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 or firmware 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 encountered 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 developed 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 lead to its results considerably exceeding other decompilers. We publish a prototype implementation of dewolf and all survey results on GitHub [15], [40].

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 [2]. Typically, most security analysts heavily rely on manual analysis to achieve their given objective, including a deep understanding of a binary's capabilities. During this usually non-automated analysis, or more precisely manual static analysis, a *decompiler* is a crucial tool to derive a source code-like representation of a given binary. The comprehension of decompiled code is typically far less complicated than the binary's disassembly. Therefore, a decompiler may significantly accelerate and ease analysis, rendering it an essential tool for binary analysis in general. However, due to the information loss during compilation, a decompiler can not entirely revert the compilation process and has to make compromises that fit the developers' objective. Consequently, most approaches for decompilation are developed specifically for a certain use-case or with a particular focus, such as aiming at recompilability for further automated processing [6], [43]. Although using a decompiler is an essential part of the static analysis workflow [44], [49], only a few recent academic publications on decompilation focus on *human* analysis [19], [47]. Besides, the recent DARPA Cyber Grand Challenge showed that the performance of autonomous binary analysis is still far behind the abilities of reverse engineering experts [1]. Although researchers like Mantovani et al. [31] try to understand how human analysts approach unknown binaries, their research is still in its very early stages. We still deem manual analysis as one of the most important applications for decompilation. As a consequence, 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. For example, we revise the restructuring of DREAM++ by enabling the use of continue statements for control-flow interruptions and by refining the for-loop recovery. Furthermore, dewolf incorporates various common algorithms such as subexpression elimination as well as the elimination of dead code, loops, and paths. Moreover, we introduce a novel approach for out-of-SSA, without case distinctions, that potentially reduces the number of variables, and a custom logic engine. Finally, we present various readability improvements such as utilizing compiler idiom handling, eliminating redundant casts, and improving the representation of constants. Although some improvements include seemingly minor modifications, such as the display format of constants in the output, they lead to considerable enhancements of the output quality for human analysts according to our evaluation.

Besides, we achieve a remarkable level of stability for our *research* decompiler using continuous reliability and correctness testing (see Section III-E). Overall, our developed decompiler is more stable than prototype implementations provided for other research decompilers such as DREAM++, while also considerably exceeding the code quality of previous approaches. We publish the prototype implementation of dewolf as Binary Ninja plugin [15] 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). Because we focus on a human-readable decompiler output, user surveys seem like the most reasonable choice for evaluation even given their natural limitations (see Section IV-D). 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 contributed to the identification of problems that we

approached in Section III, such as developing a custom logic engine in order to simplify logic expressions in the output. Finally, in the third user survey, we included a comparison with other state-of-the-art decompilers to assess the significance of our approach. Overall, we received 135 complete responses, with each survey having at least 37 participants and gathering many valuable results and insights. Although our initial re-implementation of `DREAM++` was already very close to the state-of-the-art decompilers Ghidra and Hex-Rays, the survey results for dewolf show significant improvements. More precisely, the final survey results indicate that dewolf's output for the provided function is clearly favored over others and ranked best by more than four out of five participants. Furthermore, the survey results illustrate that dewolf's output is indeed suitable for malware comprehension.

One key finding of the user surveys suggests that decompilation should consider bolder approaches to reconstruct the control-flow. Apparently, most current decompilers try to construct 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. 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 [15] 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

One of the earliest and most relevant publications in the field of decompilation is the PhD thesis of Cifuentes [8]. In this thesis, Cifuentes proposed a modular decompiler architecture where the actual decompilation algorithms, such as control- and data-flow analysis, are decoupled from those depending on the input binary architecture or the output high-level language. To the best of our knowledge, most modern decompilers follow the proposed principle of a modular design. The author also discussed general decompilation challenges and introduced algorithmic approaches to address various of them. For example, Cifuentes proposed recovering 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.

Emmerik [41] suggested the usage of the static single assignment form (SSA-form) of a program to facilitate data-flow analysis needed for decompilation. The author argued that classical data-flow algorithms, such as expression propagation, can be significantly simplified and sped up when applied to the program in SSA-form. While transforming a program into its SSA-form is straightforward, finding an optimal algorithm for an *out-of-SSA* transformation is not trivial [39]. Nevertheless, many existing intermediate languages support both a SSA- and a non-SSA-form to facilitate data-flow analysis algorithms, including those introduced by Binary Ninja.

There are several academic decompilers that each focus on addressing specific decompilation 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 *goto*-statements as possible to increase the comprehensibility of the decompiled code.

The Retargetable Decompiler (RetDec) [26] also utilizes a modular design. To disassemble the binary, Křoustek et al. use capstone [38], and LLVM IR [30] as an intermediate language to minimize the overhead of adding support for new architectures. Overall, there are many other publications regarding RetDec addressing different aspects such as compiler idiom handling [25], [27].

Another novel research approach for decompilation is RevEngE [4]. Instead of decompiling the whole sample, they developed a *debug-oriented decompilation* approach. More precisely, during debugging, the analyst can decompile specific code regions. Furthermore, it is possible to decompile regions more than once with different parameters.

Engel et al. [14] improved structural analysis and extended it to handle C-specific control statements such as *break*, *continue* and *return*. They call this extension to structural analysis *single entry single successor (SESS) analysis*. One goal of this extension is to reduce the number of goto-statements because they argue that as few gotos as possible improve readability. Additionally, to reduce the number of goto-statements, the authors define so-called *tail regions*, which are regions that implicitly define an exit edge and its successor. The authors also use a *combined-condition pattern* which combines two if-statements having one common branch, resulting from conditions such as `if`(A & B).

Schwartz et al. [6] proposed *iterative refinement* as an alternative for structural analysis with a focus on soundness. They remove control-flow edges that impede existing algorithms by inserting a goto to allow the restructuring to continue. Additionally, they introduced the *semantic preservation* technique to guarantee that the original meaning of a program remains unchanged during restructuring. They also suggested *correctness* as a metric of decompiled output quality. The approach was implemented in the Phoenix decompiler that is based on the BAP binary analysis framework.

Yakdan et al. [47] developed *pattern-independent control-flow restructuring*, a novel algorithm to recover C control-flow constructs without generating any goto-statements. 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 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 a *semantic-preserving transformation* technique that converts so-called *abnormal loops* into their single-entry-exit equivalent. Although such transformations allow subsequent goto-free restructuring, they lead to more complex control-flow graphs and ultimately longer output. The authors implemented a decompiler prototype called `DREAM++` as an IDA Pro 6.5 plugin and further improved it by introducing readability enhancements. They also conducted a user survey to evaluate their approach [45].

The revng-c decompiler [19] is implemented on top of the revng binary analysis framework and introduces another new yet partially similar approach to produce goto-free output. The authors argue that the main obstacle to high-quality human-readable decompilation is that complex control-flow graphs contain a lot of tangled paths. Consequently, they *comb* complicated paths in the control-flow graph by duplicating particular graph nodes, which are either single basic blocks or already restructured and collapsed regions. The resulting output is indeed goto-free and has a less tangled control-flow, but it may contain duplicated code snippets. In order to avoid extensive copying of large regions, the authors perform *untangling* which is the duplication of specific paths leading to the exit of the control-flow. When untangling is necessary, it is performed before combing to duplicate fewer nodes. Presumably, there is no universally valid guideline stating whether duplicated code should be preferred over a more complex control-flow as this may often depend on the given function and the analyst's personal preference.

In contrast to research approaches, commercial decompilers strongly focus on providing a stable environment and usability features to facilitate manual and automated binary analysis. The Hex-Rays decompiler [21] integrated into IDA Pro can be considered the de-facto industrial standard. Although a few talks discuss some of Hex-Rays decompilation techniques [17], [18], the main decompilation approaches and algorithms remain unpublished. On the other hand, Ghidra [34], the reverse engineering framework developed by the NSA, is widely considered to be a free alternative to Hex-Rays and 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. While the developers did not scientifically publish any of their utilized approaches, both the documentation and source code are available on GitHub [35]. Moreover, there are other less commonly mentioned decompilers besides Ghidra and Hex-Rays including FoxDec, JEB, angr, radare2's r2dec, Rizin's Cutter, 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 [23], [24], [28]. 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 or traditional 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 [23], [24] or even use traditional control- and data-flow recovery to improve the output [28].

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. Even though 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. Additionally, finding a bug in incorrect decompilation is far more straightforward in a codebase employing a traditional approach than identifying the reasons for the model producing wrong results. Finally, as current approaches illustrate, 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 of which the assembly cannot be understood as easily as of simple functions. Unfortunately, real-world programs and especially malware samples are typically rather large which renders current neural-network-based approaches unsuitable for reverse engineering of real-world malware.

III. APPROACH

In this section, we introduce *dewolf*, our improved approach for decompilation based on the previous state-of-the-art research approach `DREAM++`. This section is structured as follows: First, we will discuss our decompiler and intermediate language selection criteria in Section III-A. Next, in Section III-B, we will introduce our approach dewolf and describe its improvements over previous approaches. Finally, we will conclude this section with additional details about some specifics of dewolf (see Sections III-C to III-E).

A. Framework Selection Criteria

Our goal is to improve the decompiler output in terms of readability by developing new and improving existing approaches. The first step is to determine which decompilers or decompilation approaches are suitable for readability-focused research. Previous studies have shown that gotos significantly aggravate analysts following the control-flow during static analysis [14], [19], [46]. Consequently, we decided to follow a goto-free approach and base our research on such an approach. To the best of our knowledge, `DREAM++` and revng-c are the only two recent approaches producing goto-free output. Therefore, these are the only two suitable decompilation approaches as a base for dewolf.

While both of those approaches use quite similar techniques for control-flow recovery and restructuring, revng-c uses code duplication in addition to structural variables to avoid goto-statements. However, even though Gussoni et al. [19] use untangling to reduce the amount of copied code, they still cannot prevent the duplication of large code blocks. Although inserting a few lines of code may be beneficial when notably simplifying the control-flow, inserting large and potentially nested code snippets is often undesirable. We ultimately opted to use the algorithms introduced by `DREAM++` over those from revng-c as code duplication may be unfavorable for the readability of the output. Additionally, `DREAM++` achieved noteworthy results in their conducted user surveys while also

focusing on human analysis. Finally, starting with an approach that does not copy any instructions allows us to deliberately choose when, where, and how many instructions we copy in a future approach, potentially respecting a user's preference.

Because the openly published prototype implementation of `DREAM++` was not well-maintained and does not work on recent IDA versions, we decided to re-implement the algorithms introduced in the corresponding publications [45], [46]. As part of the adoption, we want to base our approach on an intermediate language of an existing binary analysis framework due to the following reasons. First, this avoids re-implementing well-known low-level algorithms such as translating a given function to SSA-from. Second, this also ensures that our approach is platform-independent. We decided not to integrate our approach into an existing decompiler-framework such as Ghidra or Hex-Rays because this usually impairs their workflow and requires extensive implementation efforts independent from the approach itself.

There are multiple intermediate languages and related lifters that can lift a given binary file each having their advantages and limitations [11], [18], [30]. 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 intermediate language to assist data-flow analysis, (iii) typed variables instead of registers as well as the elimination of stack-usages 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 opted to use the Medium Level Intermediate Language (MLIL) from Binary Ninja in SSA-form. The MLIL allows us to develop decompilation approaches platform-independently while still supporting all platforms supported by Binary Ninja. Additionally, MLIL implements our desired abstractions like variables instead of registers and offers a well-maintained and -documented API. The impact of Binary Ninja on our results is limited to basic and well-known algorithms described in the selection criteria.

B. Methodology

In the remainder of this section, we only describe differences or improvements we made over the existing approach. All details not explicitly mentioned in this section are equal or at least similar to the publications related to `DREAM++` [46], [47]. For instance, the control-flow restructuring has been kept identical except for some minor adjustments described below and some improvements of the switch-recovery. As already mentioned, the prototype implementation of `DREAM++` as IDA Pro plugin is quite fragile and lacks support for 64 bit binaries. Consequently, we re-implemented its algorithms as a Binary Ninja Plugin [42] utilizing the medium-level intermediate language (MLIL) in SSA-form as a starting point.

Figure 1 shows the structure of dewolf. Similarly to the decompiler structure proposed by Cifuentes [8], 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, more specifically, the output produced by Binary Ninja in our prototypical implementation. The majority of our improvements to

Fig. 1: The four phases of the dewolf decompiler.

enhance the decompiler output or stability over `DREAM++` are implemented as pipeline stages and are platform-independent, like the universal decompiling machine proposed by Cifuentes.

Generally, we introduce three types of improvements: First, we include commonly known or already published algorithms not yet implemented by `DREAM++`, e.g., using `continue` statements in loops. Second, we improved some of the algorithms already integrated into `DREAM++`, such as stability improvements for for-loop recovery. Finally, we introduce a few completely new algorithms, like a novel approach for out-of-SSA using graph coloring. Two of the more complex improvements will be discussed more thoroughly in Sections III-C and III-D. To illustrate the improvements, Figure 2 shows an example function that has been decompiled with the initial re-implementation of `DREAM++` and also with the most recent version of dewolf. The introduced improvements have a significant impact on the readability and the structure of the output, at least for this particular function.

In the remaining part of this section, we will give a short overview of achieved improvements over `DREAM++` and shortly summarize each improved algorithm. Because most improvements are either well-documented elsewhere or relatively small, we will not discuss most of them in more detail. Since we have published our prototype on GitHub [15], anyone interested in more specific details of the introduced algorithms can look up the well-documented source code.

1) Liveness Analysis: We use liveness analysis to construct the interference graph that we need in various algorithms of dewolf. For example, we use it to create the dependency graph used during the out-of-SSA transformation. In our approach, we utilize liveness sets by exploring paths from variable usages according to Brandner et al. (Algorithm 4) [5].

2) Switch Variable Detection: To enhance the comprehension of complicated control-flows, programmers frequently use switch statements. However, in assembly generated by modern compilers, switch control-flows are often implemented using indirect jumps and jump-tables. To recover switch control-flows in those cases, we reconstruct the original switch variable by tracing the variable used to calculate the jump-offset until we reach its definition. Eventually, we can use this information to restructure a region using a switch to increase the readability.

3) Subexpression Elimination: After propagating expressions, there still may be certain expressions that 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 expressions that are already defined and used in other instructions as well as expressions that occur multiple times. Using the dominator tree, we find the closest possible

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

(a) Source code of an example function.

```
1  int64_t xor(int64_t arg1, int64_t arg2, int64_t arg3,
↪      int64_t arg4){
2      printf("%lu",arg2);
3      int32_t var_c = 0x0L;
4      while(true){
5          int64_t rax_14 = var_c;
6          if ((arg2 <= var_c)){ break; }
7          char* rax_4 = (arg1 + var_c);
8          uint64_t rsi_1 = *rax_4 ;
9          int64_t rax_6 = var_c;
10         int64_t rdx_1 = 0x0L;
11         char* rax_9 = (arg3 + ((rdx_1:rax_6) % arg4));
12         uint64_t rcx = *rax_9;
13         int64_t rax_12 = (arg1 + var_c);
14         uint64_t rdx_4 = (rsi_1.esi ^ rcx.ecx);
15         *rax_12 = rdx_4.dl;
16         int32_t var_c = (var_c + 0x1L); }
17     return rax_14; }
```

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

```
1  long xor(void * arg1, long arg2, long arg3, long arg4){
2      int i;
3      int var_1;
4      void * var_0;
5      printf(/* format */ "%lu\n", arg2);
6      for (i = 0; arg2 > i; i++){
7          var_0 = arg1 + i;
8          *var_0 = *var_0 ^ *(arg3 + (i + (0L << 0x40)) %
↪              arg4); }
9      var_1 = i;
10     return var_1; }
```

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

Fig. 2: Comparison between the output of our DREAM++ re-implementation and dewolf.

position to its usages for insertion to avoid unnecessary interference between variables. Eventually, we insert a definition for the common subexpression at the identified location and replace all its occurrences.

4) Constant Representation: In general, constant values are often displayed as hex-numbers in the decompiler output. However, as constant values can represent integers, characters, addresses, or else, a hexadecimal 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.

5) 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 quite 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. Therefore, we can recover for-loops in many additional cases by altering and relaxing the rules proposed by DREAM++. 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.

6) Compiler Idioms: Unresolved compiler idioms 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 consider compiler idioms, we decided to include algorithms to revert them. More specifically, we opted to integrate a very recently published approach for compiler idiom handling [12] which uses automatically generated assembly patterns to annotate and revert the most common idioms.

7) Array Access Detection: In assembly language or intermediate languages, the concept of arrays does not exist. Instead, an array element access is represented as a dereference operation to a variable plus an offset, i.e., *(base + valid_offset). We find such candidates and mark them as potential array element access. Furthermore, we compute the base and the index, depending on the offset and the array size. Eventually, we discard candidates not fulfilling certain requirements, such as a consistent array size. The decompiler output displays dereference operations marked as array access as base[i].

8) 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 usually is not part of the analyst's objective and may significantly decrease efficiency. We identify such conditions and then eliminate *dead* paths or paths that are unreachable. Similarly, we can also remove dead loops which occur when the loop condition is always unsatisfied during the first loop entry. Overall, both algorithms remove unreachable basic blocks and can significantly reduce analysis workload.

9) 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 to access only parts of a given register, various cast operations can be introduced when lifting the assembly code to a more high-level intermediate representation. 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.

10) Continue in loops: The DREAM++ approach solely uses break-statements to indicate when the control-flow may exit a given loop. In contrast, we improved the control-flow restructuring by also using continue-statements to decrease the nesting depth where possible. More specifically, during the pattern-independent restructuring proposed by DREAM++, we do not exclusively add break-nodes on edges used to exit the loop. Instead, we additionally add continue-nodes on every edge having the loop-head as a sink.

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. As a consequence, decompilers often employ a logic engine to simplify such expressions [45] but also to, e.g., identify unreachable code during the elimination of dead loops/paths or 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 [10], z3 can be considered the most relevant theorem prover and state-of-the-art.

Unfortunately, although being the state-of-the-art SMT solver, z3 is not optimal for our purpose either. Although being a full-fledged theorem prover and offering 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:

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

Even though this simplification does not alter the logic, it raises the complexity and length of the output considerably. This lack of simplification is particularly relevant since unsigned operations are predominant when handling pointers on any architecture. Additionally, z3 has problems simplifying conditions that often occur during the restructuring, e.g., z3 is not able to make the following simplification:

$$(a = 1 \lor a = 2 \lor a = 3 \lor a = 4) \models 1 \leq a \leq 4.$$

Without a doubt, the complexity of logic statements as well as 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 provided by z3, we designed it for the particular use case of modeling logic expressions generated from assembly language. More specifically, using a custom logic engine allows us to include exactly the simplifications we need during decompilation.

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. Further, each operation and bit vector (representing a variable or constant) is represented by a graph node. Finally, we represent the relation between the operations and bitvectors 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 [33]. Besides, it offers many practical advantages, e.g., we do not need sophisticated methods to deal with the ordering of operands or to copy and replace terms in an expression. Finally, we can traverse edges to compute the graph predecessors or successors to view relations easily, e.g., between subexpressions. We open-sourced the logic engine alongside the decompiler on GitHub [16].

D. Improved Out-of-SSA

Many data-flow-algorithms are applied to a program in SSA-form because it simplifies many algorithmic challenges [41]. However, because in SSA-form each variable is defined exactly once, basic blocks with multiple entries may cause variables to have a different value depending on the control-flow. To resolve this problem, φ-*functions* are used to assign variable values depending on the control-flow. Conceptually, all φ-functions are executed simultaneously in each basic block. However, there is no analogous concept of φ-functions in the C-language. Thus, we have to eliminate them at some point to obtain valid C output. Unfortunately, removing φ-functions is not trivial due to the lost-copy and swap-problem [39]. Hence, it is no surprise that there is still ongoing research about this topic [5], [39]. However, most of these algorithms are rather complicated and distinguish many different cases. Therefore, we developed a more straightforward algorithm. One benefit of our approach is that we do not distinguish between multiple cases depending on the usage of variables in the φ-functions of a basic block.

Our approach consists of three algorithms, namely *removing circular dependencies* of φ-functions, *lifting* φ-functions, and *renaming* variables. It is possible to remove the circular dependency and lift the φ-functions first, and then rename the variables or the other way around. Similar to other approaches, we remove the circular dependency on φ-functions by inserting copies. We argue that 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 $\varphi_{l+1} = \varphi_1$). Hence, it is impossible to order φ-functions such that we execute them successively when they depend circularly on each other. After resolving the circular dependencies, we can order the φ-functions into a possible execution order. Given such an order, it is possible to move the instructions represented by the φ-functions to the predecessor blocks. In the following, we will describe each of the three steps in more detail. Figure 3 illustrates an overview of the approach applied to a cfg.

1) Remove circular dependency: As mentioned before, the goal of this algorithm is to insert copies such that we can execute the φ-functions one after another. To find the variables for which we have to insert copies, we construct a dependency graph as shown in Figure 3b. This graph contains one vertex for each variable that is defined via a φ-function and we add an edge (u, v) between two variables u, v if v is used in the φ-function defining u. This means, having the φ-functions $u = \varphi(0, v)$ and $v = \varphi(y, z)$, we add the edge (u, v) to the dependency graph. The edge (u, v) indicates that we have to execute the φ-function defining u before the φ-function

(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.

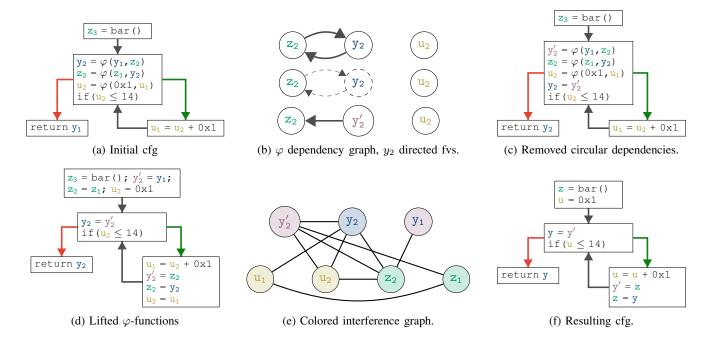Fig. 3: Illustration of each step from our out-of-SSA algorithm applied to an example cfg.

defining v because we need the value of v before the φ-functions reassigns it for the computation of u. Now, a cycle in the dependency graph is equivalent to having φ-functions that depend circularly on each other.

To resolve the circular dependency between the φ-functions, we compute a directed feedback vertex set (directed fvs) in the dependency graph; a directed fvs is a set X of vertices whose removal results in an acyclic graph. Although computing a directed fvs is NP-hard [22], we can find such a set in constant time because we only have a constant number of vertices in each basic block and because we do this computation for each basic block independently. Nevertheless, we only use an approximation to compute the directed fvs. Now, for every φ-function $x = \varphi(\ldots)$ defining a variable x contained in the directed fvs X, we replace the φ-function by $\text{copy\_}x = \varphi(\ldots)$. Furthermore, we add the instruction $x = \text{copy\_}x$ after the last φ-function because variable x must have the value of $\text{copy\_}x$ after all as shown in Figure 3c. Afterward, the dependency graph is acyclic because it does not contain any vertex of the set X, and all copies are isolated in the dependency graph. Finally, we arrange the φ-function according to a topological order of the dependency graph, implying that we can execute them in this order.

2) Lift φ-functions: To apply this algorithm the circular dependency of the φ-functions must be resolved and the φ-functions must be ordered in a valid execution order. Thus, this algorithm always runs directly after resolving the circular dependencies and sorts the φ-functions accordingly.

Since the φ-functions can be executed successively, we can "lift" the instructions represented by each φ-function to the predecessor basic blocks. More precisely, for each φ-function $x_0 = \varphi(x_1, x_2, \ldots, x_l)$ we have a mapping that maps from each predecessor basic block to the variable or constant assigned to x_0 when the flow enters over 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 basic block does not end with a branch. Otherwise, we add a new basic block between the current basic block and BB_i and add the assignment in the new basic block. After this step, all φ-functions are removed by moving the definitions to the predecessor basic blocks as shown in Figure 3d. The lifted assignments are ordered according to the order of the φ-functions. After this step, the program is no longer in SSA-form.

3) Rename Variables: There are many possible ways to rename the variables. The easiest renaming strategy is to assign each variable its name in SSA-form, i.e., the variable v with label 4 gets name v_4. However, this leads to many variables and many copy assignments.

In general, when renaming variables, we only have to make sure that two variables solely get the same name when they do not interfere. One possible way to achieve this is to use graph coloring. More precisely, given any valid coloring of the interference graph like in Figure 3e, we can assign all variables of the same color class the same name. Sadly, finding a vertex-coloring with a minimum number of colors is NP-hard [22]. However, there are many special cases where this problem is solvable in polynomial time, e.g., in chordal graphs, and there exist multiple heuristic and greedy algorithms. Currently, we use lexicographical BFS to find the order in which we color the vertices. We always use an existing color, if possible, and introduce a new color otherwise. To get rid of as many copy-assignments as possible, we have some additional rules on how to choose between multiple possible existing colors.

As mentioned before, it is possible to apply the algorithms in two different orders. When renaming variables first, the program is still in SSA-form which implies that the interference graph is chordal [20]. Thus, we can compute an optimal coloring in linear time using our coloring algorithm. Although

this initially leads to a minimal number of variables, starting with the renaming can produce more circular dependencies between the φ-functions. Hence, we may have to insert more copies when resolving the circular dependencies.

On the other hand, the interference graph may no longer be chordal once we removed the φ-functions by removing the circular dependencies and lifting them. Consequently, our algorithm may not compute an optimal coloring. However, due to our additional rules on how to choose between the existing colors and the fact that the graph is nearly chordal, the result is still sufficient.

E. 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. This section outlines our approach for decompiler testing and the strategies we used to improve dewolf's reliability and correctness.

1) Decompiling a fixed set of 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 to our continuous integration pipeline to proactively identify problematic functions for new analyses.

2) Fuzzing with Csmith: We fuzzed dewolf with Csmith-generated functions [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.

3) Differential fuzzing of re-compiled code: 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 (i) do not call other functions, (ii) do not use global variables, and (iii) do not use structures or unions. By excluding vast swathes of possible programs, we can focus on semantic transformations applied by dewolf.

We tested the originally compiled binaries against their decompiled and re-compiled counterparts. To perform the tests, we generated harnesses for use with the Nezha framework [36]. The differential fuzzing tests helped us to fix multiple integer promotion casting bugs as well as type recovery bugs which we would not have identified otherwise. Figure 4 shows an example for an identified bug.

Our testing experience showed that fuzzing and differential testing are readily applicable to decompilers. While we were only 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

As already mentioned, our improvements for decompilation approaches from the previous sections are heavily focused on enhancing decompilation for manual static analysis by humans.

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

(a) Original source-code, adapted from the LLVM test suite [29], resulting in `test_case(69794316) == 64012` (2 bytes).

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

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

Fig. 4: 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.

In general, the quality of code can either be evaluated using quantitative measurements, i.e., code metrics, or by conducting user surveys. Some of the most common code metrics are *Source Line of Code*, *Halstead Software Science*, *ABC*, *McCabe Cyclomatic Complexity*, *Henry and Kafura's Information Flow*, and *Nesting depth*. However, a major downside of using such metrics for evaluation is that each metric favors a particular behavior which may not necessarily lead to a better code readability. For example, the revng-c approach [19] uses the McCabe Cyclomatic Complexity for their evaluation. However, this metric 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 code metrics such as McCabe Cyclomatic Complexity *and* Lines of Code are not sufficient to assess the readability since reducing the lines of code does not always help a human analyst. For example, tweaking expression propagation to generate rather long lines of code can be quite difficult to understand for a human analyst although reducing the lines of code. To conclude, each code quality metric considers only one aspect of the code and hence has advantages and disadvantages [3]. Consequently, the most natural way to evaluate our approach in terms of readability is to conduct user surveys. Principally, a user survey is *the only* way to measure improvements in readability or comprehension for a human analyst, as they can be measured by comparing outputs or extracting a baseline from the answers. Therefore, using an extensive survey to study analysts' preferences regarding code constructs or output formats is most appropriate.

As described in Section III, our decompilation approach consists of various improvements which we can either evaluate individually or collectively as a whole. Although the individual

		Responses		**Time (in minutes)**		
#	Execution	Total	Full	25%	50%	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 month of execution, the number of total resp. full responses, and the average resp. median time per full response.

Fig. 5: 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.

and incremental evaluation would allow assessing the individual impact of each improvement, this would result in an unmanageable number of combinations which goes far beyond what we are able to evaluate with user surveys. Additionally, we presume that the impact of each individual contribution is rather small and may not result in measurable improvements. Consequently, we opted for an evaluation of our decompiler approach combining all introduced improvements.

During our research, we conducted a total of three user surveys, each with a focus on the evaluation of one of the following three aspects: First, identifying the limitations of DREAM++, finding constructs preferrable by analysts, as well as relevant research questions. Second, we evaluated whether the improved output of our approach is sufficient to comprehend a realistic malware sample since this is a crucial task of a decompiler. Finally, we compared our approach to the open-source and commercial state-of-the-art decompilers Ghidra and Hex-Rays. In general, each conducted user survey consisted of multiple parts and was of considerable length (see Table I). In addition to the already mentioned parts, we also included questions for self-assessment and feedback regarding the survey itself. Especially the feedback about the surveys helped us improving them in each iteration. One disadvantage of such a broad user survey is the high number of incomplete responses. However, after the first survey, we managed to decrease the proportion of incomplete responses by optimizing the question structure.

1) Survey Participants: Gathering participants for user surveys with a highly specific topic like decompilation or reverse engineering is not particularly easy. Certainly, 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.

However, it is arguable that people with a solid understanding of the C programming language can participate in our survey as well. First, most decompiler outputs are syntactically very similar to C code which is comprehensible by programmers, computer scientists, and others. 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 [31]. Consequently, we did not limit our survey to participants with experience in reverse engineering and were pleasantly surprised by the extensive participation.

In Figure 5 one can see that most participants have experience with C according to their self-assessment (see Sec-

tion 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. Nevertheless, we gathered more than 40% of participants with allegedly substantial reversing skills in the final survey. The contribution of the skill levels is not surprising since our participants mainly consisted of students who completed a lecture in malware or program analysis, colleagues with experience in malware analysis or at least with C coding skills, and some professional reverse engineers.
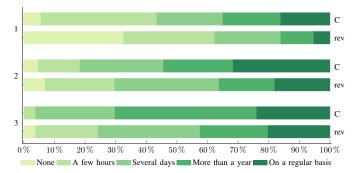
2) Survey Platform: We opted to use an online platform to conduct our surveys to not be limited to geographically close participants, to avoid time constraints, and to ensure anonymous data collection. More specifically, we used a privately hosted instance of LimeSurvey [13]. At the beginning of each survey, we sent a survey link to potential participants. In total, each survey was active for two weeks, allowing the participants to find time to complete the survey questionnaire.

Due to the page limit, we will only discuss a subset of the survey questions and results in this paper. Appendix A contains a few more details on some of the results. Additionally, we separately published the complete questionnaires, results, and samples of all surveys on GitHub [40].

A. Survey 1: Limitations of Dream

One main goal of the first user survey was to evaluate how our re-implementation of the DREAM++ approach performs against the commercial and open-source state-of-the-art. We decided to use the Ackermann function (Figure 6) compiled with GCC 10.0.1 since it is a non-trivial arithmetic function featuring non-primitive recursion. Then, we decompiled the Ackermann function with dewolf (DREAM++ re-implementation), Ghidra 9.1.2., and Hex-Rays 7.5. SP2, and we randomly assigned each participant one output. Overall, we collected data about the comprehension and subjective trends regarding the presentation of the three different outputs.

To verify how well the participants understood the given function and to validate the participant's 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 7 shows the comprehension results divided by the decompiler that generated the output

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

Fig. 6: Source code snippet for the first user survey. The function `A(...)` implements the Ackermann function.
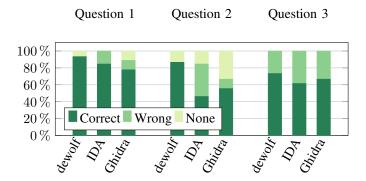

Fig. 7: Comprehension results of the Ackermann function. The questions are included in Appendix A.

for the respective participant. Additionally, we asked some 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.

We used the second part of the survey to compare small decompiled samples of all three decompilers. Therefore, each participant was presented with all three outputs side-by-side and rated them concerning various criteria like different control-flow structures. One goal was to detect possible improvements for the re-implementation of DREAM++, and another goal was to gain an insight into user preferences.

Overall, our decompiler achieves results only slightly diverging from the state-of-the-art, except for compiler idioms, which were handled neither in the initial implementation nor the DREAM++ approach. Furthermore, the survey indicates that even the re-implementation of DREAM++ can still compete with state-of-the-art decompilers. Consequently, DREAM++ provides a great starting point for additional research.

In the third part of the survey, we tried to find general user preferences regarding decompiler output. On the one hand, there are many code constructs or output formats where researchers and users have a clear consensus. For example, most would agree that a high nesting depth or representing an integer value as a binary number is generally undesirable. On the other hand, there is a variety of cases where seemingly no clear preference exists such as the usage of gotos. Therefore, we selected code constructs to understand user preferences regarding different aspects. Further, we investigated the construct's effects on comprehension from a user's perspective. Additionally, we validated whether ideas that originated from related work or previous research are worth further investigation. The topics we asked participants include but are not limited to preferences regarding program structure like switch versus if-else, value and expression propagation, the use of goto, duplicate code, loop types, the display of various condition types, and common subexpression elimination.

It turned out that many participants were particularly dissatisfied with seemingly small details that may still require elaborated approaches, for example, the representation of constants or the restructuring of (for-)loops. Moreover, the fact that modern decompilers do not handle all compiler idioms turned out to be a substantial issue. Consequently, we decided to tackle issues criticized by participants regarding the output of our re-implementation of DREAM++. Furthermore, we also approached two more complex and particularly substantial issues mentioned by participants, namely the complexity of expressions and conditions and the number of variables and their names. To address these problems, we developed a custom logic engine (Section III-C) 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. Additionally, 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, in developing dewolf and new approaches, we considered numerous possibilities for new configurations. Therefore, many algorithms in dewolf are highly configurable by the user.

B. Survey 2: Comprehension

The goal of the second survey was to evaluate the comprehensiveness of dewolf's output. More precisely, to assess the applicability of dewolf, we validated whether participants can use dewolfs output to successfully comprehend a realistic malware function. 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 [37]. As a consequence, 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.

Unfortunately, we only have a limited number of survey participants and each only agrees to spend a limited amount of time for the surveys. In addition, real-world malware samples are exponentially increasing in size as well as complexity [7], and DGAs regularly are implemented across multiple functions and may have dependencies which are both hard to represent properly in a user survey. Consquently, we had to decide between a real-world DGA that potentially would lead to less or even none completed survey participations and a synthetic DGA which we opted for eventually. To make the DGA as realistic as possible, we studied numerous real-world DGAs using

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

Fig. 8: The artificial domain generation algorithm of the second survey which produces domains with 8 random characters excluding the letter z with four different TLDs.

Fig. 9: DGA Comprehension results of the second survey for all participants and split by C-skill level.

DGArchive [37] and programmed a very similar yet slightly less complex DGA in a single function that still generates realistic domains as a real-world DGA would. Figure 8 shows the DGA used for this survey.

Each participant received dewolf's output of the given DGA, compiled with VS2015, depicted in Figure 15. Then, we asked the participants about the purpose of the function to assess whether the participants determine its functionality, i.e., that it is a DGA. In total, more than 60% of the participants 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 to all participants 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 by the participants. Most participants were able to identify the used TLDs and correctly classified *.com* as the most used TLD. Finally, we showed the participants five domains and asked whether the given algorithm could potentially generate these domains. Again, the majority of the participants answered those questions correctly. The results of the comprehension questions are summarized in Figure 9 whereas all questions are contained in Appendix B.

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 9 shows that C skills seem to have an impact on the answer score of the participants. 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 which we
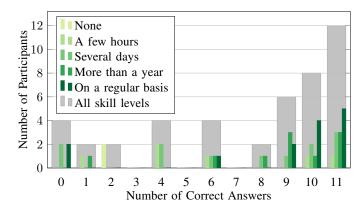
consequently did not consider in the final survey. Similar to the first survey, this feedback significantly contributed to the approach introduced in Section III. Overall, the second survey demonstrates that dewolf produces output that allows analyzing malware successfully. Furthermore, it appears that adequate C skills can be advantageous but are not crucial.

C. Survey 3: Comparison

The primary goal of the final user survey was to compare our approach introduced in Section III (dewolf) with 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 to additional approaches due to the survey limitations discussed in Section IV-D. Additionally, at the time of conducting the third survey, neither the Pseudo-C output from Binary Ninja nor revng-c were publicly released. The output from other decompilers we included in the second survey turned out to produce output so significantly less readable than Hex-Rays and Ghidra that we chose to not consider them in the final user survey (see [40]).

For the comparison, we constructed a sample that contains as many different difficult decompilation aspects as possible while retaining a manage-able size. We compiled the sample function from Figure 10 with GCC 10.3.1 and decompiled the resulting sample with Hex-Rays and Ghidra (Figure 16). 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 11). The results indicate that dewolf was the most favored for the considered function. Also, Hex-Rays was still significantly ahead of Ghidra. This result clearly underlines the positive impact of all introduced approaches, although each individually being relatively simple, at least on the participants' perception.

All in all, the participants positively mentioned several aspects of dewolf's output. For instance, 44 participants praised the switch structure we reconstructed in contrast to a higher nesting depth produced by the other two decompilers. Another convincing aspect was the lack of unnecessary casts

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

22             case 1110: strcat(hex, "E"); break;
23             case 1111: strcat(hex, "F"); break; }
24         binary /= 10000; }
25     printf("Binary number: %lld\n", binary);
26     printf("Hexadecimal number: %s", hex);
27     return 0; }
```

Fig. 10: Code Snippet for the third user survey to generate the hex representation of a given binary number.
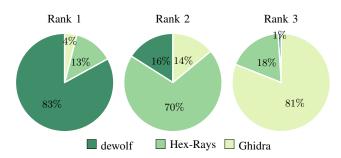
Fig. 11: Ranking of the decompilers for the function in Figure 10. A total of 53 participants ranked dewolf best.

in dewolf's output. However, the participants also disliked some details about dewolf's output. For example, dewolf was incapable of correctly detecting the array, whereas Hex-Rays was. To clarify, although dewolf was ranked first in this sample, we do not claim that dewolf is superior in general. Nevertheless, we want to point out that 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 (again) asked the participants some questions regarding their preferences for decompiler output. The idea behind this part was to question the generally taken assumption that reverse engineers want highly accurate decompilation output, i.e., an output similar to the assembly structure. Unsurprisingly, many decompilation approaches and commercial decompilers aim to achieve this goal [6], [43]. Nevertheless, some approaches like DREAM++ may already cause functions being restructured slightly different from their original appearance to some extent. Overall, our impression is that most approaches try to generate output very similar to the assembly structure.

We decided to scratch the surface of this unwritten rule by including a few questions to verify whether survey participants would accept substantial differences to the assembly structure in favor of increasing the decompiler output's readability. For instance, the participants had to choose between three equivalent reconstruction options for the same function. The function was restructured by either having a more complicated structure, containing various levels of duplicated code, or an

alternative version where a section was extracted into a new artificial function. In total, we evaluated three restructuring approaches for different scenarios to obtain an overall impression. Figure 17 contains an example of this question type.

As expected, some participants complained that a decompiler 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. Survey Limitations and Threats to Validity

The evaluation of a decompilation approach with user surveys has certain limitations, and consequently has our evaluation. Nevertheless, we will point out in this section why user surveys are favorable when evaluating readability improvements despite their limitations. The most notable limitation is that only a small number of people are willing and able to participate in a user survey related to reversing. Especially the number of professional reversers is rather small and becomes even smaller when only considering the ones willing to spend time on a user survey. Additionally, recent research indicates that even popular paid services claiming to provide qualified survey participants can and should not be used to overcome this limitation [9]. Consequently, like previous studies with a similar topic [31], [47], we have to work with a relatively limited number of participants compared to less-technical user surveys.

Next, the limited number of participants also influences the survey extent as each participant is only willing to spend a certain amount of time. The first user survey shows that most participants would not spend significantly more than 30min to complete a survey. Even the samples we utilized in our surveys resulted in noticeably longer participation times (see Table I) and already caused issues with participants not completing the survey. Thus, using real-world malware samples which are far more complex, individually evaluating the impact of all improvements, or comparing our approach against more than one or two other approaches is unfeasible. Therefore, we decided to evaluate the combined impact of all introduced improvements of dewolf using real-world-like samples. To tweak the difficulty and participation time, we had student assistants do test runs of each survey.

The last limitation of our survey design is the lack of an appropriate assessment of participants' skills for each survey. In the first survey, we included both a self-assessment and an assessment based on questions about the Ackerman Function. Since we did not observe any significant divergences between those two, we decided to only include self-assessment questions in the following surveys. Having only the self-assessment saved valuable participants' time and allowed us to include additional content. Consequently, when considering results split by the participant's skills, these are solely built on the provided self-assessment and should be tempered with caution. Regardless, the survey results indicate that the participant's reversing skills are far less relevant compared to the capability of understanding code which is very helpful for comprehending a given function.
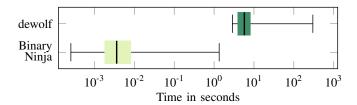
Fig. 12: Boxplots visualizing the runtime when decompiling the samples in Coreutils with dewolf and Binary Ninja.

Despite all those limitations of user surveys, they are still the best way to evaluate approaches that improve readability for analysts. Although researchers commonly use code metrics to assess the quality of their code-base, these are inadequate for our evaluation: The first user study shows that no sound default configuration for decompiler outputs exists. Consequently, it is impossible to define formal criteria for code quality because readability is merely subjective. To summarize, regardless of the discussed limitations (see Section IV-D) user surveys are the most fitting method to evaluate decompilation approaches focusing on readability.

E. Runtime Evaluation

We heavily focus on readability improvements and consequently neither optimized our approach nor our prototype implementation in terms of 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 important aspect of decompilation approaches. To demonstrate that even without speed optimizations or being implemented in native code, our Python prototype is sufficiently fast on real-world binaries, we decompiled all binaries in Coreutils on an i7-10850H with 12GBs of RAM and measured the decompilation times. Given a timeout of 5 minutes, the dewolf prototype was able to decompile 94% of all binaries and 75% of them in less than 8 seconds. Figure 12 shows the measured decompilation times for dewolf and also for Binary Ninja. As presumed, the decompilation times of Binary Ninja are significantly faster. Because all of Binary Ninja's analyses run at the start of dewolf to obtain the MLIL, they can be considered a lower bound for dewolf's runtime.

F. Summary and Outlook

We conclude that future research on decompilation should presumably not be limited to reconstructing a binary's assembly structure as accurately as possible. Instead, to help human analysts, it may be highly beneficial to improve readability and comprehensibility by allowing more flexible output generation while maintaining semantic equivalence. Our survey results indicate that users accept decompiler output with changed program structure to improve the readability. Consequently, we have to overthink existing approaches that heavily avoid substantial changes to the program structure and instead take them actively into consideration. Ultimately, our results establish multiple possibilities for decompilation, opening entirely new options for output formats. Additionally, it encourages techniques that were previously hastily discarded due to the above reasons.

The results of our surveys also imply that configurability is deeply desired by many participants. 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. In conclusion, configurability should be considered for future decompilation approaches.

Finally, we expect that the survey results may 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 [40].

V. CONCLUSION

In this paper, we made three main contributions. First, we conducted and discussed three user surveys providing valuable information for research in the field 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 to the limitations on current approaches from a user perspective, and (iii) as starting points for new research. A key finding of our surveys is that the majority of participants do not necessarily favor decompilers producing output as close as possible to the structure implied by the assembly. Instead, we discovered that users prefer more readable and comprehensive code even when it introduces substantial structural differences to the disassembly. These results may have a broad impact on future research since existing approaches rarely oppose the disassembly. Furthermore, the evaluation indicates that rather small details, like the representation of constants, can already have a positive impact in some cases. Consequently, future decompilation approaches should allow user configuration for such details when possible. This result raises the question of whether the existing metrics for decompiler evaluation apply to human analysis.

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 and 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 Hex-Rays and Ghidra 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.

13

REFERENCES

[1] "z3defcon ctf final scores." https://www.defcon.org/html/defcon-24/dc-24-ctf.html, 2022.

[2] AV-TEST GmbH, "Malware Statistics & Trends Report," https://www.av-test.org/en/statistics/malware/, accessed: 2022-01-28.

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

[4] 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, pp. 1–12.

[5] 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.

[6] 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, pp. 353–368.

[7] 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, pp. 325–345.

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

[9] 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, pp. 537–548.

[10] 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, pp. 337–340.

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

[12] 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, pp. 1–7.

[13] N. C. Engard, "Limesurvey http://limesurvey. org: Visited: Summer 2009," 2009.

[14] 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, pp. 21–27.

[15] FKIE-CAD, "dewolf," https://github.com/fkie-cad/dewolf, 2022.

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

[17] I. Guilfanov, "Decompilers and beyond," *Black Hat USA*, pp. 1–12, 2008.

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

[19] 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, pp. 637–651.

[20] S. Hack, *Interference graphs of programs in SSA-form*. Universität Karlsruhe, Fakultät für Informatik, 2005.

[21] Hex-Rays, "Hex-rays decompiler," https://hex-rays.com/decompiler/, 2022.

[22] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[23] 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, pp. 346–356.

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

[25] J. Křoustek, "Retargetable analysis of machine code," p. 190, 2015.

[26] ——, "Weather forecast for today? advert flood coming from east," [online], October 2015, available on http://now.avg.com/weather-forecast-for-today-advert-flood-coming-from-east/.

[27] 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, pp. 1507–1514.

[28] R. Liang, Y. Cao, P. Hu, and K. Chen, "Neutron: an attention-based neural decompiler," *Cybersecurity*, vol. 4, no. 1, pp. 1–13, 2021.

[29] LLVM Project, "test-suite Guide," [Online] https://llvm.org/docs/TestSuiteGuide.html, 2020, accessed: 2020-09-23.

[30] ——, "LLVM Language Reference Manual (Version 13.0)," [Online] https://llvm.org/docs/LangRef.html, 2021, accessed: 2022-01-28.

[31] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/mantovani

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

[33] P. Mutzel, "Algorithmic Data Science (Invited Talk)," in *36th International Symposium on Theoretical Aspects of Computer Science (STACS 2019)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), R. Niedermeier and C. Paul, Eds., vol. 126. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 3:1–3:15. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10242

[34] National Security Agency, "Ghidra," https://ghidra-sre.org, 2022.

[35] ——, "Ghidra source code," https://github.com/NationalSecurityAgency/ghidra, 2022.

[36] 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. [Online]. Available: https://doi.org/10.1109/SP.2017.27

[37] 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, pp. 263–278.

[38] C. Project, "Capstone: The ultimate disassembler," https://www.capstone-engine.org, accessed: 2022-01-31.

[39] 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, pp. 194–210.

[40] steffenenders, "Decompiler user survey results," https://github.com/steffenenders/dewolf-surveys, 2020-2021.

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

[42] Vector 35, "Binary Ninja," https://binary.ninja, 2022.

[43] 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, pp. 247–264.

[44] 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, pp. 1875–1892.

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

[46] 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, pp. 158–177.

[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. [Online]. Available: https://doi.org/10.1145/1993498.1993532

[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, pp. 3053–3069.

APPENDIX

A. 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 a set of questions about its comprehensiveness and their judgment of the output quality on a Likert scale. Figure 14 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 on the number of variables, their typing, and restructuring.

B. User Survey II

To verify the comprehensibility of dewolf's output, we decompiled the DGA of Figure 8 with dewolf. The output is shown in Figure 15. On the positive side, dewolf correctly restructures the switch statement as well as the array-access for the domains (variable `var_0`). Additionally, dewolf is capable of reversing the compiler 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 (variable `var_4`). Moreover, dewolf is very conservative with propagation pointers leading to unnecessary copies of variables, for example, in line 7. However, 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   ○ squzuzfz
  - ○ xfbcbcic   ○ rlpmpmgmjdh
  - ○ facebook

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 generated with our initial 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 generated with Hex-Rays and 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 generated with Ghidra 9.1.2.

Fig. 13: Decompiler outputs for the function from Figure 6 used for the first user survey.

- `.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 are *squzuzfz* and *rlpmpmgmjdh* because the character z is excluded in the for-loop and the length of each domain is eight.

C. 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 16. The ranking results are illustrated in Figure 11 and shortly discussed in Section IV. Figure 16c shows

Fig. 14: Boxplots visualizing the Likert scale values evaluated in the first user survey for the outputs from Figure 13.

```c
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. 15: The output of dewolf given the DGA of Figure 8.

Although this does not strictly reflect the assembly due to the missing jump table calculation, we argue it is easier to understand. 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 be similar to the assembly structure. Therefore, we restructured a sample in three different ways, once by copying a sequence of code (Figure 17a), one by using an additional variable to reconstruct the flow (Figure 17b), and once by extracting part of the code into a function (Figure 17c). Additionally to the comparison in Figure 17, 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.

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 16a). In contrast to both, dewolf restructured this part into a switch statement by identifying a candidate variable for a switch construct (Figure 16b).

```
1   undefined8 FUN_00401c3d(void){
2     size_t sVar1; undefined8 local_68;

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

24    printf("Enter any binary number: ");
25    __isoc99_scanf(&DAT_00403259,
   ↪     &local_18);
26    local_c = (uint)local_18;
27    for (; 0 < local_18; local_18 =
   ↪     local_18 / 10000) {
28      local_10 = (int)local_18 +
   ↪     (int)(local_18 / 10000) *-10000;
29      if (local_10 == 0x457) {
30        sVar1 = strlen((char *)&local_68);
31        *(undefined2 *)((long)&local_68 +
   ↪     sVar1) = 0x46; }
32      else { if (local_10 < 0x458) {
33        if (local_10 == 0x456) {
34          sVar1 = strlen((char
   ↪     *)&local_68);
35          *(undefined2 *)((long)&local_68
   ↪     + sVar1) = 0x45; }
36        else { if (local_10 < 0x457) {
37          if (local_10 == 0x44d) {
38            sVar1 = strlen((char
   ↪     *)&local_68);
39            *(undefined2 *)((long)
   ↪     &local_68 + sVar1) = 0x44;
   ↪     }
40          else { if (local_10 < 0x44e) {
41            if (local_10 == 0x44c) {

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

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

```
1   long sub_401c3d() {
2     size_t var_5; long i;
3     long var_0; long var_2; long var_4;
4     long * var_3;
5     printf(/* format */ "Enter any
   ↪     binary number: ");
6     var_3 = &var_0;
7     __isoc99_scanf(/* format */ "%lld",
   ↪     var_3);
8     var_2 = 0L;
9     for(i = var_0; i > 0L; i /= 0x2710){
10      var_4 = i % 0x2710;
11      switch(var_4) {
12      case 0:
13        var_3 = &var_2;
14        var_5 = strlen(var_3);
15        *(&var_2 + var_5) = 0x30; break;
16      case 1:
17        var_3 = &var_2;
18        var_5 = strlen(var_3);
19        *(&var_2 + var_5) = 0x31; break;
20      case 10:
21        var_3 = &var_2;
22        var_5 = strlen(var_3);
23        *(&var_2 + var_5) = 0x32; break;
24      case 11:
25        var_3 = &var_2;
26        var_5 = strlen(var_3);
27        *(&var_2 + var_5) = 0x33; break;
76     printf(/* format */ "Binary number:
   ↪     %lld\\n", var_0 & 0xffffffff);
77     var_3 = &var_2;
78     printf(/* format */ "Hexadecimal
   ↪     number: %s", var_3);
79     return 0L;
```

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

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

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

23    v9 = 0;
24    printf("Enter any binary number: ");
25    __isoc99_scanf("%lld", &v10); v12 =
   ↪     v10;
26    while ( v10 > 0 ) {
27      v11 = v10 % 10000;
28      if ( v11 == 1111 ) {
29        *(_WORD *)&s[strlen(s)] = 70; }
30      else if ( v11 <= 1111 ) {
31        if ( v11 == 1110 ) {
32          *(_WORD *)&s[strlen(s)] = 69; }
33        else if ( v11 == 1101 ) {
34          *(_WORD *)&s[strlen(s)] = 68; }
35        else if ( v11 <= 1101 ) {
36          if ( v11 == 1100 ) {
37            *(_WORD *)&s[strlen(s)] = 67;
   ↪     }

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

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

Fig. 16: Decompiler output excerpts for the function from Figure 10 used for the third user survey.

```
1   /* Block #0 */
2   if(var_0 > 10){
3     while(var_1 > 0){
4       if(var_2 == 2){ /* Block #4 */
5         if(var_4 == 4){ /* Block #8 */
6           continue;}
7         /* Block #9 */
8       }else{/* Block #5 */}
9       while(var > 1){
10        printf("Enter a number \\n");
11        scanf("%d", &numb1);
12        if(var % numb == 0){var/=numb;}
13        else{var -= numb;}
14        printf("The new number is %d,
   ↪     \\n", var);}
15      return var;}}
16  /* Block #3 */
17  if(var_3 > 3){/* Block #6 */}
18  else{/* Block #7 */}
19  /* Block #11 */
20  while(var > 1){
21    printf("Enter a number \\n");
22    scanf("%d", &numb1);
23    if(var % numb == 0){var /= numb;}
24    else{var -= numb;}
25    printf("The new number is %d, \\n",
   ↪     var);}
26  return var;
```

(a) Copying the marked region.

```
1   /* Block #0 */
2   if(var_0 > 10){
3     while(true){
4       if(var_1 > 0){
5         exit_1 = 0;
6         break;}
7       if(var_2 == 2){ /* Block #4 */
8         if(var_4 == 4){ /* Block #8 */
9           continue;}
10        /* Block #9 */
11      }else{/* Block #5 */}
12      exit = 1;
13      break;}}
14  if(var_0 <= 10 || exit_1 == 0){
15    if(var_3 > 3){/* Block #6 */}
16    else{/* Block #7 */}
17    /* Block #11 */}
18  while(var > 1){
19    printf("Enter a number \\n");
20    scanf("%d", &numb1);
21    if(var % numb == 0){var /= numb;}
22    else{ var -= numb; }
23    printf("The new number is %d, \\n",
   ↪     var); }
24  return var;
```

(b) Adding an additional variable exit_1.

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

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

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

Fig. 17: The comparison of different restructuring options to avoid gotos for loops with multiple exits.