dewolf: Improving Decompilation by leveraging User Surveys

Steffen Enders*, Eva-Maria C. Behner*, Niklas Bergmann*, Mariia Rybalka*, Elmar Padilla*,
Er Xue Hui†, Henry Low†, and Nicholas Sim†
*Fraunhofer FKIE, Germany. {firstname.lastname}@fkie.fraunhofer.de
†DSO National Laboratories, Singapore. {exuehui,lzhenghe,sweishen}@dso.org.sg

***Abstract*—Analyzing third-party software such as malware is a crucial task for security analysts. Although various approaches for automatic analysis exist and are the subject of ongoing research, analysts often have to resort to manual static analysis to get a deep understanding of a given binary sample. Since the source code of provided samples is rarely available, analysts regularly employ decompilers for easier and faster comprehension than analyzing a binary's disassembly.**

In this paper, we introduce our decompilation approach *dewolf*. We describe a variety of improvements over the previous academic state-of-the-art decompiler and some novel algorithms to enhance readability and comprehension, focusing on manual analysis. To evaluate our approach and to obtain a better insight into the analysts' needs, we conducted three user surveys. The results indicate that dewolf is suitable for malware comprehension and that its output quality noticeably exceeds Ghidra and Hex-Rays in certain aspects. Furthermore, our results imply that decompilers aiming at manual analysis should be highly configurable to respect individual user preferences. Additionally, future decompilers should not necessarily follow the unwritten rule to stick to the code-structure dictated by the assembly in order to produce readable output. In fact, the few cases where dewolf already cracks this rule leads to its results considerably exceeding other decompilers. We publish a prototype implementation of dewolf and all survey results [1], [2].

I. Introduction

The number of malware-related incidents is steadily increasing, and with it is the workload of security analysts to analyze provided binary samples [20]. Typically, security analysts heavily rely on manual static analysis to achieve their given objective, including a deep understanding of a binary's capabilities. Because the comprehension of a source code-like representation is typically far less complicated than the binary's disassembly, a *decompiler* may significantly accelerate and ease analysis, rendering it an essential tool for binary analysis in general. However, due to information loss during compilation, a decompiler is incapable of entirely reverting the compilation process and has to make compromises that fit the developers' objective. Therefore, most approaches for decompilation are developed for a specific use-case or with a particular focus, such as aiming at recompilability for further automated processing [9], [41]. Although using a decompiler is an essential part of the static analysis workflow [42], [49], only a few recent academic publications on decompilation focus on *human* analysis [23], [47]. Since the performance of autonomously analyzing binaries is still far behind the abilities of reverse engineering experts [15] and the research about how those experts approach unknown binaries is in its early stages [30], we still deem that manual analysis is one of the most important applications for decompilation. Consequently, we specifically aim at enhancing the output quality in terms of comprehensibility and readability to speed up manual analysis.

In this paper, we propose a novel approach for decompilation called *dewolf*. We base our approach on the previous research state-of-the-art `DREAM++` (see Section III-A) and introduce a wide variety of improvements, such as revising the restructuring by introducing continue statements for control-flow interruptions and by refining the for-loop recovery. Furthermore, we incorporate known algorithms such as the elimination of subexpressions, dead code, loops, and paths. We also introduce readability improvements such as utilizing instruction idiom handling, eliminating redundant casts, and improving constant representations. Finally, we introduce a new approach for Out-of-SSA that potentially reduces the number of variables, and a custom logic engine. We use continuous reliability and correctness testing (see Section III-F) to increase the stability for our *research* decompiler while exceeding the output quality of prototype implementations for other research decompilers such as `DREAM++`. We publish the prototype of dewolf as a Binary Ninja plugin [2] to allow future research in the area of decompilation or related fields.

Apart from this technical contribution, we discuss the results of three user surveys we conducted between the years 2020 and 2021 (see Section IV). Overall, we received 135 complete responses, with each survey having at least 37 participants, and gathered many valuable results and insights. During those surveys, we studied various aspects including user preferences regarding decompiler output, weakness identification of current approaches, and the evaluation of dewolf regarding comprehension. Each survey significantly contributed to the identification of problems that we approached in Section III. Finally, in the third user survey, we included a comparison with other state-of-the-art decompilers to assess the significance of our approach. The results indicate that dewolf's output for the function of the third survey is clearly favored by more than four out of five participants.

One key finding of the user surveys suggests that decompilation should consider bolder approaches to reconstruct the control-flow. Currently, most decompilers try to construct

Workshop on Binary Analysis Research (BAR) 2023
3 March 2023, San Diego, CA, USA
ISBN 1-891562-84-3
https://dx.doi.org/10.14722/bar.2023.23001
www.ndss-symposium.org

the control-flow similar to the structure dictated by the given assembly, e.g., only reconstructing a switch if a jump-table exists. However, a less conservative reconstruction can enhance the human readability of the decompiled code tremendously, even if the results deviate from a structural translation of the disassembly. Indeed, decompilation approaches like revng-c and `DREAM++` already apply certain reconstructions that do not necessarily originate from the considered assembly, such as copying instructions or introducing conditions to avoid goto-statements (gotos). Nonetheless, our survey results suggest that decompilation approaches could go much further to improve readability, leading to a lot of potential for future research while still retaining semantic equivalence.

To summarize, we make the following three contributions:

1) A new decompilation approach, consisting of multiple individual readability improvements, whose output quality considerably exceeds `DREAM++` as well as Ghidra and Hex-Rays in certain aspects.
2) Open-sourcing a prototype implementation of dewolf on GitHub [2] to allow future research and accessible development of new decompilation approaches.
3) Publishing the entire user survey results to allow other researchers insight into our participants' perception of favorable code constructs and output.

II. Related Work

Cifuentes [11] laid the foundation for modern decompilers by proposing a modular decompiler architecture where the actual decompilation algorithms are decoupled from those depending on the input binary architecture or the output high-level language. Most modern decompilers still follow this design principle. Additionally, Cifuentes discussed general decompilation challenges and introduced algorithmic approaches to a subset, like a proposal to recover high-level constructs via *interval analysis*, which allows mimicking the nesting order. Here, a set of predefined patterns for known constructs is used to restructure an interval to a single node. Sharir [37] introduced an approach called *structural analysis* extending the interval analysis by producing a program representation in which structured control-flow patterns are detected and recovered. Afterwards, Engel et al. [18] extended structural analysis to handle C-specific control statements such as *break*, *continue* and *return* to reduce the number of gotos. This extension is called *single entry single successor (SESS) analysis*. Eventually, Schwartz et al. [9] proposed an alternative for structural analysis focusing on soundness called *iterative refinement* which removes control-flow edges that impede existing algorithms and inserts a goto to allow the restructuring to continue. This approach is implemented in the Phoenix decompiler. Emmerik [39] suggested the usage of the static single assignment form (SSA-form) to facilitate data-flow analyses, such as expression propagation, needed for decompilation. While transforming a program into its SSA-form is straightforward, finding an optimal algorithm for an *Out-of-SSA* transformation is not trivial [38].

There are several academic decompilers that each focus on addressing specific challenges. One central research field is the recovery of high-level control-flow constructs such as loops and if-else conditions, also known as control-flow restructuring or recovery. The goal is to produce *structured* code consisting of proper control-flow constructs and containing as few *gotos* as possible to increase output comprehensibility. The Retargetable Decompiler (RetDec) [27] uses capstone for disassembling and LLVM IR as an intermediate language to minimize the overhead of adding support for new architectures. Overall, many other publications regarding RetDec exist addressing different aspects such as idiom handling [28]. Another recent approach is RevEngE [7]. Instead of decompiling the whole sample, they developed a *debug-oriented decompilation* approach. More precisely, during debugging, the analyst can decompile specific code regions making it is possible to decompile regions more than once with different states originating from different execution paths.

Yakdan et al. [47] developed *pattern-independent control-flow restructuring* to recover C control-flow constructs without generating any gotos. In contrast to previous approaches, their algorithm does not rely on predefined patterns for control-flow constructs. Instead, it utilizes conditions based on the reachability of nodes in the control-flow graph (cfg) to infer the corresponding high-level constructs. However, this also requires loops having a single entry and exit point. Because not all real-world samples fulfill this requirement, the authors introduced an algorithm converting *abnormal loops* into their single-entry-exit equivalent. Although such transformations allow a goto-free restructuring, they lead to more complex cfgs and ultimately longer output. The authors implemented a prototype called `DREAM++` as an IDA Pro 6.5 plugin and improved it by adding readability enhancements. They also conducted a user survey to evaluate the approach [44].

The revng-c decompiler [23] is implemented on top of the revng binary analysis framework and introduces a new, yet similar, approach to produce goto-free output. According to the authors, the main obstacle to high-quality human-readable decompilation comes from complex cfgs containing many tangled paths. Thus, they *comb* complicated paths in the cfg by duplicating particular nodes, which are either single basic blocks or already restructured and collapsed regions. The output is indeed goto-free and has a less tangled control-flow but may contain duplicated code. To reduce duplication, they perform *untangling* before combining, which is the duplication of specific paths leading to the exit of the cfg. Presumably, there is no universally valid guideline stating whether duplication should be preferred over a more complex structure as this depends on the given function and personal preference.

In contrast to research approaches, commercial decompilers strongly focus on providing a stable environment and usability features. The Hex-Rays decompiler integrated into IDA Pro can be considered the de-facto industrial standard. Although a few talks discuss some of Hex-Rays decompilation techniques [21], [22], the main decompilation approaches and algorithms remain unpublished. Ghidra, developed by the NSA, is widely considered the state-of-the-art open-source decompiler. Instead of advancing the field of decompilation in general, Ghidra aims to ease decompilation for manual analysis. Moreover, other less commonly mentioned decompilers exist, including FoxDec, JEB, angr, radare2's r2dec, Snowman, and Binary Ninja's Pseudo C.

In the past few years, several decompilation approaches based on (recurrent) neural networks (NN) or neural machine translation have been proposed [25], [26], [29]. Although these approaches are allegedly language-independent, they are typically limited to short code snippets due to network design and memory requirement restrictions. Most authors argue that writing and maintaining conventional decompilers is slow, costly, unscalable, or requires a high level of expert knowledge. However, neural approaches often rely on sophisticated pre- and post-processing to add language domain knowledge to the model [25], [26] or even use traditional control- and data-flow recovery to improve the output [29]. The most critical issue with approaches employing NNs or similar is that models typically learn the appearance of the code rather than capturing semantic equivalence. Although there is a loss of information during compilation, which could justify not relying on semantics, the resulting binary still contains enough information to be translated in a semantic preserving way to the target language by exact and well-researched algorithms. Finally, the utilization of NNs requires a lot of computational power to decompile even very simple code snippets. However, decompilers are particularly essential for complex functions in which the assembly cannot be understood as easily as for simple functions. Unfortunately, real-world programs and especially malware samples are typically rather large which renders current NN-based approaches unsuitable for reverse engineering real-world malware.

III. Approach

In this section, we introduce *dewolf*, our improved approach for decompilation based on the research approach `DREAM++`. First, we will discuss our decompiler and intermediate language selection criteria in Section III-A. Next, in Sections III-B to III-E, we will introduce our approach dewolf and describe its two major improvements over previous approaches as well as some additional methodical details. Finally, we will conclude in Section III-F with a short description about our reliability and correctness testing.

A. Framework Selection Criteria

There are various decompilation approaches that aim at reducing the number of gotos [18], [46] or even completely dispense them [23], [47]. Some of those studies suggest that gotos can aggrevate analysts following the control-flow during static analysis. Besides, there are various approaches that eliminate gotos from source code to improve code readability and its control flow structure [19], [33], [43]. We agree that goto-free code is easier to comprehend and consequently decided to base dewolf on a goto-free approach. To the best of our knowledge, `DREAM++` and revng-c are the only two recent approaches producing goto-free output. While both of those approaches use quite similar techniques for control-flow recovery and restructuring, revng-c uses code duplication additionally to structural variables to avoid gotos. However, even though Gussoni et al. [23] use untangling to reduce the amount of copied code, they still cannot prevent the duplication of large code blocks. We ultimately opted to use `DREAM++` over revng-c as code duplication may be unfavorable for the readability of the output. Besides, `DREAM++` achieved noteworthy results in their conducted surveys while also focusing on human analysis.

Because the implementation of `DREAM++` was not well-maintained and does not work on recent IDA versions, we re-implemented the algorithms introduced in the original publications [44], [45]. As part of the adoption, we base our approach on an intermediate language (IL) of an existing binary analysis framework to avoid re-implementing well-known low-level algorithms such as translating a function to SSA-form. We decided against an integration into an existing decompiler-framework such as Ghidra because this usually impairs their workflow and requires extensive implementation efforts independent of the approach itself.

There are multiple ILs and related lifters that can lift a given binary [16], [22]. For our research, we decided on the following selection criteria: (i) An accessible and comprehensive API to allow fast prototyping, (ii) an included SSA-form for the IL to assist data-flow analysis, (iii) typed variables instead of registers, the elimination of stack-usages, and generally resolving compiler-specifics to further backup the platform-independence of our approach, (iv) function call parameters linked to each function call, and (v) a well-maintained framework. Given these criteria, we decided to use the Medium Level IL (MLIL) from Binary Ninja in SSA-form since it fulfills all of our five requirements. The impact of this choice and Binary Ninja on our results is limited to basic and well-known algorithms as described. A complete description of the MLIL is available in the official documentation [40].

B. Methodology Overview

In the remainder of this section, we only describe differences or improvements we made over the existing `DREAM++` approach. All details not explicitly mentioned in this section are equal or, at least, very similar to the publications related to `DREAM++` [45], [47]. For instance, the control-flow restructuring has been kept identical except for some minor adjustments described below and some improvements for recovering switch constructs. Similarly to the *universal decompiling machine* [11], dewolf is structured into the four phases *frontend*, *preprocessing*, *pipeline stages*, and *backend*. While frontend and backend are congruent with the design by Cifuentes, we utilize a preprocessing phase to normalize the frontend's output, produced by Binary Ninja, in our prototypical implementation. The majority of our improvements over `DREAM++` are implemented as pipeline stages and are platform-independent.

Our approach incorporates different kinds of improvements: First, Sections III-C and III-D introduce two novel algorithms, an approach for Out-of-SSA using graph coloring and a custom logic engine. Next, Section III-E describes improvements for algorithms already integrated into `DREAM++`, such as stability improvements for for-loop recovery, and commonly known or already published algorithms not yet implemented by `DREAM++`, e.g., using *continue* statements in loops [18]. Figure 1 shows a function that we decompiled with our re-implementation of `DREAM++` and a recent version of dewolf which shows the significant impact of our improvements on the readability and output structure, at least for this particular function. The Out-of-SSA algorithm led to the reduced number of variables. Furthermore, we are now able to detect the array-access of the input `arg1` and the for-loop including the renaming of the loop-variable.

```c
void xor(char* s1, size_t slen, char* key, size_t keylen){
  printf("%lu\n", slen);
  for(int i=0; i<slen; ++i){
    s1[i] = s1[i] ^ key[i%keylen]; } }
```

(a) Source code of an example function.

```c
int64_t xor(int64_t arg1, int64_t arg2, int64_t arg3,
↪  int64_t arg4){
  printf("%lu",arg2); int32_t var_c = 0x0L;
  while(true){
    int64_t rax_14 = var_c;
    if ((arg2 <= var_c)){ break; }
    char* rax_4 = (arg1 + var_c); uint64_t rsi_1 = *rax_4;
    int64_t rax_6 = var_c; int64_t rdx_1 = 0x0L;
    char* rax_9 = (arg3 + ((rdx_1:rax_6) % arg4));
    uint64_t rcx = *rax_9;
    int64_t rax_12 = (arg1 + var_c);
    uint64_t rdx_4 = (rsi_1.esi ^ rcx.ecx);
    *rax_12 = rdx_4.dl; int32_t var_c = (var_c + 0x1L); }
  return rax_14; }
```

(b) dewolf's output of initial re-implementation of DREAM++.

```c
long xor(void * arg1, long arg2, long arg3, long arg4) {
  int i, var_0;
  printf(/* format */ "%lu\n", arg2);
  for (i = 0; arg2 > i; i++) {
    arg1[i] = arg1[i] ^ *(arg3 + (i+(0L<<0x40)) % arg4);}
  var_0 = i; return var_0;}
```

(c) dewolf's output which is more concise and readable.

Fig. 1: Comparison between the output of our DREAM++ re-implementation and dewolf.

C. Custom Logic Engine

Logic expressions in binary executables are often strongly optimized to the target architecture and therefore are not always an intuitive translation of the original expressions. Consequently, decompilers often employ a logic engine to simplify such expressions [44] and also identify unreachable code during control-flow restructuring. However, the amount of logic engines featuring full support for logic on bit-vectors is quite limited. While the DREAM++ approach uses SymPy and z3 for condition simplification, it turned out that SymPy is too slow even when simplifying moderately long conditions. The z3 Theorem Prover is an open-source logic engine supporting satisfiability modulo theories (SMT) and bit-vector logic. Considering the vast number of forks on github [32] and citations of their initial publication [14], z3 can be considered the most relevant theorem prover and state-of-the-art. Unfortunately, although it is the state-of-the-art SMT solver, z3 is not optimal for our purpose. Even though it is a full-fledged theorem prover and offers full support for bit-vectors, some design choices of z3 are unsuited to model popular processor architectures such as i386. For example, z3 favors signed operations over unsigned operations. While this seems like a reasonable choice, it may lead to unexpected simplification results when used for decompilation. For example, version 4.8.10 may simplify an unsigned operation into a conjunction of two operations, e.g., with all operands represented by 32-bit vectors:

$$a \leq_u 25 \models a[5:31] = 0 \wedge a[0:4] \leq 25.$$

This simplification does not alter the logic, but it considerably raises the complexity and length of the output. This lack of simplification is particularly relevant since unsigned operations are predominant when handling pointers on any architecture.

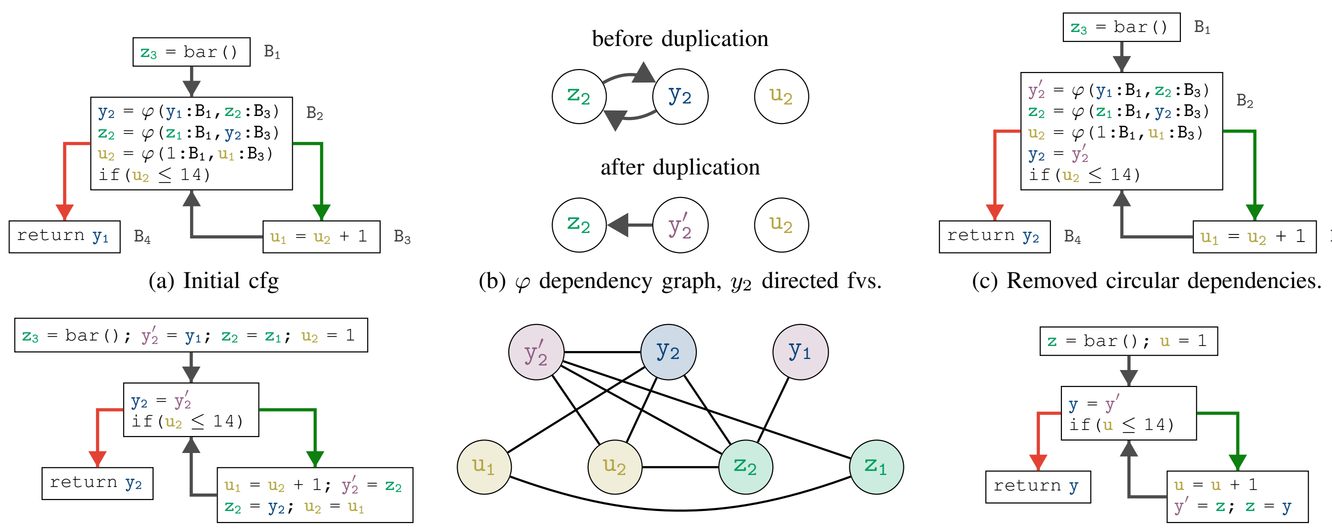
Without a doubt, the complexity of logic statements and branching conditions has a direct impact on the readability of the decompiled code. Because we noticed several simplification problems and had significant obstacles with signed operations, we eventually opted to develop a graph-based logic engine fitting our requirements. Although it only provides a small subset of the functionality offered by z3, we designed it for the particular use case of modeling logic expressions generated from assembly language.

The custom engine utilizes graphs to model logic formulas and defines simplification methods for various common combinations of different operations. We store each formula as part of a graph representing all conditions for a given context which can contain multiple terms and clauses. Each operation and bit-vector (representing a variable or constant) is represented by a node. We represent the relation between the operations and bit-vectors via directed edges, either definition edges linking a variable to its definition or operand edges connecting an operation with its operands. Using our graph representation, we can define a lightweight form of equality based on graph coloring algorithms, also used for graph similarity. Besides, it offers many practical advantages, e.g., we do not need sophisticated methods to deal with the ordering of operands. Finally, we can traverse edges to view relations easily, e.g., between subexpressions. We open-sourced the logic engine alongside dewolf on GitHub [3].

D. Improved Out-of-SSA

Using the SSA-form for analysis during decompilation ultimately requires an Out-of-SSA transformation to obtain valid C output. Unfortunately, removing the φ-functions introduced by the SSA-form is not trivial due to the lost-copy and swap-problem [6], [38]. We developed a graph-based algorithm whose correctness follows directly by design and runs linearly in the program size. In contrast to existing approaches [38], we do not rely on complicated case distinctions and use well-known algorithms. Our approach works similarly to one of Boissinot et al. [6] where the authors insert copies for all variables used in φ-functions, which allows them to give variables of the same φ-function the same name. In contrast, we only duplicate a subset of variables defined by φ-functions to *remove* circular dependencies and to allow a successive execution of the φ-functions. Afterward, it is possible to *lift* the φ-functions to the predecessor blocks. Finally, we *rename* the variables. In the following, we will describe each step in more detail. Figure 2 shows an overview of the approach.

1) Remove circular dependency: To find the variables for which we have to insert copies, we construct a dependency graph as shown in Figure 2b with one vertex for each variable defined by a φ-function and an edge (u, v) between variables u, v if v is used in the φ-function defining u. The edge (u, v) indicates that we have to execute the φ-function defining u before the one for v to use the correct value of v for the computation of u. Hence, in Figure 2b we add the edge (z_2, y_2) due to the φ-function $z_2 = \varphi(z_1, y_2)$. Now, the φ-functions $\varphi_1, \varphi_2, \ldots, \varphi_l$ *depend circularly* on each other if they are all contained in the same basic block and if the variable, defined via φ_i (for $1 \leq i \leq l$), is used in φ_{i+1} (with $\varphi_{l+1} = \varphi_1$). Thus, a cycle in the dependency graph is equivalent to a circular dependency of φ-functions. To resolve



(a) Initial cfg

(b) φ dependency graph, y_2 directed fvs.

(c) Removed circular dependencies.

(d) Lifted φ-functions

(e) Colored interference graph.

(f) Resulting cfg.

Fig. 2: Illustration of each step from our Out-of-SSA algorithm applied to an example cfg.

such circular dependencies, we compute a directed feedback vertex set (directed fvs) in the dependency graph; a directed fvs is a vertex set X whose removal results in an acyclic graph. Although computing a directed fvs is NP-hard [24], we can find an approximation for such a set in constant time since we only have a constant number of vertices in each basic block and since we do this computation for each basic block independently. Now, for every φ-function $x = \varphi(\dots)$ defining a variable x contained in the directed fvs X, we replace the φ-function by $x' = \varphi(\dots)$. Furthermore, we add the instruction $x = x'$ after the last φ-function because variable x must have the value of x' after all, as shown in Figure 2c. Afterward, the dependency graph is acyclic since it does not contain any vertex of the set X. In Figure 2b, we have a cycle, and removing node y_2 would result in an acyclic graph. By replacing y_2 with y_2' in the definition of the φ-function, we obtain an acyclic dependency graph (Figure 2b). Finally, we arrange the φ-function according to a topological order of the dependency graph so that we can execute them in this order.

2) Lift φ-functions: Given resolved circular dependencies and a valid execution order for the φ-functions, we can "lift" the instructions represented by each φ-function to the predecessors of the basic block. More precisely, for each φ-function $x_0 = \varphi(x_1, x_2, \dots, x_l)$, we have a mapping from each predecessor block to the variable or constant assigned to x_0 when entering from this basic block. We add the assignment $x_0 = x_i$ at the end of the basic block BB_i corresponding to x_i when the block does not end with a branch. Otherwise, we add a new basic block between the current block and BB_i and add the assignment in the new basic block. After this, we remove all φ-functions by moving the definitions to the predecessor blocks as shown in Figure 2d. The lifted assignments have the same order as the corresponding φ-functions.

3) Rename Variables: A trivial renaming strategy would be to assign each variable its name in SSA-form. However, this leads to numerous variables and copy assignments. Alternatively, we can group all non-interfering variables and give them the same name. One possible way to achieve this is to use a valid coloring of the interference graph like in Figure 2e and then group all variables of the same color. We create the interference graph by using the liveness-algorithm of Brander et al. [8, Algorithm 4] and adding edges between variables with intersecting live-ranges. Although vertex-coloring with a minimum number of colors is NP-hard [24], we can use a modified lexicographical BFS to find an approximation and the order in which we color the vertices to minimize copies.

E. Other Improvements

In this section, we describe relatively small improvements and utilized adaptions of already published approaches included in dewolf. Despite being seemingly minor modifications, they lead to considerable enhancements in the output quality for human analysts, according to our evaluation.

1) Constant Representation: Constant values are often displayed as hex-numbers in the decompiler output. However, as constant values can represent integers, characters, addresses, or else, a hex-number might not always be the most suitable representation. Consequently, we approximate the most likely representation depending on the constant's type and value to enhance the comprehension of the decompiler output.

2) Elimination of Redundant Casts: Binary programs produced by modern compilers are usually heavily optimized to make efficient use of the available registers. Since many assembly languages allow accessing only parts of a given register, various cast operations can be introduced when lifting the assembly code to a more high-level IL. After propagating expressions, we may end up with many nested casts, some being semantically irrelevant. To declutter the output and considerably increase its readability, we introduce a set of rules for when a cast operation can be omitted due to irrelevance.

3) For-loop Recovery: While DREAM++ already included transformation rules from while-loops into for-loops, the introduced requirements to transform a given loop are actually rather strict. Because these strict rules result in only very few for-loop transformations in real-world samples, we developed a more stable for-loop recovery with relaxed for-loop requirements. For example, in contrast to DREAM++, we allow that the incrementation of the loop counter can be another than the loop body's *last* instruction as long as the defined and used variables remain unchanged before the next loop-iteration.

4) Continue in loops: The DREAM++ approach uses the loop control statement break to indicate that the loop terminates, but not the opposite continue-statement, forcing the next loop iteration. We use both statements to decrease the nesting depth when possible [18]. More specifically, during the restructuring proposed by DREAM++, we do not exclusively add break-nodes on edges used to exit the loop, we also add continue-nodes on every edge where the loop-head is the sink.

5) Switch Variable Detection: To enhance the comprehension of complicated control-flows, programmers frequently use switch statements instead of nested if-statements. However, in assembly generated by modern compilers, switch control-flows are often implemented using indirect jumps and jump-tables. Approaches like Ghidra, angr, and RADARE2 use different methods to obtain the switch-expression from an indirect jump [34]. An indirect jump in Binary Ninja's MLIL consists of two instructions: A calculation of the jump-location, using the jump-table and a jump-offset, followed by `jump(address)`. To find the switch expression, we trace back the expression used to calculate the jump-offsets to its definition and use it to replace the target of the indirect jump.

6) Array Access Detection: In assembly and most ILs, there is no concept for arrays. Instead, an array element access is represented as a dereference operation to a variable plus an offset, i.e., `*(base+offset)`. We find such candidates and mark them as potential array element access. Then, we compute the base and the index depending on the offset and array size. Finally, we discard candidates not fulfilling particular requirements, such as a consistent array size. The decompiler output displays marked operations as `base[i]`.

7) Subexpression Elimination: After propagating expressions, they may still occur multiple times. Unfortunately, depending on the complexity of such expressions, comprehending them multiple times is quite time-consuming for the analyst. During subexpression elimination, we identify already defined expression used in other instructions, as well as expressions that occur multiple times [4], [12]. Using the dominator tree, we find the closest possible position to its usages for insertion to avoid unnecessary interference between variables. Eventually, we insert a definition for the subexpression at the identified location and replace all its occurrences.

8) Instruction Idiom Handling: Unresolved instruction idioms generated by modern compilers can be very challenging for the comprehension of a given function as they typically translate to complicated arithmetic calculations rather than the intended high-level expression. While DREAM++ did not handle instruction idioms, we decided to include a recently published approach [17] using automatically generated assembly patterns to annotate and revert the most common idioms.

9) Elimination of Dead Paths and Loops: After propagating expressions, some branch conditions may be rendered unfulfillable (or the opposite) and therefore introduce impossible paths ultimately leading to unreachable code. Certainly, analyzing dead code is usually not part of the analyst's objective and may significantly decrease efficiency. We identify such conditions and then eliminate *dead* or unreachable paths [4]. Similarly, we can also remove dead loops occurring when the loop condition is always unsatisfied during the first loop entry. Overall, both algorithms remove unreachable basic blocks and can significantly reduce the analysis workload.

```c
int test_case(int argc) {
  short s1 = (argc >= 3) ? argc : -769;
  unsigned short us2 = (unsigned short) s1;
  return us2; }
```

(a) Original source-code, adapted from the LLVM test suite, resulting in `test_case(69794316) == 64012` (2 bytes).

```c
unsigned long test_case(int arg1) {
  short var_0;
  if (arg1 > 2) var_0 = (short)arg1;
  else var_0 = -769;
  return (unsigned int)var_0; } // should be ushort
```

(b) Recompiled output from dewolf decompiler, resulting in `test_case(69794316) == 4294965772` (4 bytes).

Fig. 3: A casting bug found using differential fuzzing. Both values are equal to -1524 in their respective signed variants, but leads to confusion when used as-is by a solver.

F. Reliability and Correctness Testing

While developing the above improvements, we made significant changes to the decompilation process, each typically based on a small number of test functions. However, it is impossible to consider the entire universe of binary functions when developing any new algorithm, resulting in bugs and crashes when decompiling other functions. In addition to handwritten test cases, we used coreutils 8.32 and a subset of the LLVM test suite to find regressions and fix decompilation bugs. We integrated tests for the coreutils dataset into our continuous integration pipeline to proactively identify problematic functions for analyses. Furthermore, we fuzzed dewolf with functions generated by Csmith [48] to find bugs that did not occur in the above set. While this was most useful during early development, it did not help to identify subtler semantic issues.

Finally, we used differential fuzzing to test the semantic correctness of our lifter and transformations. We limited our attempts for re-compilation of decompiled code to the simplest possible functions from the LLVM test suite and generated by Csmith, which do not contain calls to other functions, global variables, structures, or unions. As those constructs oftentimes cause issues during recompilation, their exclusion allowed us to focus on the semantic transformations described above. The functions utilized and generated by Csmith with stated limitations have a median of 61 lines of code. We tested the originally compiled binaries against their decompiled and recompiled counterparts. To perform the tests, we generated harnesses for use with the Nezha framework [35]. The differential fuzzing tests helped us to fix integer promotion casting bugs as well as type recovery bugs which we would not have identified otherwise. Figure 3 shows an example of an identified bug. Our testing experience showed that fuzzing and differential testing are readily applicable to decompilers. While we were only

#	Execution	Responses Total	Responses Full	Time (in minutes) 25%	Time (in minutes) 50%	Time (in minutes) 75%
1	2020-10	84	37	41	75	126
2	2021-04	98	44	17	35	57
3	2021-10	85	54	24	38	57

TABLE I: Metadata of the conducted surveys including the average resp. median time per full response.

able to re-compile and differentially fuzz a small subset of randomly-generated functions, it has increased our confidence in the accuracy and correctness of dewolf.

IV. Evaluation

We conducted three user surveys (see Table I) to evaluate the following three aspects: First, identifying the limitations of DREAM++, finding constructs preferred by analysts, as well as relevant research questions. Second, we evaluated whether the improved output of dewolf is sufficient to comprehend a realistic malware sample. Finally, we compared dewolf to the open-source and commercial state-of-the-art decompilers Ghidra and Hex-Rays. In general, each conducted user survey included a self-assessment and feedback for the survey itself, which helped us to improve them in each iteration.

1) Survey Participants: Gathering participants for user surveys on a highly specific topic like decompilation or reverse engineering is not particularly easy. Of course, the first choice for participants are professional reverse engineers or people who are professionally active in malware analysis. Unfortunately, there are neither plenty professionals in these areas nor are they available to take part in time-consuming studies resulting in fewer participants compared to less-technical surveys, being confirmed by surveys with related topics all having similar participation numbers [30], [45], [49].

However, it is arguable that people with a solid understanding of the C programming language can also participate in our survey. First, most decompiler outputs are syntactically very similar to C code, i.e., comprehensible by programmers. Second, comprehension and preference for the format of C-like source code does not require reversing skills. Finally, the ultimate goal is to generate output also non-experts can comprehend because learning reversing and assembly is very difficult and time-consuming [30]. Consequently, we did not limit our survey to participants with experience in reverse engineering and were pleasantly surprised by the extensive participation. Figure 4 shows that most participants have experience with C according to their self-assessment (see Section IV-D). In the second and third survey roughly 60% of the participants stated they had C-experience. As expected, the number of participants with reversing skills is comparatively low. In the final survey, we gathered more than 40% of participants with allegedly substantial reversing skills . Our participants mainly consisted of program or malware analysis students, colleagues with malware analysis experience, and professional reversers.

2) Survey Platform: We opted to use an online platform to conduct our surveys to not be limited to geographically close participants, to avoid time constraints, and to ensure anonymity. At the beginning of each survey, we sent a link to a privately hosted instance of LimeSurvey to potential participants.

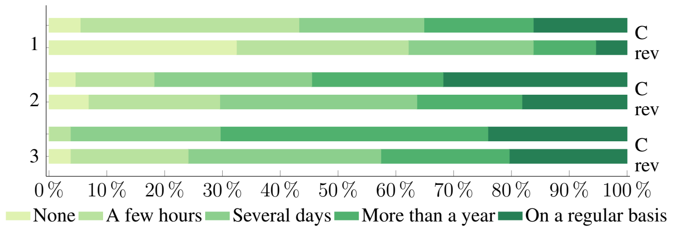


Fig. 4: The participants' reversing and C experience across all user surveys. For the first survey (ranking from 1 to 10), we merged 1+2 to *None*, 3+4 to *A few hours*, and so on.

```c
int A(int m, int n) {
    if (!m) return n + 1;
    if (!n) return A(m - 1, 1);
    return A(m-1,A(m,n-1)); }
```

Fig. 5: Source code snippet for the first user survey. The function `A(...)` implements the Ackermann function.

A. Survey 1: Limitations of Dream

One main goal of the first user survey was to evaluate how our re-implementation of DREAM++ performs against the commercial and open-source state-of-the-art. We decided to use the Ackermann function (Figure 5) compiled with GCC 10.0.1 since it is a non-trivial arithmetic function featuring non-primitive recursion. We decompiled the function with dewolf (DREAM++ re-implementation without the improvements from Section III), Ghidra 9.1.2., and Hex-Rays 7.5. SP2, and randomly assigned each participant one output. To avoid participants recognizing particular decompilers, we normalized all decompiler outputs. For example, we introduced a common naming scheme for variables and function names. Overall, we collected data about the comprehension and subjective trends regarding the presentation of the three different outputs.

To verify the participants' understanding of the given function and to validate the self-assessment, we asked which values the function returns for two different sets of parameters and which parameter combinations result in the most recursive calls. Figure 6 shows the comprehension results divided by the decompiler that generated the output for the respective participant. Additionally, we asked questions about the output quality. Since we identified no grave differences between dewolf and the other decompilers in comprehension and quality, we felt confident that dewolf produces competitive results.

In the final part of the survey, we presented participants with small side-by-side comparisons to gain insights into user preferences. While there are many code constructs where participants have a clear consensus, such as to prevent high nesting depths, there are many cases where seemingly no clear preference exists, such as the usage of gotos. The topics we asked participants include preferences regarding program structure like switch versus if-else, value and expression propagation, and the use of gotos. It turned out that many participants were particularly dissatisfied with seemingly small details that may still require elaborated approaches, e.g., the representation of constants or (for-)loop restructuring. Moreover, the fact that modern decompilers do not handle all instruction idioms turned

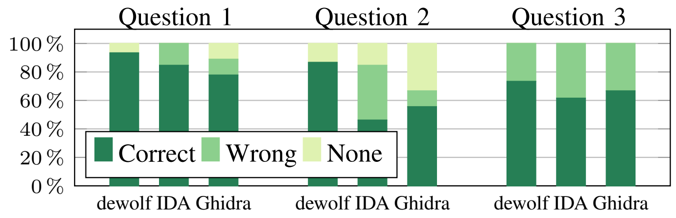


Fig. 6: Comprehension results of the Ackermann function. The questions are included in Appendix 2.

```c
char* dga() {
  unsigned char* domain; char seed;
  domain = malloc(sizeof(SYSTEMTIME));
  seed = (char)GetTickCount();
  GetSystemTime((LPSYSTEMTIME)domain);
  for(char i = 0; i < DOMAIN_LENGTH; i++) {
    domain[i]=((unsigned char)(domain[i]^seed)%24)+97;}
  int* end = domain + DOMAIN_LENGTH;
  end[0] = "\x2e\x63\x6f\x6d"; end[1] = 0;
  switch(seed % 8){
    case 7: case 5: end[0] ^= "\x00\x11\x1a\x6d"; break;
    case 1: case 6: end[0] ^= "\x00\x17\x00\x6d"; break;
    case 2: end[0] ^= "\x00\x0d\x0a\x19"; break;
    default: break; }
  return domain; }
```

Fig. 7: Artificial DGA of survey 2 which produces domains with 8 random chars except z with four different TLDs.

out to be a substantial issue. Consequently, we decided to tackle issues criticized by participants regarding our output. Additionally, we approached two more complex and particularly substantial issues namely the complexity of expressions and conditions and the number of variables. To address these problems, we developed a custom logic engine and focused on Out-of-SSA as well as on handling variable copies.

Another aspect frequently mentioned by participants was the lack of configurability. Indeed, users may have contrasting preferences and can benefit from diverse configuration options. Besides, preferences often diverge depending on the given sample or function. Hence, it is no surprise that most participants are disappointed with the extent to which some algorithms of state-of-the-art decompilers are configurable. More specifically, they would appreciate tweaking decompiler settings to fit their individual preferences or the characteristics of a given function. Consequently, many algorithms in dewolf are highly configurable by the user (see Appendix 1).

B. *Survey 2: Comprehension*

The purpose of the second survey was to evaluate the comprehensiveness of dewolf's output (including the improvements from Section III). More precisely, we validated whether participants can use dewolf to comprehend a realistic malware function to assess its applicability. We decided that an adequate malware component to be analyzed in our survey would be a domain generation algorithm (DGA). DGAs implement a kind of domain flux designed to improve the resilience of network communication utilized by modern malware to avoid a single-point-of-failure during network communication by not using hardcoded addresses for the network infrastructure of the malware [36]. Thus, during malware analysis, analysts are often interested in the *detailed* implementation of DGAs to obtain domains of interest, in contrast to, e.g., crypto algorithms where no complete understanding may be required once identified. We opted to use a synthetic DGA since real-world malware samples are exponentially increasing in size and complexity [10]. Besides, DGAs are regularly implemented across multiple functions and may have dependencies that are hard to represent properly in a survey. To make the DGA as realistic as possible, we studied numerous real-world DGAs using DGArchive [36] and programmed a very similar yet slightly less complex DGA in a single function (see Figure 7) generating realistic domains as a real-world DGA would.

Each participant received dewolf's output of the DGA from Figure 14, compiled with VS2015. We then asked about the functions purpose to assess whether the participants determine its functionality, i.e., that it is a DGA. In total, more

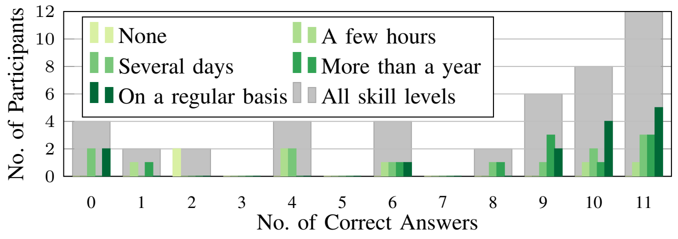


Fig. 8: DGA Comprehension results of the second survey for all participants and split by C-skill level.

than 60% were able to identify the purpose of the function correctly. Noticeably, the majority of participants that gave an incorrect answer had either no C- respectively reversing skills or rushed through the survey in less than 15 minutes. Since we continued with more in-detail questions involving the structure and form of the generated domains, we explained the function's purpose after the first question. More specifically, we asked about the domain length, character set, and top-level domains, allowing us to establish which parts of the algorithm were understandable. Most participants identified the used top level domains (TLDs) and correctly classified *.com* as the most used. Finally, we showed the participants five domains and asked whether the given algorithm could potentially generate them. The comprehension results are summarized in Figure 8 whereas all questions are contained in Appendix 3. Overall, slightly more than 70% of all participants answered at least half of the questions correctly. Since most of the questions were not trivial, we are delighted by these results. Further, Figure 8 shows that C skills seem to have an impact on the answer score. This impact is not surprising and strengthens the result even more because most incorrect answers were seemingly not caused by dewolf's output but by the lack of expertise.

Finally, the participants compared dewolf's output of the DGA with a few other decompilers. The goal was to collect valuable feedback, user opinions about the different decompiler outputs, and possible improvements. The results indicate that, for the given sample, Hex-Rays, Ghidra, and dewolf produce an eminently better output than the others. Consequently, we only consider these three decompilers in the final survey.

```
int convert_binary_to_hex() {
  long long binary; char hex[65] = ""; int remainder;
  printf("Enter any binary number: ");
  scanf("%lld", &binary);
  while(binary > 0) {
    remainder = binary % 10000;
    switch(remainder) {
      case 0: strcat(hex, "0"); break;
      case 1: strcat(hex, "1"); break; /* ... */

      case 1111: strcat(hex, "F"); break; }
    binary /= 10000; }
  printf("Binary number: %lld\n", binary);
  printf("Hexadecimal number: %s", hex);
  return 0; }
```

Fig. 9: Code Snippet for the third user survey to generate the hex representation of a given binary number.

C. Survey 3: Comparison

The primary goal of the final user survey was to compare dewolf to other decompilers. Due to the instability of the DREAM++ implementation, we only compared dewolf to Ghidra 10.0.3. and Hex-Rays 7.61. SP1. Besides, we already compared our initial re-implementation of DREAM++ to both Hex-Rays and Ghidra in the first user survey. We opted against comparing our approach with more approaches due to the survey limitations discussed in Section IV-D. Additionally, when conducting the third survey, neither Pseudo-C from Binary Ninja nor revng-c were publicly released.

For the comparison, we constructed a semantically meaningful function (see Figure 9) that includes various challenges for decompilers while retaining a manageable size. These aspects include the utilization of different datatypes including an array, various constant integer values and strings, nested control-flow structures, i.e. a loop containing a switch with non-consecutive cases, and an instruction idiom. We compiled the function with GCC 10.3.1 and decompiled the resulting sample with Hex-Rays, Ghidra, and dewolf (Figure 15). In contrast to the first survey, each participant was presented with all three samples in random order and marked with pseudonyms. Then, each participant ranked them from most to least favorable (Figure 10). The results indicate that dewolf was the most favored for the considered function, with Hex-Rays still being significantly ahead of Ghidra. This clearly underlines the positive impact of all introduced improvements, although each individually being relatively simple, at least on the participants' perception. Aspects of dewolf's output that were frequently praised by participants include the reconstructed switch compared to a higher nesting depth produced by the other decompilers, and the lack of unnecessary casts. In contrast, participants disliked that dewolf was incapable of correctly detecting the array, whereas Hex-Rays was. Although dewolf was ranked first in this sample, we do not claim that dewolf is superior in general. Still, dewolf is capable of applying restructuring techniques not supported by the other decompilers and exceeds their output in certain properties.

In the second part of this survey, we asked the participants some questions regarding their preferences for the decompiler output to verify the generally taken assumption that reverse engineers want highly accurate decompilation output, i.e., an output similar to the assembly structure. Unsurprisingly, most

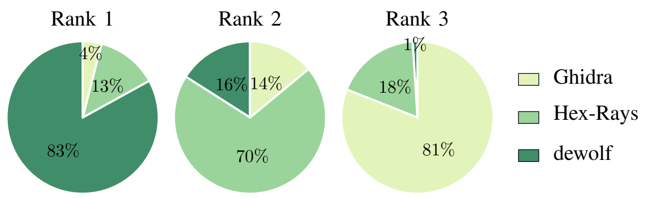


Fig. 10: Ranking of the decompilers for the function in Figure 9. A total of 53 participants ranked dewolf best.

decompilation approaches and commercial decompilers aim to achieve this goal [9], [41], although some approaches like DREAM++ already slightly violate it. We decided to scratch the surface of this unwritten rule by including some questions to verify whether participants accept substantial differences to the assembly structure in favor of increasing the readability. For instance, they had to choose between three semantically equivalent reconstruction options. The function was restructured by either having a more complicated structure, containing various levels of duplication, or an alternative version by extracting a section into a new artificial function. In total, we considered three different scenarios to obtain an overall impression, such as shown in Figure 16. As expected, some participants complained that one should not alter the structure of the considered program imposed by the disassembly. However, depending on the scenario, the majority preferred the alternative version even though being clearly different from the assembly structure. Furthermore, some participants specifically praised the better readability translating to less required analysis time and possibly better analysis results.

D. Discussion

Typically, code quality is evaluated using quantitative measurements, i.e., code metrics. However, a significant downside of using such metrics is that each metric favors a particular behavior [5] and may not necessarily lead to a better readability during manual static analysis. For example, the revng-c approach [23] uses the McCabe Cyclomatic Complexity [31] which favors their way of duplication without measuring its disadvantages: When copying a node with ≥ 2 predecessors and one successor such that each duplication has one predecessor, the code size may considerably increase whereas the cyclomatic complexity does not change. When duplicating arbitrarily big basic blocks without any successor, the cyclomatic complexity may even decrease, although such duplicating obviously harms the code's readability and decreases its quality. Even combinations of metrics such as McCabe Cyclomatic Complexity *and* Lines of Code are insufficient to assess the readability as reducing the lines of code does not always help a human analyst. Consequently, we decided to exclusively use user surveys to evaluate the improvements of our approach and to study analysts' preferences regarding code constructs or output formats. The first user study even confirmed that it is nearly impossible to define formal criteria for code quality because readability is purely subjective, and no sound default configuration for decompiler outputs can please all analysts.

Although the individual and incremental evaluation of all improvements from Section III would allow assessing the individual impact of each improvement, this would result in an

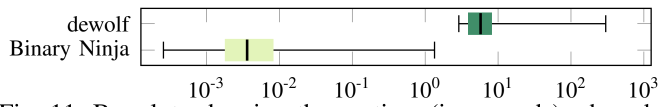


Fig. 11: Boxplots showing the runtime (in seconds) when decompiling coreutils samples with dewolf and Binary Ninja.

unmanageable number of combinations that goes far beyond what we can evaluate. Additionally, we presume that the impact of each individual contribution is rather small and may not result in measurable improvements. Hence, we opted for an evaluation of our decompiler approach containing all introduced improvements. We further decided against using more complex real-world malware samples or comparing against more than two other approaches to address the very limited number of participants, i.e., people, especially professionals, willing and able to participate in a user survey related to reversing [30], [47]. Recent research also indicates that even popular paid services, claiming to provide qualified survey participants, should not be used to overcome this limitation [13].

Finally, we decided to only include a self-assessment in the second and third survey since we did not observe any significant divergences between the self-assessment and an assessment based on questions about the Ackermann Function during the first survey. This decision saved valuable participants' time and allowed us to include additional content. Consequently, results split by the participant's skills are solely built on the provided self-assessment and should be tempered with caution. Regardless, our results indicate that reversing skills are far less relevant compared to the capability of understanding code which is very helpful for comprehension.

E. Runtime Evaluation

We heavily focus on readability improvements and consequently neither optimized our approach nor prototype implementation with respect to speed. While we want to leave a precise runtime evaluation of the presented approaches' subject for future work, we understand that a *reasonable* runtime is an essential aspect of decompilation approaches. To demonstrate that our Python prototype is sufficiently fast on real-world binaries, even without speed optimizations or being implemented in native code, we decompiled all binaries in coreutils on an i7-10850H with 12GBs of RAM and measured the decompilation times. With a timeout of 5 minutes, the dewolf prototype can decompile 94% of all binaries and 75% of them in less than 8 seconds. Figure 11 shows the measured decompilation times for dewolf and Binary Ninja. As presumed, the decompilation times of Binary Ninja are significantly faster. Because all of its analyses run at the start of dewolf to obtain the MLIL, they can be considered a lower bound for dewolf's runtime.

V. Conclusion and Outlook

In this paper, we made three main contributions: First, we conducted and discussed three user surveys providing valuable information for research in the area of reverse engineering and decompilation. We publish the survey results allowing other researchers to use them in three ways: (i) as a guideline of user preferences for decompiler output or code in general, (ii) insights into the limitations of current approaches from a user perspective, and (iii) as starting points for new research. A key finding of our surveys is that a majority of participants do not necessarily favor decompilers producing output as close as possible to the structure implied by the assembly as long as it improves the readability. Instead, to help human analysts, it can be highly beneficial to allow a more flexible output generation while maintaining semantic equivalence. Even substantial structural differences to the disassembly should be considered in favor of significant readability improvements. Since existing approaches rarely oppose the disassembly and avoid substantial changes to the program structure, we have to overthink and actively re-consider them. Ultimately, our results establish multiple possibilities for decompilation and can have a broad impact on future research, even opening entirely new options for output formats.

The results of our surveys also imply that many participants deeply desire configurability. Furthermore, the evaluation indicates that small details, such as the representation of constants, can already have a positive impact in some cases. Because readability seems to be highly subjective, analysts can benefit from individual high-level representations influenced by decompiler parameters. Besides, as the characteristics of functions vastly differ between samples, configuration options could be very useful to tune the decompiler for each given use-case. Consequently, future decompilation approaches should allow user configuration for such details when possible. This also raises the question whether the existing metrics for decompiler evaluation even apply to human analysis considering the wide variety of user preferences. Finally, we expect that the survey results can be helpful to other researchers in the field of decompilation, reverse engineering, or even topics dealing with comprehending source code. In particular, it is possible to use some of the results as a baseline for further research. We publish the complete questionnaires and participants' responses to allow further analyses and conclusions from others [1].

Based on the survey results, we introduced a new decompilation approach named dewolf with significant improvements over the previous research state-of-the-art. In addition to countless small-scale improvements, we developed a custom logic engine and a novel, well-described Out-of-SSA algorithm. The evaluation indicates that dewolf is not only a considerable improvement over the previous state-of-the-art but is also suitable to comprehend and analyze malware. Moreover, the output quality of dewolf noticeably exceeds the commercial and open-source state-of-the-art decompilers for several samples, according to survey participants. Lastly, we open-source our decompiler prototype dewolf to provide other researchers with a suitable base for their work. Due to its stability and its modular design, dewolf is exceptionally well suited to integrate and evaluate new approaches. Further, we hope that dewolf motivates and assists more research focusing on manual analysis to cope with the increasing security analysis load.

Acknowledgements

We are sincerely grateful to all participants that took part in our user surveys and provided us with valuable feedback. We also thank Daniel Plohmann for reading various draft versions of this paper and suggesting numerous improvements. Finally, we acknowledge Felix Prahl-Kamps and Marvin Marks for their extensive help during the prototype implementation.

References

[1] "Surveys," https://github.com/steffenenders/dewolf-surveys, 2020-2021.

[2] "dewolf," https://github.com/fkie-cad/dewolf, 2022.

[3] "dewolf-logic," https://github.com/fkie-cad/dewolf-logic, 2022.

[4] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: principles, techniques, and tools*. Addison-wesley Reading, 2007, vol. 2.

[5] H. R. Bhatti, "Automatic measurement of source code complexity," 2011.

[6] B. Boissinot, A. Darte, F. Rastello, B. D. De Dinechin, and C. Guillon, "Revisiting out-of-ssa translation for correctness, code quality and efficiency," in *2009 International Symposium on Code Generation and Optimization*. IEEE, 2009.

[7] M. Botacin, L. Galante, P. de Geus, and A. Grégio, "Revenge is a dish served cold: Debug-oriented malware decompilation and reassembly," in *Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium*, 2019.

[8] F. Brandner, B. Boissinot, A. Darte, B. D. De Dinechin, and F. Rastello, "Computing liveness sets for ssa-form programs," Ph.D. dissertation, INRIA, 2011.

[9] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *22nd USENIX Security Symposium (USENIX Security 13)*, 2013.

[10] A. Calleja, J. Tapiador, and J. Caballero, "A look into 30 years of malware development from a software metrics perspective," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016.

[11] C. Cifuentes, "Reverse compilation techniques," Ph.D. dissertation, QUEENSLAND UNIVERSITY OF TECHNOLOGY, 1994.

[12] J. Cocke, "Global common subexpression elimination," in *Proceedings of a symposium on Compiler optimization*, 1970.

[13] A. Danilova, A. Naiakshina, S. Horstmann, and M. Smith, "Do you really code? designing and evaluating screening questions for online surveys with programmers," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021.

[14] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[15] Defcon, "Defcon ctf scores," https://www.defcon.org/html/defcon-24/dc-24-ctf.html, 2022.

[16] T. Dullien and S. Porst, "Reil: A platform-independent intermediate representation of disassembled code for static code analysis," 2009.

[17] S. Enders, M. Rybalka, and E. Padilla, "Pidarci: Using assembly instruction patterns to identify, annotate, and revert compiler idioms," in *2021 18th International Conference on Privacy, Security and Trust (PST)*. IEEE, 2021.

[18] F. Engel, R. Leupers, G. Ascheid, M. Ferger, and M. Beemster, "Enhanced structural analysis for c code reconstruction from ir code," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems*, 2011.

[19] A. M. Erosa and L. J. Hendren, "Taming control flow: A structured approach to eliminating goto statements," in *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)*. IEEE, 1994.

[20] A.-T. GmbH. (2022) Malware statistics & trends report. [Online]. Available: https://www.av-test.org/en/statistics/malware/

[21] I. Guilfanov, "Decompilers and beyond," *Black Hat USA*, 2008.

[22] ——, "Decompiler internals: Microcode," https://i.blackhat.com/us-18/Thu-August-9/us-18-Guilfanov-Decompiler-Internals-Microcode-wp.pdf, Tech. Rep., 2018.

[23] A. Gussoni, A. Di Federico, P. Fezzardi, and G. Agosta, "A comb for decompiled c code," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[24] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972.

[25] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018.

[26] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, "Towards neural decompilation," *arXiv preprint arXiv:1905.08325*, 2019.

[27] J. Křoustek, "Retargetable analysis of machine code," Ph.D. dissertation, Faculty of Information Technology, Brno University of Technology, CZ, 2015.

[28] J. Křoustek and F. Pokorný, "Reconstruction of instruction idioms in a retargetable decompiler," in *4th Workshop on Advances in Programming Languages (WAPL'13)*. Kraków, PL: IEEE, 2013.

[29] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity*, vol. 4, no. 1, 2021.

[30] A. Mantovani, S. Aonzo, Y. Fratantonio, C. Talos, and D. Balzarotti, "RE-Mind: a first look inside the mind of a reverse engineer," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022.

[31] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering*, no. 4, 1976.

[32] Microsoft Corporation, "z3," https://github.com/Z3Prover/z3, 2022.

[33] S. Pan and R. G. Dromey, "A formal basis for removing goto statements," *The Computer Journal*, vol. 39, no. 3, 1996.

[34] C. Pang, R. Yu, Y. Chen, E. Koskinen, G. Portokalidis, B. Mao, and J. Xu, "Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021.

[35] T. Petsios, A. Tang, S. Stolfo, A. D. Keromytis, and S. Jana, "Nezha: Efficient domain-independent differential testing," in *2017 IEEE Symposium on Security and Privacy (Oakland)*. IEEE, 2017.

[36] D. Plohmann, K. Yakdan, M. Klatt, J. Bader, and E. Gerhards-Padilla, "A comprehensive measurement study of domain generating malware," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[37] M. Sharir, "Structural analysis: A new approach to flow analysis in optimizing compilers," *Computer Languages*, vol. 5, no. 3-4, 1980.

[38] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam, "Translating out of static single assignment form," in *International Static Analysis Symposium*. Springer, 1999.

[39] M. J. Van Emmerik, "Static single assignment for decompilation," Ph.D. dissertation, 2007.

[40] Vector 35, "Binary Ninja mlil," https://docs.binary.ninja/dev/bnil-mlil.html, 2022.

[41] F. Verbeek, P. Olivier, and B. Ravindran, "Sound c code decompilation for a subset of x86-64 binaries," in *International Conference on Software Engineering and Formal Methods*. Springer, 2020.

[42] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[43] M. H. Williams and G. Chen, "Restructuring pascal programs containing goto statements," *The Computer Journal*, vol. 28, no. 2, 1985.

[44] K. Yakdan, "A human-centric approach for binary code decompilation," Ph.D. dissertation, University of Bonn, Germany, 2018.

[45] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, "Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.

[46] K. Yakdan, S. Eschweiler, and E. Gerhards-Padilla, "Recompile: A decompilation framework for static analysis of binaries," in *2013 8th International Conference on Malicious and Unwanted Software:" The Americas"(MALWARE)*. IEEE, 2013.

[47] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations." in *NDSS*. Citeseer, 2015.

[48] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 283–294.

[49] M. Yong Wong, M. Landen, M. Antonakakis, D. M. Blough, E. M. Redmiles, and M. Ahamad, "An inside look into the practice of malware analysis," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

Appendix

1) Configurability: One particular strength of dewolf is its extensive configurability. The survey results indicate that decompiler configurability can significantly improve analysis, according to the participants. In dewolf, most stages and algorithms have extensive configuration options. Additionally, the user can configure which pipeline stages should run and in which order, with the only restriction that the Out-of-SSA stage has to be run after all data-flow algorithms and before the control-flow algorithms. The order of the data-flow and control-flow algorithms can be chosen arbitrarily. It is also possible to run stages multiple times. In the following, we will describe the configuration options for three exemplary stages.

The stage *instruction length handler* ensures that individual instructions are not too long. To address different user preferences regarding maximum length, we allow the user to set the maximum number of operations per instruction. Our algorithm for *common subexpression elimination* finds redundant expressions and introduces new variables for such. It offers multiple options such as a lower bound for the occurrences of an expression to be extracted. In the *code generator*, users can, for example, decide whether compound operations (+ =) or integer incrementation suffices (i++) should be used. Additionally, the user can configure how to represent constants and the aggressiveness of the array detection, i.e., how often to represent `*(a+i*4)` as `a[i]`.

2) User Survey I: Figure 13 shows the decompiler outputs we generated with dewolf, at that time essentially being a re-implementation of `DREAM++`, Hex-Rays, and Ghidra. To reduce the space consumption of the outputs, we deleted some linebreaks and inline selected conditions. We showed each participant one of those random decompiler outputs. The participants then had to answer questions about its comprehensiveness and their judgment of the output quality on a Likert scale. Figure 12 shows the overall results for each question for all three considered decompilers indicating only small divergences in most categories. Four questions expose divergence between them and their control questions. Interestingly, they contain all questions of two categories: variables and restructuring. Apparently, participants do not have a general preference regarding the number of variables, their typing, and restructuring.

3) User Survey II: To verify the comprehensibility of dewolf's output, we decompiled the DGA of Figure 7 with dewolf. The output is shown in Figure 14. On the positive side, dewolf correctly restructures the switch statement and the array-access for the domains (`var_0`). Additionally, dewolf is capable of reversing the instruction idioms in lines 8 & 13, e.g., the modulo 24 computation. Nevertheless, there is still room for improvement. For example, dewolf does not detect the array access for the TLDs (`var_4`). Moreover, dewolf is very conservative with propagation pointers leading to unnecessary copies of variables, e.g., in line 7. But, we already attacked this problem. The current version of dewolf propagates the instruction in line 7 into line 8. We asked each participant the following comprehension questions:

- What does this function return?
  - ○ No Idea
  - ○ Manipulates the system time
  - ○ Allows Virtual Machine Detection
  - ○ Generates random domains
  - ○ Encrypts memory regions
- Please type all utilized top-level-domains (TLDs).
- Which Top-Level domain will be utilized the most?
- Which of these second-level domains can potentially be generated by the function? (Yes/No answer option)
  - ○ simpmpfp
  - ○ xfbcbcic
  - ○ facebook
  - ○ squzuzfz
  - ○ rlpmpmgmjdh

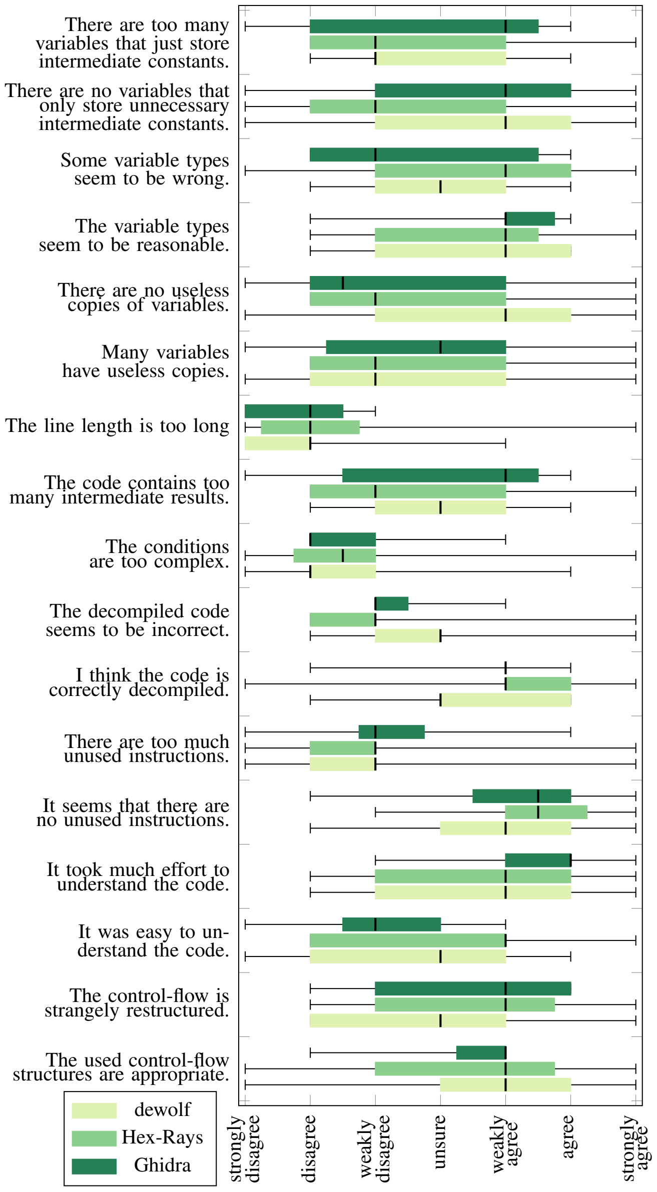


Fig. 12: Boxplots visualizing the Likert scale values evaluated in the first user survey for the outputs from Figure 13.

The correct answer to the first question is *Generates random domains*. To identify all utilized top-level domains, one has to look at the strings assigned to the array `end` resp. the pointer `var_4`. This leads to the following four TLDs:

```
unsigned long A(int arg1, int arg2) {
  unsigned long var_0;
  if (arg1 == 0) { var_0 = arg2; }
  else {
    var_0 = arg2;
    while(true) {
      arg2 = (arg1 + -0x1) &
↪   0xffffffff;
      if ((unsigned int)var_0 == 0) {
        if (arg2 == 0) { var_0 = 0x1;
↪   break; }
        arg1 = arg2; var_0 = 0x1;
↪   continue; }
      var_0=A(arg1,((unsigned
↪   int)var_0)+0xffffffff);
      if (arg2 == 0) { break; }
      arg1 = arg2; } }
  return ((unsigned int) var_0) + 0x1;
↪   }
```

(a) Output:re-implementation of DREAM++.

```
__int64 __fastcall A(__int64 arg_1,
↪   int arg_2){
  unsigned int var_0;
  if ( (_DWORD)arg_1 ) {
    do{
      while ( 1 ) {
        var_0 = arg_1 - 1;
        if ( !arg_2 ) { break; }
        arg_2 = A(arg_1, (unsigned
↪   int)(arg_2 - 1));
        arg_1 = var_0;
        if (!var_0) {return (unsigned
↪   int)(arg_2+1);}}
      arg_2 = 1; arg_1 = var_0;
    }while ( var_0 ); }
  return (unsigned int)(arg_2 + 1); }
```

(b) Output: Hex-Rays, IDA 7.5.200728 (SP2).

```
ulong A(ulong arg_1,ulong arg_2){
  int var_0; uint var_1; arg_2 = arg_2
↪   & 0xffffffff;
  var_0 = (int)arg_2; var_1 =
↪   (uint)arg_1;
  while (var_1 != 0) {
    while( true ) {
      var_1 = (int)arg_1 - 1;
      if ((int)arg_2 != 0) { break; }
      arg_2 = 1; var_0 = 1; arg_1 =
↪   (ulong)var_1;
      if (var_1 == 0) { goto Label_1;
↪   } }
    arg_2 = A(arg_1,(ulong)((int)arg_2
↪   - 1));
    var_0 = (int)arg_2; arg_1 =
↪   (ulong)var_1; }
  Label_1: return (ulong)(var_0 + 1);
↪   }
```

(c) Output: Ghidra 9.1.2.

Fig. 13: Decompiler outputs for the function from Figure 5 used for the first user survey.

```
int sub_401000() {
  char i; char * var_0; char var_1; char var_3; int var_4;
  var_0 = malloc(16);
  var_1 = GetTickCount();
  GetSystemTime(lpSystemTime: var_0);
  for (i = 0; i < 8; i++) {
    var_3 = var_0[i];
    var_0[i] = ((unsigned int)(var_3^var_1)%24 &0xff)+'a';
  }
  var_4 = var_0 + 8;
  *var_4 = 0x6d6f632e;
  *(var_4 + 4) = 0x0;
  switch((((int) var_1) % 0x8) - 0x1) {
  case 0:
  case 5:
    *var_4 = *var_4 ^ 0x6d001700;
    break;
  case 1:
    *var_4 = *var_4 ^ 0x190a0d00;
    break;
  case 4:
  case 6:
    *var_4 = *var_4 ^ 0x6d1a1100;
    break;
  }
  return var_0;
}
```

Fig. 14: The output of dewolf given the DGA of Figure 7.

- `.com` (`"\x2e\x63\x6f\x6d"`)
- `.ru` (`"\x00\x11\x1a\x6d"`)
- `.tu` (`"\x00\x17\x00\x6d"`)
- `.net` (`"\x00\x0d\x0a\x19"`).

The most utilized TLD is `.com` because we have eight different cases and `.com` is used for three, whereas the others are only chosen for up to two cases. Finally, the only second-level domains not potentially generated by the domain generation algorithm are *squzuzfz* and *rlpmpmgmjdh* because the character z is excluded in the loop and the length of each domain is eight.

4) User Survey III: During the first part of the third survey, each participant was given and asked to rank all three decompiler outputs from Figure 15. The ranking results are illustrated in Figure 10 and shortly discussed in Section IV. Figure 15c shows that Hex-Rays restructured the loop-body very closely to the structure of the assembly code by utilizing an if/else structure. Similarly, Ghidra also produced an if/else structure, but with a significantly higher nesting depth by utilizing cascading if statements (Figure 15a). In contrast to both, dewolf restructured this part into a switch statement by identifying a candidate variable for a switch construct (Figure 15b). Although this does not strictly reflect the assembly due to the missing jump table, we argue it is easier and faster to comprehend. Further differences between the outputs mentioned by the participants include but are not limited to:

- The loop-type used for restructuring, i.e., Ghidra and dewolf output a for-loop and Hex-Rays a while-loop.
- The mixed constant representation of dewolf.
- The usage of casts, i.e., Ghidra used nearly twice as many casts than IDA and that dewolf used no casts.
- Hex-Rays superior type and array reconstruction.

In the second part, we questioned the general assumption that the structure of the decompiled output should always be similar to the assembly structure. Therefore, we restructured a sample in three different ways, once by copying a sequence of code (Figure 16a), one by using an additional variable to reconstruct the flow (Figure 16b), and once by extracting part of the code into a function (Figure 16c). Additionally to the comparison in Figure 16, we also asked the same question when the marked region consists of one, five, or 15 lines of code. As expected, copying one line of code was alright when it simplifies the structure. However, the more complicated and longer the structure gets, the participants do not like that it is copied. The idea of extracting complicated structures into a function was picked up very positively by most participants.

Because the full questionnaires and results from all three surveys are extraordinarily extensive, we opted to publish them on GitHub [1]. The repository also contains a PDF with the name `summary_surveys.pdf` rendering the questionnaires and results of all three surveys.

```c
undefined8 FUN_00401c3d(void){
  size_t sVar1; undefined8 local_68;

  long local_18; int local_10;
  uint local_c; local_68 = 0;

  printf("Enter any binary number: ");
  __isoc99_scanf(&DAT_00403259,
↪   &local_18);
  local_c = (uint)local_18;
  for (; 0 < local_18; local_18 =
↪   local_18 / 10000) {
  local_10 = (int)local_18 +
↪   (int)(local_18 / 10000) *-10000;
  if (local_10 == 0x457) {
    sVar1 = strlen((char *)&local_68);
    *(undefined2 *)((long)&local_68 +
↪   sVar1) = 0x46; }
  else { if (local_10 < 0x458) {
    if (local_10 == 0x456) {
      sVar1 = strlen((char
↪   *)&local_68);
      *(undefined2 *)((long)&local_68
↪   + sVar1) = 0x45; }
    else { if (local_10 < 0x457) {
      if (local_10 == 0x44d) {
        sVar1 = strlen((char
↪   *)&local_68);
          *(undefined2 *)((long)
↪   &local_68 + sVar1) = 0x44; }
      else { if (local_10 < 0x44e) {
        if (local_10 == 0x44c) {

  printf("Binary number:
↪   %lld\\n",(ulong)local_c);
  printf("Hexadecimal number:
↪   %s",&local_68);
  return 0; }
```

(a) Output generated with the decompiler integrated into Ghidra 10.0.3.

```c
long sub_401c3d() {
  size_t var_5; long i;
  long var_0; long var_2; long var_4;
  long * var_3;
  printf(/* format */ "Enter any
↪   binary number: ");
  var_3 = &var_0;
  __isoc99_scanf(/* format */ "%lld",
↪   var_3);
  var_2 = 0L;
  for(i = var_0; i > 0L; i /= 0x2710){
    var_4 = i % 0x2710;
    switch(var_4) {
    case 0:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x30; break;
    case 1:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x31; break;
    case 10:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x32; break;
    case 11:
      var_3 = &var_2;
      var_5 = strlen(var_3);
      *(&var_2 + var_5) = 0x33; break;

   printf(/* format */ "Binary number:
↪   %lld\\n", var_0 & 0xffffffff);
  var_3 = &var_2;
  printf(/* format */ "Hexadecimal
↪   number: %s", var_3);
  return 0L;
```

(b) Output generated with dewolf integrating all improvements from Section III.

```c
_int64 sub_401C3D(){
  char s[8]; // [rsp+0h] [rbp-60h]
↪   BYREF
  __int64 v2; // [rsp+8h] [rbp-58h]

  __int64 v8; // [rsp+38h] [rbp-28h]
  char v9; // [rsp+40h] [rbp-20h]
  __int64 v10; // [rsp+50h] [rbp-10h]
↪   BYREF
  int v11; // [rsp+58h] [rbp-8h]
  unsigned int v12; // [rsp+5Ch]
↪   [rbp-4h]
  *(_QWORD *)s = 0LL;
  v2 = 0LL;

  v9 = 0;
  printf("Enter any binary number: ");
  __isoc99_scanf("%lld", &v10); v12 =
↪   v10;
  while ( v10 > 0 ) {
  v11 = v10 % 10000;
  if ( v11 == 1111 ) {
    *(_WORD *)&s[strlen(s)] = 70; }
  else if ( v11 <= 1111 ) {
    if ( v11 == 1110 ) {
      *(_WORD *)&s[strlen(s)] = 69; }
    else if ( v11 == 1101 ) {
      *(_WORD *)&s[strlen(s)] = 68; }
    else if ( v11 <= 1101 ) {
      if ( v11 == 1100 ) {
        *(_WORD *)&s[strlen(s)] = 67;
↪   }

  v10 /= 10000LL; }
  printf("Binary number: %lld\\n",
↪   v12);
  printf("Hexadecimal number: %s", s);
  return 0LL;}
```

(c) Output generated with Hex-Rays and IDA 7.61 (SP1).

Fig. 15: Decompiler output excerpts for the function from Figure 9 used for the third user survey.

```c
/* Block #0 */
if(var_0 > 10){
  while(var_1 > 0){
    if(var_2 == 2){ /* Block #4 */
      if(var_4 == 4){ /* Block #8 */
        continue;}
      /* Block #9 */
    }else{/* Block #5 */}
    while(var > 1){
      printf("Enter a number \\n");
      scanf("%d", &numb1);
      if(var % numb == 0){var/=numb;}
      else{var -= numb;}
      printf("The new number is %d,
↪   \\n", var);}
    return var;}}
/* Block #3 */
if(var_3 > 3){/* Block #6 */}
else{/* Block #7 */}
/* Block #11 */
while(var > 1){
  printf("Enter a number \\n");
  scanf("%d", &numb1);
  if(var % numb == 0){var /= numb;}
  else{var -= numb;}
  printf("The new number is %d, \\n",
↪   var);}
return var;
```

(a) Copying the marked region.

```c
/* Block #0 */
if(var_0 > 10){
  while(true){
    if(var_1 > 0){
      exit_1 = 0;
      break;}
    if(var_2 == 2){ /* Block #4 */
      if(var_4 == 4){ /* Block #8 */
        continue;}
      /* Block #9 */
    }else{/* Block #5 */}
    exit_1 = 1;
    break;}}
if(var_0 <= 10 || exit_1 == 0){
  if(var_3 > 3){/* Block #6 */}
  else{/* Block #7 */}
  /* Block #11 */}
while(var > 1){
  printf("Enter a number \\n");
  scanf("%d", &numb1);
  if(var % numb == 0){var /= numb;}
  else{ var -= numb; }
  printf("The new number is %d, \\n",
↪   var); }
return var;
```

(b) Adding an additional variable exit_1.

```c
/* Block #0 */
if(var_0 > 10){
  while(var_1 > 0){
    if(var_2 == 2){ /* Block #4 */
      if(var_4 == 4){ /* Block #8 */
        continue;}
      /* Block #9 */
    }else{/* Block #5 */}
    return call_sub(var);}}
/* Block #3 */
if(var_3 > 3){/* Block #6 */}
else{/* Block #7 */}
/* Block #11 */
return call_sub(var);

int call_sub(int var){
  while(var > 1){
    printf("Enter a number \\n");
    scanf("%d", &numb1);
    if(var % numb == 0){var /= numb;}
    else{var -= numb;}
    printf("The new number is %d,
↪   \\n", var);}
  return var;}
```

(c) Extracting part of the code into a function.

Fig. 16: The comparison of different restructuring options to avoid gotos for loops with multiple exits.