engineering community [15, 46, 62]. However, experimental tests show that they suffer from similar issues (e.g., lack of local variable recovery), and to date, such lifters are *not* commonly adopted in academia. In fact, researchers prefer to implement their own in-house IR lifter for such tasks [22].

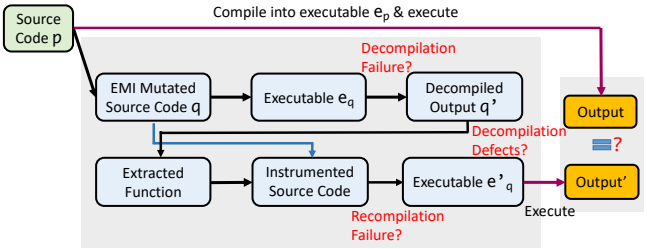

Figure 3: Workflow of our study. Workflow elements (p, e_q, e'_q, etc.) are consistent with description in Alg. 1.

Overall, we interpret that analyzing IR code *does not* alleviate need for decompilation, but, instead, could fall into a trap consisting of a "multilingual" scenario where we have to mediate source code-level analysis facilities with low-level code and their incompatible memory models. In summary, our observation again advocates the understanding and development of modern decompilers, which would overcome many common challenges in the first place and enable a seamless integration of source code-level analysis facilities with low-level code study.

4 METHODOLOGY AND STUDY SETUP

Fig. 3 depicts the workflow of this empirical study. We start by generating random C programs with a popular generator for compiler testing, Csmith [68]. For each input C code p generated by Csmith,

we mutate p with the EMI technique (see Sec. 2.2), and compile the mutated program q (i.e., an EMI variant) into executable e_q. The decompiler will take e_q as the input and produce the decompiled output, another piece of C code. Given that the decompiled C code is not directly recompilable (for the explanation, see Sec. 4.3), we extract the function from the decompiled output (we configure Csmith to generate C code with one function besides `main`) and use this function to replace its corresponding code chunk in q to generate an instrumented q^*. Doing so gives us another piece of executable file e^* compiled from q^*. To check decompilation correctness, we compare the execution outputs of e (compiled from the input C code) and e^*. If a deviant output is identified, we manually inspect and compare p and q^* in depth (see Sec. 4.2) and aggregate the harvested information to deduce empirical findings.

Study Scope. As depicted in Fig. 3, we capture the following kinds of issues in this study:

- **Decompilation failures** represent errors or crashes thrown by decompilers. This usually indicates a simple decoding bug or ill-format input executables.
- **Recompilation failures** represent errors yielded by the compiler when we are recompiling decompiled outputs. This indicates bugs or implementation limits in decompilers.
- **Decompilation defects** represent deviant results when we execute and compare recompiled executable and its reference input. Deviant outputs imply the semantics of the input executable is broken in the decompiled output, which is likely derived from a decompiler bug.

We aim to reveal and understand various logic bugs or implementation pitfalls that result in incorrect decompiled C code. Therefore, the *recompilation failures* and *decompilation defects* are the major focus (see Table 3 for the findings). While our main focus is not *decompilation failures* since fuzz testing tools could likely find such issues during in-house development, decompilation failures are still recorded and reported in Table 3.

The prerequisite for binary decompilation is disassembling; decompilation is performed to lift the disassembled output into higher-level representations (Sec. 2.1). As mentioned above, while precise disassembling is known to be hard in principle, current algorithms have been shown to work very well in practice and to perform fool-proof disassembling of real-world applications [31, 47, 64, 65]. Therefore, in this study, we assume that binary disassembling is *reliable*; we focus on analyzing defects in decompilation procedures, where existing research has rarely explored.

We are *not* testing extreme cases to stress decompilers. Decompilers are known to be error-prone for highly optimized and obfuscated code. Indeed, obfuscated and highly-optimized code are not considered by prior "functionality-preserving" disassembling and decompilation research as well [11, 17, 31, 64, 65, 67]. Instead, we aim to understand to what extent modern decompilers are revamped with respect to conventional C programs and provide practical and inspiring insights for decompiler developers and users. We limit our study to unobfuscated and unoptimized binaries compiled on x86 platforms (see discussions regarding other settings in Sec. 6).

Algorithm 1 Decompiler Testing.

```
1:  function IsDeviant(p, q)
2:      e_p ← Compile(p)
3:      e_q ← Compile(q)
4:      q' ← Decompile(e_q)
5:      e'_q ← Compile(Instrument(q, q'))
        ▷ Implementation of Instrument is given in Sec. 4.3
6:      if Execute_and_compare(e_p, e'_q) == false then
7:          return true
8:      else
9:          return false
10: function Testing(P)
        ▷ P: a set of C programs generated by Csmith
11:     S ← ∅
12:     for each p_k in P do
13:         if IsDeviant(p_k, p_k) == true then
14:             add (p_k, p_k) in S
15:             continue
16:         q_k ← p_k
17:         for 1 ... MAX_ITER do
18:             q*_k ← Mutate(q_k)
19:             if IsDeviant(p_k, q*_k) == true then
20:                 add (p_k, q*_k) in S
21:             if Rand(0, 1) < A(q_k → q*_k, p_k) then      ▷ Formulation of
    A(q_k → q*_k, p_k) can be found in [8].
22:                 q_k ← q*_k
23:     return S
```

4.1 Equivalence Modulo Inputs (EMI) Testing

As mentioned above, in this research, we reuse a well-developed compiler testing technique named EMI testing [19, 40, 60]. Recall during decompilation (Sec. 2.1), modern decompilers perform data flow and control flow analyses and transformations, which are conceptually comparable to compiler passes. Therefore, we envision that the full-scale mutations enabled by the EMI technique can adequately expose decompiler defects. Our study confirms this intuition: EMI testing is highly effective in this new setting (for details, see our findings in Sec. 5).

Alg. 1 specifies the workflow of the EMI testing. Function ***Testing*** is the main entry point of our algorithm, and ***IsDeviant*** performs the compilation, instrumentation, and comparison to find deviant outputs of two given programs. In particular, ***IsDeviant*** compiles input programs p and q into two executable files, e_p and e_q, respectively; then, it decompiles e_q into another piece of program q' (line 4). We then instrument q' to generate a recompilable program (for the implementation of ***Instrument***, see Sec. 4.3) and further compile the code into executable e'_q (line 5). We then execute these two executable files and compare the execution output (line 6). Note that a program generated by Csmith *does not* require user-provided input; it performs random computations and returns a checksum of its global variables. Therefore, we directly execute programs and compare their outputs, the checksum of global variables.

The input of ***Testing*** is a set of arbitrary C programs generated by Csmith, and this function iterates each C program p_k for the testing (line 12–22). Before performing the EMI mutation, we first use ***IsDeviant*** to check the recompilation correctness of the seed program p_k (line 13), and in case a deviant output is found, we record this case (line 14–15) and move to the next program p_{k+1}.

For each seed program p_k, we iterate the EMI mutation for *MAX_ITER* iterations (*MAX_ITER* is set as 30 in the implementation). At each iteration, we generate a new variant q^*_k given the

current variant q_k as the input (line 18). `Mutate` subsumes mutations of both live code and dead code regions [40, 60]; we identify the live code region of p_k by executing it and recording the covered statements. While the mutation of dead code regions is mostly straightforward, being performed by inserting and removing unreachable statements, the mutation of live code regions is slightly trickier. Following existing research on mutating live code [60], we insert a set of opaque predicates (we implemented all three opaque predicate schemes proposed in [60]) into the live code region for mutation. When program pairs of deviant outputs are found, we add the pairs into $\mathcal{S}$ (line 20).

Additionally, instead of the blind mutation strategy (i.e., randomly selecting some statements for mutation) proposed in the first EMI paper [40], we reimplement an advanced EMI mutation strategy guided by the Markov Chain Monte Carlo (MCMC) optimization procedure to explore the search space [41]. For each new EMI variant p_k^*, we compare it with variant p_k and compute a program distance, indicating how different they are (line 21). We accept p_k^* with an acceptance ratio (line 22). This method is designed to generate more diverse EMI mutations progressively since EMI variants with high distance values will lead to a higher chance to be kept [41]. We formulate this MCMC optimization in [8]. After collecting all the program tuples with deviant outputs (line 23), we resort to a manual study to comprehend the root causes of these suspicious outputs.

4.2 Manual Inspection

As mentioned above, Csmith-generated C code computes a checksum of its global variables as the execution output. Once a deviant output is identified, we manually pinpoint erroneous statements causing such deviants in the decompiled C code. The manual inspection forms a typical debugging process: starting from a global variable that yields a deviant value, we identify all of its assignments and recursively backtrack the deviant values used for assignments until we identify the root cause (i.e., the erroneous statements) in the decompiled EMI variants.

This step is costly: examining and checking all suspicious findings took two reverse engineering analysts about 180 man-hours. Each analyst has an in-depth knowledge of reverse engineering and rich experience in binary code analysis and CTF competitions. In this way, we ensure the accuracy of our study and the credibility of our findings to a great extent. We have reported our findings to the developers, and some typical cases have been promptly confirmed and to be fixed (see Sec. 5). Moreover, with about 350 man-hours, we locate 13 buggy code fragments in open-source decompilers that lead to erroneous statements in the decompiled C code (see discussions in Sec. 5.1).

4.3 Study Setup

The proposed study is implemented in Python, with 3,707 LOC (measured by `cloc` [23]). We now discuss challenges and practical solutions involved in setting up this study.

Challenges for Recompilation. The standard EMI technique employs a straightforward testing oracle by comparing execution results of a seed program and its EMI variants. However, shortly our findings will show that it is difficult to directly recompile the outputs of decompilers (see Sec. 5.2.2 for the details). Our study shows that many *undefined* symbols (e.g., ELF binary-specific symbols) commonly exist in the global data sections of the decompiled outputs. Below, we discuss solutions to address this issue.

Implementation of *Instrument*. We acknowledge the difficulty of recompiling the decompilation outputs; to enable an automated workflow for testing and empirical study, we seek to extract functions from the decompiled C code and use the decompiled functions to replace their corresponding code chunk in the source code of the decompiler input (see "Instrumented Source Code" in Fig. 3). We then execute the Csmith-generated C code and its instrumented EMI variant and check for any execution result deviation.

The intuition is that typically within each decompiled function, symbols (e.g., local variables) are defined in a complete manner, and therefore, the whole chunk becomes a "closure." In fact, as just mentioned, the undefined symbols are mostly placed in global data sections, and based on our observation, they usually do not interfere with computations within each function. To perform this extraction, we configure Csmith by bounding the maximum number of functions in its output as one (in addition to `main`). As will be reported in our findings, this method revives the "recompilability" of modern decompilers when processing most C programs.

Handling Discarded Global Variable Names. Names of most variables are discarded after compilation, and in the decompiled outputs, variables are renamed meaninglessly (`v0`, `v1`, etc.). Although doing so does not affect the usage of local variables, this study requires to correctly resort the usage of global variables, since global variables are frequently referred by Csmith-generated code (recall that the execution result of a Csmith-generated program is the checksum of its global variables).

We address this issue by creating a ghost local variable for each global variable, and replace the usage of global variables with their corresponding local variables. Consider the example below:

```
int ga;
void set_var(int la){
  // synchronize global var. with local var.
  ga = la;
}
void foo(){ // Csmith generated function
 int la;
 ... // usage of ga are replaced by la
 set_var(la);
 // compute checksum of ga and print output
}
```

After compiling and decompiling, the local variable name (`la`) is discarded; however, global variable `ga` will be updated before exiting `foo` since its corresponding local variable in the decompiled code is still kept as the parameter of `set_var`. We configure Csmith to avoid the usage of C pointers, and therefore, we do not need to resolve alias issues when identifying the usage of global variables.

Configurations. We use Csmith (ver. 2.3.0) [68] to generate seed C programs. `gcov` (ver. 7.4.0) is employed to obtain code coverage

information and to identify live code for EMI mutations. We compile Csmith's outputs with gcc (ver. 7.4.0) into 32-bit x86 binary code with no optimization.

Tentative studies show that seed C programs with complex data structures impede recompilation. Therefore, in this study (except Sec. 5.2.2), we configure Csmith and ensure that its outputs contain only a subset of C grammars. We exclude C struct, union, array, and floating points. As mentioned in this section, we disable pointers to simplify the manual study and avoid alias analysis when creating ghost local variables for global variables. Note that while this configuration simplifies data structures, Csmith still generates programs with *highly complex* control structures, arithmetic operations, and type castings.

Intuitively, decompilation problems are more obvious for large and complex software. However, we note that using large software, while could likely provoke errors, is not feasible in this research. As mentioned in Sec. 4.2, we manually inspect every erroneous decompiled code to understand which statement is decompiled wrongly. This manual inspection already takes about 180 man-hours. Errors in complex and large software could make the manual inspection too costly or even infeasible. Nevertheless, our findings on real-world decompilers are general enough to affect the decompilation of any C code and therefore fixing our findings will presumably promote the decompilation of large and complex C software.

Decompilers. Table 1 reports the decompilers used in our study. IDA-Pro [34] and JEB3 [52] are both the state-of-the-art commercial decompilers that have been widely used in many research and industry projects. To strengthen the generalizability of our study, we also evaluate RetDec [38] and Radare2 [2], two popular free decompilers that are actively maintained by the community. The implementation details of commercial decompilers are mostly obscure to the public. In contrast, RetDec shares a very promising vision to bridge binary code analysis with the LLVM ecosystem. It is built as a reverse engineering frontend of the LLVM framework, and users are allowed to implement their analysis and instrumentation passes using LLVM. Radare2 recently integrates the Ghidra decompiler [50] developed by NSA. The Ghidra plugin leverages the front-end of Radare2 for disassembling, and use the Ghidra decompiler (version 9.1) to convert the disassembled code into C code. Our tentative test shows that Ghidra plugin has much better decompilation accuracy than the native decompilation support of Radare2 which is still immature.

These four decompilers, to the best of our knowledge and experience, represent the best two commercial and best two non-commercial C decompilers. We indeed tentatively explored other decompilers in our study. For instance, Snowman [5] is seen to produce worse decompiled outputs compared with these four decompilers, and there is no manual provided [7].

Decompilers are generally designed for an "out-of-the-box" usage. We cannot find options to configure optimizations (not like compilers) or decompiling algorithms. We also cannot find documents shipped with these decompilers on how to configure them. Hence, all decompilers are studied in the standard setting.

Statistics of Test Cases. Table 2 reports statistics of the C programs used in the study. We use Csmith to randomly generate

Table 1: Decompilers employed in the study.

Tool Name	Information
IDA-Pro [34]	Commercial
JEB3 [52]	Commercial
RetDec [38]	Free; maintained by the community
Radare2/Ghidra [2, 50]	Free; maintained by the community and NSA

Table 2: Statistics of the C programs used in the study. Line of code (LOC) is measured with `cloc` [23].

Total # of programs generated by Csmith	1,000
Total LOC in Csmith generated C programs	142,888
Total # of EMI variants	9,707
Total LOC in EMI variants	2,361,590

1,000 C programs. These 1,000 programs are the seed inputs of EMI mutation for each decompiler. The total number of generated EMI variants is 9,707 (see Table 3 for the breakdown). As mentioned in this section, we configured Csmith to produce one function (in addition to `main`) for each C code it generates. Each seed program (on average 148 LOC) contains a medium size function with presumably complex control structures and many global variables.

5 FINDINGS

Table 3 provides an overview of our findings. Out of in total 9,706 test cases (exclude one decompilation failure), we find 408 (4.2%) recompilation failures. While most decompiled C code can be successfully recompiled and executed, 1,014 (430+584; 10.4%) outputs are erroneous: the decompiled C code shows deviant execution results compared with the seed. JEB3 outperforms the other three decompilers, given less decompilation defects (30+9). Despite one decompilation failure in JEB3 (errors thrown by the decompiler), all the cases can be decompiled smoothly.

We further put these decompilation defects into five categories (under the "Characteristics" column) by identifying and classifying erroneous statements in decompiled C code. As mentioned in Sec. 4.2, we form a group of reverse engineering analysts to classify the findings manually. This ensures the accuracy of our research. Nevertheless, we admit the difficulty, given the large number and highly-optimized decompiled C code (see Sec. 5.3.4 for corresponding discussion). Therefore, we do classification at our *best effort*. The first three columns stand for errors (in total 861) found in types, variables, and control flow structures of decompiled C code. We also find 147 errors that are presumably due to decompiler optimizations. "Others" report 6 errors that are likely due to bugs in the IR-to-C translation stage. We give further discussions and case study regarding each category in Sec. 5.3.

Processing Time. Our experiments are launched on a machine with Intel Core i5-8500 3.00 Hz CPU and 4 GB memory. Although processing time is in general not a concern for decompilation, we record and report that it takes on average 5.0 CPU seconds to decompile one case. We interpret that de facto decompilers perform efficiently in processing commonly-used binary code.

Confirmation with the Decompiler Developers. We have reported all the failures and defects found in this research (findings

Table 3: Result overview. As depicted in Alg. 1, if a Csmith generated program p_0 has inconsistent functionality compared with its decompiled output p_0^*, we skip the EMI mutation on p_0 and directly report it as one decompilation defect (the "Csmith Output" subcolumn). Otherwise, we generate 30 EMI mutations from each p_0 as EMI testing inputs. The total number of EMI mutations are reported in "# of EMI Variants." We manually analyzed each defect and summarized findings in "Characteristics".

Tool Name	# of EMI Variants	Decompilation Failures	Recompilation Failures	Decompilation Defects		Characteristics of Decompilation Defects				
				Csmith Output	EMI Variant	Type Recovery	Variable Recovery	Control-Flow Recovery	Optimization	Others
IDA-Pro	3,786	0	208	1	69	2	0	4	61	3
JEB3	2,510	1	13	30	9	25	7	0	4	3
RetDec	907	0	187	346	380	315	338	25	48	0
Radare2/Ghidra	2,504	0	0	53	126	35	87	23	34	0
Total	9,707	1	408	430	584	377	432	52	147	6

in Table 3) to the decompiler developers. To seek for prompt confirmation and insights into our results, we also select at least one example in each defect category and present to the developers.

The JEB3 developer was responsive in confirming the selected cases for each defect category. He even mentioned to tentatively include some findings in the upcoming major update. To quote him:

> *"thanks so much for all that feedback!! - next update ... I hope to include some of your reports! ..."*

We also quote the IDA-Pro author's feedback below:

> *"We internally use Csmith to test the decompiler and we know that our decompiler can not handle all cases yet. We are working on them."*

We interpret that the IDA-Pro developers certainly care about the *functional correctness* of decompilation. At the time of writing, we are waiting for the response from RetDec and Radare2/Ghidra. Overall, we understand that these decompilers are developed by either small companies (like the commercial ones) or volunteers, and it certainly takes a while for confirmation and to incorporate our findings into the scheduled development pipeline. Nevertheless, responses from IDA-Pro and JEB3 developers indicate that they take our findings seriously.

5.1 Identify Buggy Code Fragments in Decompilers

We seek to pinpoint the buggy code fragments in decompilers that lead to these defects. IDA-Pro and JEB3 are commercial tools and therefore we have no way of performing root cause analysis. Radare2/Ghidra and RetDec both have source code available online, although their accompanying documents and code comments are merely provided. At this step, we spent *extensive* manual efforts (over six weeks; in total about 350 man-hours) to analyze all 913 (187 + 346 + 380) decompiler flaws detected from RetDec (in total 178,732 LOC), and all 179 (53 + 126) decompiler flaws in Radare2/Ghidra (the Ghidra plugin has 112,999 LOC). As reported in Table 4, 13 buggy code fragments are found from these two decompilers.[2] To clarify potential confusions on Table 4, recall to decide the "characteristics" of decompilation defects, we manually analyzed decompiled C code w.r.t. its input seed. We summarized erroneous code statements into five categories at our best effort

[2]This section reports and discusses high-level information in the main paper. Information regarding each buggy code fragment, including their locations, descriptions and samples erroneous outputs they could incur, can be found in [8].

Table 4: Buggy code fragments found in RetDec (3nd to 10th rows) and Radare2/Ghidra (11th to 17th rows). Column names, simplified due to the limited space, shall be easily figured by referring to Table 3.

	Recompile Failure	Decompilation Defects				Total
		Type	Var.	Control.	Opt.	
Bug_1	0	297	0	0	0	297
Bug_2	0	0	338	0	0	338
Bug_3	0	0	0	11	0	11
Bug_4	0	18	0	14	4	36
Bug_5	0	0	0	0	44	44
Bug_6	177	0	0	0	0	177
Bug_7	10	0	0	0	0	10
Total	187	315	338	25	48	913
Bug_8	0	31	87	0	0	118
Bug_9	0	0	0	6	10	16
Bug_{10}	0	0	0	3	3	6
Bug_{11}	0	1	0	5	0	6
Bug_{12}	0	0	0	5	21	26
Bug_{13}	0	3	0	4	0	7
Total	0	35	87	23	34	179

in Table 3. Nevertheless, with root cause analysis, we find that decompiler bugs may cause different errors in decompiled code. For instance, Bug_8 is a type recovery bug in Radare2/Ghidra, directly generating 31 decompiled C programs with type errors. Furthermore, given local variables of wrong types, optimization may treat these variables as part of other variables (see Sec. 5.3.2), outputting 87 decompiled C programs with missing variable errors.

All of these findings are logic bugs, causing erroneous outputs rather than decompiler crash or abnormal termination. From these 13 bugs, 12 are found from the "middle stage" of decompilation (Sec. 2.1) where decompilers perform high-level program representation recovery, while one (Bug_{13} found in Radare2/Ghidra) is in the C code generation phase: unsigned shift operation in the Ghidra IR was lifted into a signed shift in C code. From the 12 middle stage bugs, four bugs are in the type recovery modules, two bugs are in the variable recovery modules, and six bugs are in the optimization modules. Also, while Radare2/Ghidra is stated to leverage the disassembly infrastructure of Radare2 and bridge with Ghidra, we find that Radare2/Ghidra "freerides" the type recovery utility of Radare2. Bug_8 in Radare2/Ghidra indeed roots from incorrectly recovered variable types in Radare2.

5.2 Recompilation Study

We start by answering **RQ1**: *how difficult is it to recompile the outputs of modern decompilers?* To that end, we conduct a two-step study

in this section. We first evaluate the recompilation of decompilers by directly compiling their outputs. Given the general challenge in the first step, we then resort to instrument the decompiled outputs (with ***Instrument*** defined in Sec. 4.3) and analyze the remaining recompilation failures reported in Table 3.

5.2.1 Full-Scale Recompilation. We first study full-scale recompilation by directly recompiling the recovered high-level C code of de facto decompilers. To do so, we use the default option of Csmith to randomly generate 100 C programs and feed to the decompilers.[3] We then recompile their outputs and collect compiler messages.

We report that we are unable to recompile any of the decompiled outputs into functional executables. Plenty of *undefined* symbols exist in the decompiled outputs, including ELF binary specific symbols (e.g., symbols used for dynamic linkage), decompiler specific symbols or undefined variables. In short, we interpret that outputs of de facto decompilers are not directly recompilable, and more importantly, by putting decompiler and executable specific symbols in their outputs, recompilation seems not the top priority for de facto decompilers (although "readability" in the decompiled outputs is also not well supported, as we will show in Sec. 5.3.4).

Therefore, in the rest of the section, we follow what has been proposed in our study setup (Sec. 4.3) and discuss the recompilation failures of the instrumented decompilation outputs. We also present a corresponding discussion below in Sec. 5.2.3.

5.2.2 Recompilation Failure. We now discuss the causes of the recompilation failures reported in Table 3. We manually checked the compiler errors for the 408 recompilation failures and identified two key reasons that impede recompilation.

Erroneous Variable Recovery. We find many variable recovery errors in the decompiled outputs. For instance, when decompiling a C program with only one local variable with JEB3, the output is translated into the following statement:

```
// source code
unsigned int a;
```

```
// decompiled code
unsigned int v0,v0;
```

Despite the difficulty to locate the buggy component in JEB3 that causes such re-declaration issue, we report that similar problems were found in a considerable number of JEB3's outputs.

We also found recompilation failures due to incorrect recovery of function call parameters (found in JEB3 and RetDec). Consider the example below:

```
// source code
int a = 12;
int b = 10;
set_var(a,b);
```

```
// decompiled code
int a = 12;
int b = 10;
set_var(a);
```

where the decompiled output causes an inconsistency in the declaration and invocation of function `set_var`. In fact, manual inspection in Sec. 5.1 shows that all 187 recompilation failures of RetDec root from two bugs (i.e., Bug_6 and Bug_7 in Table 4) in the function prototype recovery module. In other words, fixing these two bugs of function call parameter recovery would eliminate all 187 failures.

[3]To clarify potential confusions, these 100 C programs are only used as a quick check on recompilation at this step. All the other testing and studies are from a set of 1,000 C programs, as explained in the study setup (Sec. 4.3).

Undefined Symbols. Despite that undefined symbols are mostly eliminated in this new setting, still, such issues can be found in IDA-Pro's outputs. In particular, we report that ELF binary specific symbols (e.g., symbols of Global Offset Table used for dynamic linkage) seem to be placed in outputs of IDA-Pro commonly. In contrast, the other three decompilers work generally well to avoid the abuse of undefined symbols.

5.2.3 Result Implication. Our study on full-scale recompilation (Sec. 5.2.2) shows that outputs of decompilers are still not directly recompilable. However, one previously-ignored fact revealed in this study is that generating recompilable code of many non-trivial C programs is essentially the *last mile* of modern decompilers. As shown in this research, after some *syntax-level* tweaks, most decompiled outputs become recompilable. Radare2/Ghidra shows highly encouraging findings with zero recompilation failure, and both commercial decompilers have less than 5.0% recompilation failure. IDA-Pro fails 208 cases by inserting extra ELF binary specific symbols, most of which are used for dynamic linkage. Our study further indicates the possibility to safely remove some of them since the linker will need to create these symbols in the ELF executable files when recompiling. Also, while RetDec has 187 recompilation failures, our root cause analysis shows that these errors are due to only two bugs in the function parameter recovery module, which shall be fixed together smoothly.

Although academic researchers were generally believing that outputs of C decompilers are not for reuse and not even recompilable, this study re-scopes this pessimistic stance by demonstrating that after some systematic and straightforward syntax-level changes (without any manual tweaks), modern C decompilers show decent capability of recompiling non-trivial C programs (recall our C programs have relatively simple data structures but *complex* control structures, arithmetic operations, and type castings). Therefore, we summarize and present our first finding as follows:

> **Finding:** By applying straightforward syntax-level changes without any manual tweaks, modern C decompilers already show good support of recompilation for many non-trivial C programs.

Existing research has shown a practical need to advocate the "recompilability" as a critical design goal, if not foremost, of decompilers. This study shows that it requires only syntax-level changes to revive the decompiled outputs of many non-trivial C codes. The recompilability can drastically promote the reuse of legacy software, and becomes the "next big thing" in the community. Indeed, starting from a piece of reassembleable assembly code by reusing existing research tools [31, 64, 65], we envision the feasibility to deliberately craft decompilation passes and deliver recompilability-preserving decompilation, by making readability a secondary consideration (e.g., preserving certain onerous instructions with inline assembly) or framing it in terms of good faith effort.

5.3 Decompilation Defects

This section answers **RQ2**: *what are the characteristics of typical decompilation defects?* Table 3 classifies all the deviant outputs into five categories. In the rest of this section, we elaborate on typical errors within each category.

5.3.1 Type Recovery. The lack of fool-proof type recovery is one key limit of existing decompilers. While for C code, not all the type recovery failure breaks the semantics, in this study we flag a considerable amount of recovered variable types breaking semantics (in terms of both control and data flow). Consider an example below:

```
// source code
int i = 0;
for(i=6; i<-12; i-=6){
  ... // not reachable
}
```

```
// decompiled code
unsigned int i = 0;
for(i=6; i<-12; i-=6){
 ... // reachable
}
```

where the Csmith-generated C code has a **for** loop, but will execute for zero iterations. Contrarily, the type recovery incorrectly annotates `i` with **unsigned int** in the decompiled C code, and the loop body becomes reachable in the decompiled output. In other words, the type recovery alters the *control flow*, and further changes the semantics.

We also find a considerable amount of type recovery failure that leads to an erroneous *data flow*. Consider a case below found:

```
// source code
int32_t a = 0xaaffff;
```

```
// decompiled code
int16_t a = 0xaaffff;
```

In the decompiled output, the variable type **int32_t** is incorrectly recovered as **int16_t**. As a result, `a` is initialized with a different value, since a variable of **int16_t** type can only take the lowest 16 bits (0xffff).

As reported in Table 4, 315 type errors in C code decompiled by RetDec root from two bugs (Bug_1 and Bug_4). Bug_1 happens by converting signed `mov` statements (`movsx`) in x86 assembly into IR statements without correctly considering extensions and truncations. Bug_4 missed certain function call parameters when analyzing the call site stack push operations. Bug_8, Bug_{11}, and Bug_{13} cause in total 35 type errors in Radare2/Ghidra. Bug_8 cannot precisely infer the "length" of stack variables, while Bug_{11} performs constant propagation optimization and uses a ghost variable defined in Ghidra of wrong type to replace original variables. As aforementioned, Bug_{13} lifts unsigned shift in IR into signed shift in C code.

5.3.2 Variable Recovery. We find 338 variable recovery issues from RetDec, where this decompiler seems unable to recognize certain variables and future leads to deviant execution outputs compared with the seed programs. Root cause analysis shows that Bug_2, a bug on function parameter recovery, incurs all these 338 errors. Radare2/Ghidra can also miss to identify certain local variables. Root cause analysis shows that due to bugs in type recovery module of Radare2 (i.e., Bug_8), variable is incorrectly assigned with a larger size (e.g., a 32-bit integer). Variables adjacent to this "larger" variable can overlap in the memory layout, and can be potentially deemed as part of this "larger" variable and is therefore optimized out.

We also find seven erroneous variable recoveries in JEB3 which causes "undefined behavior" in the decompiled C code. Consider a simplified case:

```
unsigned int v0;
unsigned int v1 = v0 >> 16;
```

where `v0` is defined yet uninitialized, leading to an indeterminate value in `v1`. We have confirmed that the source code does not contain any uninitialized local variables. In general, generating code with undefined behavior is undesired for decompilers. Although the root cause is yet to be determined, we suspect that `v0` in the above case is hardcoded in the output. We urge the decompiler developers to avoid generating code exhibiting undefined behavior, for instance, by initializing `v0` with zero.

5.3.3 Control-Flow Recovery. We find a total of 52 deviant outputs due to control-flow errors in the decompiled C code. We note that after taking a close look at the buggy code fragments, *all* failures in RetDec and Radare2/Ghidra are actually due to wrong type recovery and optimization bugs. Consider a simplified input program below:

```
// condition generated by EMI
if (opaque_condition){ // evaluated to false
  statement1; // not reachable
}
```

where the `opaque_condition` will be evaluated to false during runtime. However, we report that the `if` condition was optimized out in the decompiled program since the condition is incorrectly evaluated to "true" due to type recovery or optimization errors. Hence, the unreachable branch becomes reachable, reflecting a "control-structure" error in the decompiled C code. We report that Bug_{3-4} found in RetDec and Bug_{9-13} in Radare2/Ghidra all lead to such issues, although they represent different buggy code fragments in the type recovery and optimization modules.

We also find issues where the statements are mistakenly reordered in IDA-Pro outputs. Consider the case below:

```
// source code
if (cond1)
  statement1;
if (cond2)
  statement2;
```

```
// decompiled code
if (cond2)
  statement2;
if (cond1)
  statement1;
```

where in the decompiled outputs, two `if` statements are reordered, altering execution flow, and leading to deviant execution outputs.

5.3.4 Optimization. Academic researchers design decompiler optimizations to make decompiled code close to the input source code and thus more "readable" [17, 67]. However, optimizations, particularly the erroneous constant folding and constant propagation, can make decompiled C code mal-functional. As aforementioned, from the 13 bugs reported in Table 4, six are within the optimization modules of decompilers, for instance performing constant folding without correctly taking bit length into account (Bug_5), or incorrectly using a 32-bit ghost variable defined in Ghidra to replace a 16-bit variable during constant propagation (Bug_{11}).

In addition to errors, we note that the overly (and wrongly) optimized C code often becomes *too concise*. Such aggressive optimization has diminished the readability of decompiled C code to a great extent, and has caused a major challenge for our manual inspection (Sec. 5.1). While "readability" could be a subjective criterion, inspired by our observation, we measure the readability by 1) reporting the average LOC for the decompiled code, and 2) analyzing the code similarity between the decompiled outputs and the input programs. We leverage a popular software similarity analyzer, `moss` [6], to measure code similarity. The similarity score (ranging from 0 to 1.0; higher is better) indicates how close the decompiled outputs and inputs are. The results are reported as follows:

	IDA-Pro	JEB3	RetDec	Radare2/Ghidra
LOC	90	85	54	104
Similarity Score	0.51	0.50	0.54	0.49

Decompiled programs are highly concise, in the sense that their average LOC is much lower than the corresponding LOC of input programs (on average 143; see Table 2). Even worse, the average similarity score between decompiled outputs and input programs is also low. In summary, we interpret that the highly-optimized decompiled outputs obstruct the readability notably (see further discussions in Sec. 5.5).

Also, the above results may (incorrectly) indicate that decompilers "consistently" optimize their outputs, since their average similarity scores w.r.t. reference inputs are close. Nevertheless, Sec. 6 will show that indeed for an input executable, its corresponding outputs of four decompilers have very different representations, indicating a strong need for regulating optimization and code generation.

5.4 Others

We find six errors in the "Others" category: we report that the decompiled outputs have syntax-level difference (e.g., arithmetic operators) compared with the input source code. While the root cause is unknown (since these two decompilers are closed source), we suspect such issues are due to sloppy errors in the C statement translation stage, which could be fixed by developers easily.

5.5 Result Implication

In this section, we present discussion and result implication to answer research question **RQ3**: *what insights can we deduce from analyzing the decompilation defects?*

Support of Cutting-Edge Research Outputs. We have found plenty of type, variable, and control structure errors in the decompiled C code, and explained their corresponding buggy fragments in the decompilers. Although academic researchers are believed to have mostly addressed such reverse engineering challenges (since we are evaluating non-trivial but *not extreme* cases), one observation is that modern decompilers have not fully implemented those well-established research products. For instance, while the state-of-the-art research has been working on function prototype recovery for years and achieved promising results (e.g., close to 99% accuracy for function recognition in x86 binaries [14, 20, 58]), still, the de facto decompilers (e.g., RetDec) have not implemented the proposed methods and therefore make lots of errors in recovering function prototypes and parameters. Also, recovering types from x86 binary code have been formulated as a recursively-constrained type inference approach with sub-typing, recursive types, and polymorphism [42, 49]. Contrarily, C decompilers, from the disclosed documents and our observation, only implement simple inference techniques combined with heuristics and predefined patterns [33, 38, 53].

> **Finding:** The de facto decompilers still have not fully leveraged the research outputs in this field to improve reverse engineering accuracy.

Although research products cannot be used to address every corner case, most defects exposed in this study, such as RetDec's obvious limit in recovering function prototypes, shall be fixed smoothly. Overall, while modern decompilers perform decently in recovering high-level source code, we urge developers to embrace research products in this field to revamp the design and solve problems exposed in this study.

Optimization. As disclosed in Sec. 5.3.4, our study on de facto decompilers uncovers the following finding:

> **Finding:** De facto decompilers extensively simplify their outputs, even though readability and decompilation correctness are undermined simultaneously.

We encountered major difficulty when manually inspecting erroneous outputs that are highly simplified. We suspect that if it was not even presentable for us — reverse engineering analysts — to comprehend the decompiled outputs, it should be accurate to assume the outputs are often not readable enough for layman users. This clearly indicates a *mismatch* between expectations in the literature and the actual capabilities of modern decompilers. Note that in academia, the optimization module of decompilers is designed following the principle of *enhancing* the readability and making it close to the original C code [17, 67].

Although optimization can help to simplify code emitted by decompilation passes and can usually output one succinct high-level statement by folding several statements, we advocate fine-grained calibration. Currently, modern decompilers seem to go too far and notably hurt readability of their outputs. Meanwhile, motivated by how compiler optimizations are provided for usage, we urge decompiler developers to make their products configurable to flexibly select optimization passes.

6 DISCUSSION

Limitations and Threat to Validity. We now give a discussion of validity and shortcomings of this paper's approach. In this research, *construct validity* denotes the degree to which our metrics actually reflect the correctness of C decompilers. Overall, we conduct dynamic testing and manual inspection to study the outputs of de facto decompilers. Hence, while this practical approach detects decompiler bugs and reveals inspiring findings, the most possible threat is that our testing approach cannot guarantee the functional correctness of decompilers. We clarify that our work roots the same assumption as previous works in this line of research that aim to comprehend the functionality of reverse engineering toolchains with dynamic testing rather than static verification [36, 51].

We check the correctness of decompiled C code by comparing its execution output with its reference input program. Considering the execution output of each Csmith generated program is a checksum of all its global variables, decompilation errors on global data or its involved computations can be faithfully exposed. However, a possible threat is that defects can be neglected in the decompiled C code, in case they do not contribute to the execution output. One promising mitigation is to enable a static viewpoint of the decompiled output. Instead of executing the decompiled code, we envision opportunities to perform whole-program comparison and pinpoint inconsistency. We leave exploring this direction for future work.

Besides, there exists the potential threat that the proposed decompiler testing framework may not adapt to other types of programs,

since the conducted research focuses on C code decompilation. Nevertheless, we mitigate this threat to *external validity* by designing an approach that is language and platform independent. As a result, our approach is applicable to other settings outside the current scope. We believe the proposed technique is general, and we give further discussions regarding other decompilation settings soon in this section.

Decompiler Developers' Responsibility. We consider that developers should take the responsibility to constructively address the findings in this research. Our work serves as the first and systematic effort to provide guidelines. In Sec. 5.5, our findings have shown that modern decompilers still have not leveraged the full potential of research outputs. Our research sheds light on where developers can start to enhance their products, e.g., avoiding extensively simplifying the decompiled C code (see Sec. 5.3.4).

Looking ahead, we also advocate decompiler developers to embrace breakthroughs of "semantics-preserving" reverse engineering [17, 31, 47, 64, 65]. Therefore, decompiled outputs could become fool-proof "recompilable" in the first place. We also envision the need to deliver more principled techniques to verify the functional correctness of decompilation. Meeting this need will have a prominent, long-term impact in the reverse engineering community.

Cross Comparison of Decompiled C Code. Careful readers may wonder the feasibility of conducting a static "cross comparison", by decompiling the same executable with a set of decompilers and identifying differences in their outputs. However, we note that decompiled C code can have drastically different representations since different decompilers implement their own tactics and translation templates (although they share identical semantics). Here, we compile 500 programs randomly generated by Csmith into executable files. For each executable file, we use four decompilers to decompile it and cross compare the similarity (also with `moss` [6]) of four decompiled C code. We report the average similarity score as follows:

	IDA-Pro	JEB3	RetDec	Radare2/Ghidra
IDA-Pro	×	0.73	0.69	0.66
JEB3	×	×	0.68	0.65
RetDec	×	×	×	0.68
Radare2/Ghidra	×	×	×	×

The average cross similarity score is indeed low (on average 0.68), which sheds light on practical needs to advocate more consistent representations and regulations. Overall, we leave it as one future work to explore practical methods to perform cross comparison, for instance by extracting certain "semantics-level" invariants.

Other Settings. The main focus of this study is C decompilation, one challenging and fundamental task commonly encountered in real-world cybersecurity and software re-engineering missions. While the current experiments is conducted on x86 platforms, given popular decompilers like IDA-Pro and RetDec can handle different processors and formats (e.g., ARM and NIPS), we envision opportunities for the research community to generalize our findings, since the key issues, including both recompilation and decompilation defects, are mostly platform and language *independent*.

Decompiling bytecode (e.g., Android apps) is easier, and the quality of decompiled code is generally deemed as higher. Indeed, the Android repackaging attack has become an "out-of-the-box" practice, for which correctly decompiling bytecode is the pre-requisite. In contrast, decompiling non-trivial C++ code, for instance recovering its class hierarchy, is still an open problem [56]. We leave it as one further work, to generalize methodologies proposed in this work on studying other popular decompilation settings.

7 RELATED WORK

Testing techniques have been used to measure *static reverse engineering* tools. For instance, differential testing has been used to validate disassemblers [51]. Recent research uses symbolic equivalence checks (with symbolic execution and constraint solving) to pinpoint bugs in IR lifters of binary executables [24, 36]. The security community has also conducted remarkable empirical studies regarding the accuracy and usage scenarios of de facto disassemblers [11].

State-of-the-art *dynamic reverse engineering* activities mostly use symbolic execution techniques and virtual machine (VM)-based monitoring to capture abnormal and malicious behaviors of suspicious binary code. Existing research has proposed various testing techniques to examine the security and reliability of these dynamic reverse engineering tools. Red pill testing [44, 45, 57] leverages the random or differential testing (typically with a black-box setting) to compare the behavior of a VM and that of a physical machine when executing with the same input. Recent research has promoted the testing of a low-confidential emulator with inputs generated by analyzing a highly confidential emulator [43].

Existing research has laid a solid foundation on testing static and dynamic reverse engineering tools. However, a thorough and complete testing of decompilers is still the missing piece in the understanding of today's reverse engineering landscape. There is a demanding need to gain insights into how much of a problem decompilation is, given its indispensable role in building cybersecurity and software reuse applications.

8 CONCLUSION

We have performed a systematic study to investigate decompilation correctness of modern C code decompilers. Our large-scale evaluation on four popular commercial and free decompilers successfully found considerable decompiler flaws. In addition, we elaborately explained findings and summarized lessons we have learned from this study. We show that modern C decompilers have been progressively improved to generate quality outputs. Nevertheless, some classic reverse engineering challenges, including type recovery and optimization, still frequently impede modern decompilers from generating well-formed outputs. This work could provide guides for researchers and industry hackers that aim to use and improve C decompilers, and is presumably adaptable to test other decompilation settings.

ACKNOWLEDGMENTS

We thank the anonymous ISSTA reviewers for their valuable feedback. Our special thanks go to the JEB3 and IDA-Pro developers who provided us with much help, insight and advice.

REFERENCES

[1] 2014. Starcraft Reverse Engineered to run on ARM. https://news.ycombinator.com/item?id=7372414.
[2] 2016. radare2. http://www.radare.org/r/.
[3] 2018. Diablo devolved - magic behind the 1996 computer game. https://github.com/diasurgical/devilution.
[4] 2018. Firmware Mod Kit. https://github.com/rampageX/firmware-mod-kit.
[5] 2018. Snowman decompiler. https://derevenets.com.
[6] 2019. Moss: A System for Detecting Software Similarity. https://theory.stanford.edu/~aiken/moss.
[7] 2019. Output of nocode Invalid C++ Code. https://github.com/yegord/snowman/issues/196.
[8] 2020. Decompiler Flaws and Root Cause Analysis. https://www.dropbox.com/sh/kqw7e19snfeukai/AADHZ45TAL9Kxi7v9nmdXfLCa?dl=0.
[9] 2020. Decompiler Fuzzing Test with EMI mutation. https://github.com/monkbai/DecFuzzer.
[10] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. 2013. A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System. In *EuroSys '13*.
[11] Dennis Andriesse, Xi Chen, Victor van der Veen, Asia Slowinska, and Herbert Bos. 2016. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In *USENIX Sec*.
[12] Gogul Balakrishnan and Thomas Reps. [n.d.]. DIVINE: DIscovering Variables IN Executables. In *VMCAI 2007*.
[13] Gogul Balakrishnan and Thomas Reps. 2010. WYSINWYX: What You See is Not What You eXecute. *ACM Trans. Program. Lang. Syst.* 32, 6, Article 23 (Aug. 2010), 84 pages.
[14] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association.
[15] Ahmed Bougacha. 2016. Dagger. https://github.com/repzret/dagger.
[16] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. 2011. BAP: A binary analysis platform *(CAV)*.
[17] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*. 353–368.
[18] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-architecture cross-OS Binary Search *(FSE)*.
[19] Yuting Chen, Ting Su, Chengnian Sun, Zhendong Su, and Jianjun Zhao. 2016. Coverage-directed Differential Testing of JVM Implementations. In *PLDI*.
[20] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 99–116.
[21] Cristina Cifuentes. 1994. *Reverse compilation techniques*. Queensland University of Technology, Brisbane.
[22] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *USENIX Sec*.
[23] Al Danial. [n.d.]. CLOC. https://goo.gl/3KFACB.
[24] Sandeep Dasgupta, Sushant Dinesh, Deepan Venkatesh, Vikram S Adve, and Christopher W Fletcher. 2020. Scalable Validation for Binary Lifters.
[25] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *ASPLOS*.
[26] Yaniv David and Eran Yahav. 2014. Tracelet-based Code Search in Executables. In *PLDI*.
[27] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. 2013. BISTRO: Binary Component Extraction and Embedding for Software Security Applications.
[28] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2018. rev.ng: A Multi-Architecture Framework for Reverse Engineering and Vulnerability Discovery. In *ICCST*.
[29] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. Rev.Ng: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *CC*.
[30] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable Variable and Data Type Detection in a Binary Rewriter. In *PLDI*.
[31] Bauman Erick, Lin Zhiqiang, and Hamlen Kevin W. 2018. Superset Disassembly: Statically Rewriting x86 Binaries Without Heuristics. In *NDSS*.
[32] Ivan Gotovchits, Rijnard van Tonder, and David Brumley. 2018. Saluki: finding taint-style vulnerabilities with static property checking. In *NDSS*.
[33] I. Guilfanov. 2001. A Simple Type System for Program Reengineering. In *WCRE*.
[34] SA Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler and debugger.
[35] Anastasis Keliris and Michail Maniatakos Yakdan. 2019. ICSREF: A Framework for Automated Reverse Engineering of Industrial Control Systems Binaries. In *NDSS*.
[36] Soomin Kim, Markus Faerevaag, Minkyu Jung, SeungIl Jung, DongYeop Oh, JongHyup Lee, and Sang Kil Cha. 2017. Testing Intermediate Representations for Binary Analysis. In *ASE*.
[37] Taegyu Kim, Chung Hwan Kim, Hongjun Choi, Yonghwi Kwon, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2017. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *ACSAC*.
[38] Jakub Křoustek and Peter Matula. 2017. Retdec: An open-source machine-code decompiler. (2017).
[39] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. 2004. Static Disassembly of Obfuscated Binaries. In *USENIX Sec*.
[40] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler Validation via Equivalence Modulo Inputs. In *PLDI*.
[41] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Finding Deep Compiler Bugs via Guided Stochastic Program Mutation. In *OOPSLA*.
[42] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS*.
[43] Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. 2012. Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators. In *ASPLOS*.
[44] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. Testing System Virtual Machines. In *ISSTA*.
[45] Lorenzo Martignoni, Roberto Paleari, Giampaolo Fresi Roglia, and Danilo Bruschi. 2009. Testing CPU Emulators. In *ISSTA*.
[46] Microsoft. 2018. llvm-mctoll. https://github.com/Microsoft/llvm-mctoll.
[47] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *ICSE*.
[48] Lily Hay Newman. 2019. The NSA makes Ghidra, a powerful cybersecurity tool, open source. https://www.wired.com/story/nsa-ghidra-open-source-tool/.
[49] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *PLDI*.
[50] National Security Agency (NSA). 2018. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/.
[51] Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi. 2010. N-version Disassembly: Differential Testing of x86 Disassemblers. In *ISSTA*.
[52] PNF. 2018. JEB Decompiler. https://www.pnfsoftware.com/.
[53] PNF. 2018. Type Library. https://www.pnfsoftware.com/blog/native-types-and-typelibs-with-jeb/.
[54] Thomas Reps and Gogul Balakrishnan. 2008. Improved Memory-Access Analysis for x86 Executables. In *CC*.
[55] rev.ng Srls. 2018. Rev.ng. https://rev.ng/.
[56] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. 2018. Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables *(CCS '18)*. Association for Computing Machinery, 426–441.
[57] Hao Shi, Abdulla Alwabel, and Jelena Mirkovic. 2014. Cardinal Pill Testing of System Virtual Machines. In *USENIX Sec*.
[58] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *USENIX Sec*.
[59] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Information systems security*. Springer, 1–25.
[60] Chengnian Sun, Vu Le, and Zhendong Su. 2016. Finding Compiler Bugs via Live Code Mutation. In *OOPSLA*.
[61] Dullien Thomas and Sebastian Porst. 2009. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. In *CanSecWest*.
[62] trailofbits. 2018. McSema. https://github.com/trailofbits/mcsema.
[63] Pei Wang, Qinkun Bao, Li Wang, Shuai Wang, Zhaofeng Chen, Tao Wei, and Dinghao Wu. 2018. Software Protection on the Go: A Large-scale Empirical Study on Mobile App Obfuscation. In *ICSE*.
[64] Ruoyu Wang, Yan Shoshitaishvili, Antonio Bianchi, Aravind Machiry, John Grosen, Paul Grosen, Christopher Kruegel, and Giovanni Vigna. 2017. Ramblr: Making Reassembly Great Again. In *NDSS*.
[65] Shuai Wang, Pei Wang, and Dinghao Wu. 2015. Reassembleable Disassembling. In *USENIX Sec*.
[66] Shuai Wang, Pei Wang, and Dinghao Wu. 2016. Uroboros: Instrumenting stripped binaries with static reassembling. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 236–247.
[67] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations.. In *NDSS*.
[68] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and Understanding Bugs in C Compilers. In *PLDI*.