A Comb for Decompiled C Code

Andrea Gussoni
Politecnico di Milano
andrea1.gussoni@polimi.it

Alessandro Di Federico
`rev.ng` Srls
ale@rev.ng

Pietro Fezzardi
`rev.ng` Srls
pietro@rev.ng

Giovanni Agosta
Politecnico di Milano
agosta@acm.org

ABSTRACT

Decompilers are fundamental tools to perform security assessments of third-party software. The quality of decompiled code can be a game changer in order to reduce the time and effort required for analysis. This paper proposes a novel approach to restructure the control flow graph recovered from binary programs in a semantics-preserving fashion. The algorithm is designed from the ground up with the goal of producing C code that is both `goto`-free and drastically reducing the mental load required for an analyst to understand it. As a result, the code generated with this technique is well-structured, idiomatic, readable, easy to understand and fully exploits the expressiveness of C language. The algorithm has been implemented on top of the `rev.ng` [11] static binary analysis framework. The resulting decompiler, `revng-c`, is compared on real-world binaries with state-of-the-art commercial and open source tools. The results show that our decompilation process introduces between 40% and 50% less extra cyclomatic complexity.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Software reverse engineering**; Security requirements.

KEYWORDS

decompilation, reverse engineering, goto, control flow restructuring

ACM Reference Format:
Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *15th ACM Asia Conference on Computer and Communications Security (ASIA CCS'20), October 5–9, 2020, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3320269.3384766

1 INTRODUCTION

In the last decades, software has steadily become increasingly ubiquitous, and programmable electronic devices are nowadays part of every aspect of everyone's life. Most often, users have little control on the software that runs on these devices and on the life cycle of release upgrades to fix outstanding bugs. Companies tend to be very secretive about their implementations and rarely provide access to the source code of their applications, either to protect patents and trade secrets, or in the hope to provide security by obscurity. In some fields, it is also common to find legacy code that runs parts of critical infrastructures, for which gaining access to the source is not even an option, since the company that originally provided it ran out of business.

In all these scenarios, it is challenging for external analysts to conduct independent security assessments of the implementations, let alone to provide fixes for bugs and vulnerabilities.

In this context, performing an in-depth analysis of a piece of software without access to its source code is significantly more difficult. To this end, *decompilers* are powerful tools that, starting from a binary executable program, can reconstruct a representation of its behavior using a high-level programming language, typically C. These tools save the analyst from the need of looking directly the assembly code, leading to a dramatic reduction of the effort necessary to perform a security assessment, making it viable in new scenarios.

The compilation process is not perfectly reversible, which complicates the task of evaluating the quality of the results of a decompiler. Due to aggressive compiler optimizations and hand-written assembly, it is often impossible to recover the exact original source from which a binary executable was produced. A decompiler could even be used to recover C code from a Fortran program. In principle the process should work, but the recovered C code would not be the original source nor very idiomatic C.

Therefore, in practice, the goal of a decompiler is not really to produce the exact same source code that originated the program, which might be plainly unfeasible, but to produce some high-level representation easy for analysts to reason about. For this reason, it is of very important for a decompiler to produce high-quality code.

The *quality* of decompiled code can be measured in different ways. Informally, it can be described as the readability of the code, i.e., the ease with which a snippet of decompiled code can be understood by an analyst. This qualitative measure is strongly related to the mental load necessary to understand the behavior of the code, which in turns depends on the amount of information that the analyst has to track during the analysis. This information can be ascribed mainly to the complexity of the *control flow* and depends on all the possible entangled execution paths that can lead to a certain portion of the code. All these factors contribute to the mental load of an analyst.

To minimize such load, and to produce high-quality output code, decompilers adopt various techniques to restructure the control flow of decompiled programs, to make them easier to read and to reduce the burden of understanding their functioning. As an example, control flow restructuring can be used to reduce the number of `goto` statements [22], cutting the number of unstructured jumps across the program, hence reducing the mental load necessary to track all the possible paths. As another example, control flow restructuring can be used to produce `if-then-else` or loops that naturally match high-level programming constructs, or to collapse multiple `if`s in a single one if they check the same condition. All these modifications on the control flow contribute to make the code easier to understand.

In summary, this paper makes the following contributions:

- we present a novel algorithm for control flow restructuring that 1) produces well-structured programs that can always be emitted in C, without resorting to `goto` statements, 2) significantly reduces the *cyclomatic complexity* [18] of the generated C code compared to the state of the art, a measure of the complexity of the control flow strictly related to the mental load required to understand the observed code, 3) fully exploits the expressiveness of the C language (such as short-circuit of `if` conditions and `switch` statements);
- we implement the proposed approach employing the `rev.ng` binary analysis framework as a basis;
- we compare the resulting decompiler, `revng-c`, with state-of-the-art commercial and academic decompilers, on a set of real world programs, measuring the size of the decompiled code and its cyclomatic complexity.

The remainder of this work is structured as follows. Section 2 introduces the fundamental concepts necessary to understand the rest of the work. Section 3 discusses related works while Section 4 presents the design of the control flow restructuring algorithm. Section 5 shows the experimental results obtained on a set of real-world programs, the GNU coreutils, comparing the approach proposed in this work with other commercial, open-source and academic decompilers: the Hex-Rays Decompiler[1, 13], Ghidra[2], and DREAM[21, 22]. Finally, Appendix B discusses more idiomatic case studies and corner cases before the concluding remarks in Section 6.

2 BACKGROUND

This section briefly outlines the main concepts that are necessary for the understanding of the paper.

Graph basics. In this paper we give for granted a number of fundamental concepts revolving around Directed Graphs. For the interested reader, these concepts are discussed in more detail in Appendix A, and a more distinguished reference can be found in [14].

Most of this concepts should be familiar, since they are widely used in program analysis for representing the control flow of a program by means of Control Flow Graphs (CFG). In particular, we will make wide use of the following concepts.

- *control flow graph* representation of a program.
- *directed acyclic graphs* (DAG).
- *search and visits over CFGs.* In particular, we will make extensive use of the *Depth First Search* algorithm, and of the orderings it induces on a CFG, as the *preorder*, *postorder* and *reverse postorder*.
- *dominance* and *post-dominance*, and the data structures they induce, the *dominator-* and *post-dominator-tree*.

Short-circuit evaluation. In this paper, *short-circuit evaluation* refers to the semantics of boolean expressions in C.

If a boolean expression has more than one argument, each argument is evaluated only if the evaluation of the previous arguments is not sufficient to establish the value of the expression containing the boolean operator. This is particularly important when the evaluation of some operand of the boolean expression have side-effects, because only the side-effects of the arguments that are actually evaluated will be triggered.

Cyclomatic Complexity The *cyclomatic complexity* is a well-known software metric used to capture the complexity of a program. It was originally conceived by T. J. McCabe in 1976 [18].

It represents a quantitative measure of the number of linearly independent paths in a program source code. The cyclomatic complexity is computed on the control flow graph of a program.

In general, the formula to compute the cyclomatic complexity of a program is given by $M = E - N + 2P$, where M is the cyclomatic complexity itself, E is the number of edges in the CFG, N is the number of nodes in the CFG and P represents the number of connected components in the graph.

In the case of a single subroutine P is always 1 hence the formula can be simplified to $M=E-N+2$. If we consider a program as the union of all the CFGs of its subroutines the cyclomatic complexity of the program can be computed as the sum of all the cyclomatic complexities of the single subroutines.

3 RELATED WORK

This section describes the related work in two main fields: recovery of Control Flow Graphs, and decompilation.

CFG Recovery. In this work we focus on control flow restructuring, and we use as a starting point the Control Flow Graph of the function we want to analyze and decompile.

The problem of correctly recovering such graphs from binary code is well-known and lot of research work has been done in this field. The CMU Binary Analysis Platform (BAP) [4] is a binary analysis framework which disassembles and lifts binary code into a RISC like intermediate language, called BAP Intermediate Language (BIL). BAP also integrates all the techniques developed previously for BitBlaze [19]. The `rev.ng` [8, 9] project, which is an architecture independent binary analysis framework based on QEMU [3] and LLVM [16], is able to lift a binary into an equivalent LLVM IR representation. Other research groups have also dedicated efforts to tackle the problem of disassembling obfuscated code [15].

The approach presented in this paper does not rely on any specific technique for extracting CFGs from binary code, hence it is general enough to be used with any of these approaches.

Decompilation. The academic foundational work in the field of decompilers is probably Cifuentes' PhD thesis [7]. The techniques presented there have been implemented in the dcc decompiler, which is a C decompiler for Intel 80286.

In the field of commercial decompilers, Hex-Rays[1] is the defacto leader, and its decompiler is provided as a plug-in for the Interactive Disassembler Pro (IDA) [13] tool. No specific information on the internal structure of the decompiler is publicly available, apart from the fact that it uses some kind of structural analysis [12].

Phoenix [5], a decompiler tool built on top of the CMU Binary Analysis Platform (BAP) [4], uses iterative refinement and semantics-preserving transformations. The iterative part is implemented through the emission of a `goto` instruction when the decompilation algorithm cannot make progress.

A very recent entry in the field of decompilers is Ghidra, which is included in the Ghidra reverse engineering tool [2], initially developed by US National Security Agency (NSA) for internal use. The tool has been open sourced very recently (April 2019) but, as of today, no extensive documentation on the design of the components has been released.

The decompiled code generated from Ghidra bears some similarities with the Hex-Rays Decompiler, and this suggests that it also uses an approach based on structural analysis. In particular, both tools, the Hex-Rays Decompiler and Ghidra, emit C code with many `goto` statements, which is not idiomatic and results in very convoluted control flow. This fact makes it hard to keep track of all the entangled overlapping control flow paths in the code, making it hard to understand. This characteristic is also somehow shared by the Phoenix decompiler, which emits `goto` statements when it cannot make further progress.

The DREAM [22] decompiler takes a drastically different direction. The authors present various semantic-preserving transformations for the CFG, and a decompilation technique that emits no `goto` statements by design. However, to avoid `goto`s, DREAM handles "pathological" loops by means of what can be seen as predicated execution. If a CFG has a loop and a branch that jumps straight in the middle of the loop from a point outside the loop, DREAM wraps parts of the body of the loop inside a conditional statement guarded by a state variable. This design choice prevents `goto`s but generates code where multiple execution paths are entangled and partially overlap. An example can be seen in an open dataset of code snippets released by the authors [20], in Section 1.5, page 7, at lines 12–16 of code generated by DREAM. In larger functions, this can significantly increase the mental load of an analyst, especially if a loop contains more than one of these conditional blocks, possibly nested or with multiple conditions.

4 CONTROL FLOW COMBING

In this paper we make a novel choice for the generation of decompiled code that is free from `goto` statements: we accept to duplicate code in order to emit more idiomatic C code that reduces the mental load of an analyst, being more readable and easier to understand. This section focuses on the details of this technique, called *Control Flow Combing*.

The algorithm is composed of 3 stages: a *Preprocessing*, which prepares the input CFG to the manipulation, transforming it into a hierarchy of nested Directed Acyclic Graphs (DAG); the actual *Combing* stage, which disentangles complex portions of the control flow by duplicating code portions or introducing dummy nodes; a final *Matching* stage, which matches idiomatic C constructs, while trying to reduce unnecessary duplication.

Informally, the idea is to "*comb*" the Control Flow Graph, duplicating code to disentangle convoluted overlapping paths, so that the properties necessary to emit idiomatic C code naturally emerge. We pay this potential duplication as a cost necessary to handle generic binary programs. To gracefully handle common cases, the *Matching* step is performed as post-processing to reduce duplication when possible, leaving freedom and generality to the *Combing* without sacrificing the capability to emit high-quality code.

The remainder of this section is structured as follows. Section 4.1 provides an overview the high-level design goals. Section 4.2 introduces the CFG properties that are later enforced by the *Preprocessing* and *Combing* stages. Section 4.3 provides a general overview before digging into the details of the three stages: *Preprocessing* (Section 4.4), *Combing* (Section 4.5), and *Matching* (Section 4.6).

4.1 Design Goals

The fundamental goal of the algorithm presented here is to increase the quality of the produced decompiled code. As mentioned in Section 1, this means reducing the informative load on the shoulders of analysts. To achieve this, the algorithm is designed with some fundamental goals.

Generality. It must be able to work on any CFG, independently of its complexity. This is important since, in decompilation, input CFGs might be originate from hand-written or compiler-optimized machine code. To build a decompiler that consistently generates high-quality output very few assumptions can be made on the input CFGs.

Structured. It needs to transform any CFG so that it can be expressed in terms of C constructs, excluding `goto`s. `goto`s, and unstructured programming in general, can considerably increase the complexity of the *control flow*[10].

Expressive. Starting from such structured CFGs, it must be able to emit a wide range of idiomatic C constructs, such as `while` and `do-while` loops, `switch` statements, and `if` statements with or without `else` and short-circuited conditions.

4.2 CFG Properties

The *Preprocessing* and the *Combing* stages of the algorithm enforce some properties on the input CFGs. Such properties are inspired to fundamental characteristics of structured C programs and designed to mimic them. The fundamental idea of the algorithm is to enforce each of these properties one at a time. Once a property has been enforced it becomes an invariant, so that it is preserved from all the subsequent steps. In this way, the final result of applying the transformations on the original CFG will feature all the properties. Being these properties tailored to describe structured C programs, the resulting CFG at the end of the algorithm is straightforward to translate in C without `goto`s.
In the following we list the properties we aim to enforce.

Two Successors. The first important property of structured C programs is that each basic block has at most two successors. The only case that does not respect this condition is the `switch` statement, but every `switch` can always be transformed in a sequence of `if-else` statements and vice versa. The *Preprocessing* phase will always enforce this property on CFGs, deferring to the *Matching* stage the decision of whether to emit `if`s or `switch`es.

Two Predecessors. This property holds whenever a basic block in CFG has at most two predecessors.

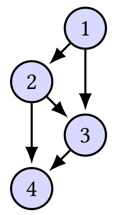


Figure 1: A CFG without *Diamond Shape* property. Node 3 is reachable from 2 but does not dominate it. To emit this code in C, a goto statement would be required (either ⟨1,3⟩ or ⟨2,3⟩).

Loop Properties. In a well-structured C program each loop has the following three characteristics.

Single Entry. In C, the only way to enter a loop, without passing from the entry node, is using the `goto` statement.

Single Successor. In C, the only way to abandon a loop without `goto`s is using the `break` statement, and all the `break` statements in a loop jump to the same point in the program: the single successor of the loop. The case of a natural exit from the loop is just an implicit `break`

Single Retreating Target. In C, all the retreating edges in a structured loop jump to the same target since, given that we have no `goto`s, they must be `continue` statements. Just like `break`s, all `continue`s in a loop jump to the same point: the single retreating target, which is also the entry node.

The fact that `break` and `continue` statements always target a single node is only true under two assumptions. The first is that there are no `switch` statements. The second assumption is that there are no nested loops since, e.g., `break` statements of a nested loop do not jump at the same target as `break`s of its parent loop. These two assumptions might sound strong, but the *Preprocessing* phase is designed to ensure that these properties are enforced in strict order, so that when the *Loop Properties* are enforced all their prerequisite are guaranteed to hold. All the loop properties described above will be enforced on the input CFG from the *Preprocessing* stage.

Diamond Shape Property. The *Diamond Shape* property holds for a DAG whenever each node with two successors dominates all the nodes between itself and its immediate post-dominator. This property is enforced by the *Combing* stage, on the DAGs generated by the *Preprocessing* stage.

This mimics the fact that in well structured C programs all the scopes are either nested or non-overlapping. In other words, enforcing this property means forcing a DAG in the form of a *diamond* where each node with more than a single successor induces a region of nodes with a *single entry* and a *single exit*.

To grasp the implications of this property, it might be useful to think about a scenario where this property does not hold. An example is portrayed in Figure 1. In this setting, it exists a conditional node, node 2 in Figure 1, and another node, node 3, reachable from the conditional node, which is not dominated by the conditional node. There exist another node, node 1, from which it is possible to reach node 3 without passing from node 2. But node 2 is a conditional node (an `if` statement in C), and since node 3 is reachable from node 2, if the program is well-structured node 3 should be either in the `then` or in the `else`, or after the `if-else` altogether. At the same time, there is a path from node 1 to node 3 that does not pass from node 2. The main problem with this scenario, is that such a graph cannot be emitted in well-structured C programs without using `goto`s. Hence the *Combing* stage enforces this property.

4.3 Overview of the Algorithm

As previously anticipated, the *Control Flow Combing* algorithm is designed in three incremental stages: *Preprocessing*, *Combing*, and *Matching*.

Preprocessing. The goal of this stage is to massage the input CFG in a shape that can be digested by the *Combing*. To do this, the *Preprocessing* incrementally enforces all the properties described in Section 4.2, except for the *Diamond Shape* property. It does so by working on a tree-like hierarchy of nested *Regions* of the CFG, called *Region Tree*. At the end of *Preprocessing* all the *Regions* in the tree are transformed into DAGs.

Combing. This stage works on the *Region Tree* generated by *Preprocessing*, which is now constituted only by DAGs. The *Combing* enforces the *Diamond Shape* property on all the DAGs in the tree. After this transformation the tree is ready to be transformed into an C Abstract Syntax Tree.

Matching. This stage uses the combed *Region Tree* to generate a C AST representation. The AST is subsequently manipulated with a set of rules to match idiomatic C constructs. The rules presented in this paper cover short-circuited `if`s, `switch` statements, and loops in the form `do {...} while(...)` and `while(...) {...}`, but others can be added. After matching idiomatic C constructs, the final C code is emitted in textual form.

4.4 Preprocessing

This section describes the *Preprocessing* stage in detail.

The first part of the *Preprocessing*, described in Section 4.4.1, is designed to divide the CFG on a hierarchy of nested *Regions*, each roughly representing a loop. The goal is to superimpose on the CFG a *Region Tree*, that represents the hierarchy of loops in the CFG itself. Each *Region* in the tree is then handled independently of the others by the next steps of *Preprocessing* and *Combing*, reducing the complexity of algorithm.

The second part of the *Preprocessing*, described in Section 4.4.2, works on the *Region Tree*, transforming each *Region* into a DAG, so that it can subsequently be handled by the *Combing* stage.

4.4.1 *Building the* Region Tree.

This process is composed by three steps. The *first* adds a *sink* node as a successor of all the exit nodes, which is necessary to compute post-dominance. The *second* starts to enforce some of the properties discusses in Section 4.2. The *third* identifies nested *Regions* and builds the *Region Tree.*

Adding the *sink* Node. In general, CFGs obtained from binary programs do not have a single exit, which is a requirement to compute the post-dominator tree, which in turn is a requirement to reason about the *Diamond Shape* property that is enforced later.

Hence, every CFG needs to be brought into a shape with a single exit. This is done by adding an artificial *sink* node, and attaching an artificial edge from each original exit basic block to the *sink*. This makes the *sink* the single exit node, allowing to compute the post-dominator tree.

This operation does not alter the semantic of the program, and is preserved by all the following steps.

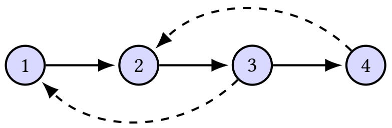


Figure 2: Merging partially-overlapping SCS. There are two SCS ($\langle 1,2,3\rangle$ and $\langle 2,3,4\rangle$) induced by the *retreating edges* $3 \rightarrow 1$ and $4 \rightarrow 2$ (dashed). They overlap but they have no inclusion relationship, therefore they are merged into a new SCS $\langle 1,2,3,4\rangle$.

Enforcing Two Predecessors and Two Successors. First, the *Two Successors* property is enforced by transforming all the `switch` statements into cascaded conditional branches, with two targets each. Similarly, the *Two Predecessors* property is enforced by taking each basic block with more than one predecessor and transforming it into a tree of empty basic blocks (*dummies*) that only jump to their single successor.

These operations do not alter in any way the semantic of the program. Moreover, the *Two Predecessors* and *Two Successors* are preserved in all the following steps.

Identifying Nested *Regions*. The core idea of this step is to merge sets of the partially overlapping loops in the CFG into an individual *Region* that we can then reason about as a single loop.

To define these *Regions*, the algorithm starts from all the *Strongly Connected Subgraphs* (SCS), i.e., subgraphs of the original CFG whose nodes are all reachable from each other. There might be several overlapping and non-overlapping SCS in a graph. Note that a SCS is a difference concept from a *Strongly Connected Component* (SCC), typically used in loop analysis. In fact, SCCs are always non-overlapping by definition, and their union represent the entire CFG. In particular, we are interested in *SCSs induced by retreating edges*. Given a *retreating edge* $e = \langle s,t\rangle$ the *SCS induced by e* is constituted by all the nodes that are on at least one path starting from t and ending in s that does not crosses t nor s.

First, the algorithm identifies all the SCS induced by all the *retreating edges* in the CFG, simply applying the definition above. Note that at this stage the resulting SCSs can still overlap, whereas to build a hierarchy between SCS it is necessary for the set of SCSs to form a partially ordered set with the strict subset relation ($\subset$). Hence, for each pair of SCS A and B, if $A \cap B \neq \emptyset$, $A \not\subset B$, and $B \not\subset A$, then $A \cup B$ is added to the set of SCS, removing A and B from the set of SCSs. When this happens, the algorithm restarts from the beginning, until a fixed point is reached. Notice that the union of two SCS is always an SCS, hence the process can proceed. An example of partially overlapping SCS that trigger this condition is shown in Figure 2.

This process converges since the $\cup$ operator is monotonic and the CFG has a finite number of nodes. At the end only a set of SCS that is partially ordered with the $\subset$ relationship is left. Each of this remaining SCS is a *Region* roughly representing a loop, or a set of loops tightly entangled together. Considering the whole CFG as a *Region* itself, the $\subset$ relationship naturally induces a tree on all the regions. The whole CFG is the root of the tree, and moving towards leaves we encounter more and more deeply nested loops. This tree structure is called the *Region Tree*.

Notice that the grouping of nodes in *Regions* does not alter the CFG, hence it does not alter the program semantic. The same holds if a node is moved inside or outside of an existing *Region*. From this point, all the steps of the algorithm only work on *Regions* and the *Region Tree*.

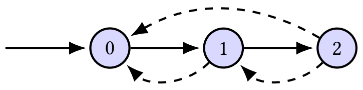


Figure 3: Electing a *Region*'s *head*. The *retreating edges* are dashed. Node 1 has one incoming *retreating edge*, while node 0 has 2. For this reason, node 0 is elected *head* of the *Region*.

4.4.2 *Turning* Regions *into DAGs.*

The goal of this phase is to turn each *Region* into a DAG that can be then be reasoned about in simpler terms. This process is composed of various incremental steps. The combination of all these steps enforces on the *Regions* all the remaining properties introduced in Section 4.2 except for the *Diamond Shape* property, i.e., the *Two Successors* property, and the *Loop Properties*. Where noted, some of steps are optional and dedicated to gracefully handle common cases.

The following steps work on a single *Region* at a time, moving from the leaves to the root of the *Region Tree*. At the beginning of this process all *Regions* but the root are still SCS. At the end of this process the *Regions* are transformed in DAGs, so that they can be treated by the next phase, *Combing*.

Electing *Regions*' Heads. The *Loop Properties* require every *Region* to have a *Single Entry* and a *Single Retreating Target*. However, at this stage, each of them may contain multiple retreating edges, possibly targeting different nodes. This step elects the entry node: the node that is target of the highest number of retreating edges. This node, the *head node*, represents the beginning of the loop body, and will be the target of all the *retreating edges* in the loop.

Retreating Edges Normalization. After the election of the *head*, all the *retreating edges* that do not point to it are considered *abnormal*, since they do not respect the *Single Retreating Target* property and, therefore, need to be handled.

Consider the graph in Figure 3: the head is node 0 and there is a single *abnormal edge* from node 2 to 1. In C parlance, this edge is not a `continue`, since it jumps to the middle of a loop. Informally, to handle this situation, we can introduce a *state variable* in the program so that the *abnormal edge* can be represented with a `continue`. In practice, this edge will target a virtual *head* node that will check the value of the *state variable* and dispatch execution at the correct location (Node 1). To discriminate between *retreating edges*, the *state variable* is set before every *retreating edge* and checked at the beginning of the loop with a dedicated construct.

This is exactly what the normalization step does for *abnormal edges*. For each *Region*, a *state variable* v is created. Then, a distinct identifier is assigned to each node with incoming *abnormal edges*, as well as to the *head* elected at the previous step. Then, a new set of nodes is created before the *head*, containing only conditional jumps that check the *state variable* to dispatch the execution at the correct target (either a target of an *abnormal edge* or the *head*). This set of nodes is called the *head dispatcher*, and its first node is called h. Finally, each *abnormal edge* $e = \langle s,t\rangle$ is replaced with a new pair of edges. The former edge of this pair is $e_h = \langle s,h\rangle$. This edge points to the entry point of the *head dispatcher*, and sets the state variable to the value associated to t, say v_t. The latter edge is added from the node in the *head dispatcher* that checks for the condition $v == v_t$ to

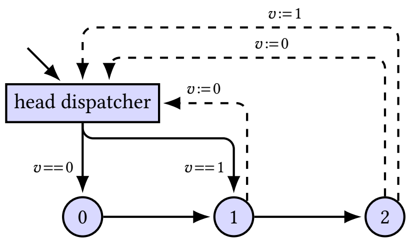


Figure 4: Normalizing *retreating edges* on the CFG from Figure 3. All the *retreating edges* (dashed) now point to the new *head dispatcher* and set the *state variable* (values are reported on the edge labels). The *head dispatcher* then jumps to the original target node.

t. Finally, the single entry point of the *head dispatcher* is promoted to new *head* of the *Region*.

Figure 4 shows the result for the normalization of *abnormal edges* applied to the CFG originally depicted in Figure 3.

The idea is to enforce the *Single Retreating Target* property, editing the control flow without altering the semantics of the original program, except for the introduction of the *state variable* v. This process also preserves the *Two Successors* property. All these properties are from now on invariant and preserved in the next steps.

Notice that redirecting the *abnormal edges* to the *entry dispatcher* may momentarily break the *Two Predecessors* property. But this does not represent a problem since the normalized *abnormal edges* are later removed and substituted with `continue` statements.

Loop Successors Absorption. This is an optional step that starts moving in the direction of the *Single Successor* loop property. It is designed to handle gracefully a scenario observed frequently in real-world example, depicted on the left in Figure 5. The *Region* ⟨0,1,2⟩ in the figure has two successors, 3 and 4. Informally it is easy to see that the *Region*, along with node 3, is substantially a loop that executes the code in node 3 on `break`. Given that one of the goals is to emit idiomatic C, this would be better represented with a loop, containing an `if` statement that executes the code in 3 and `break`s. In order to reach this form, the node 3 must be absorbed into the *Region*, as shown in Figure 5.b.

More formally, this step starts with the creation of new empty *dummy frontier* nodes on each edge whose source is in the *Region* and whose target is not (see the empty dashed node in Figure 5.b). Then, it computes the dominator tree of the entire CFG (not only the current *Region*) and adds to the *Region* all the nodes that are dominated both by the *head* of the *Region* and by at least one *dummy frontier* node.

This embodies the idea that given a node, if it is only reachable passing through the head of *Region* and from a *dummy frontier* it is in fact part of the *Region* itself, and it must be handled accordingly by the remaining steps.

This step does not alter the semantic of the program as it only adds empty *dummy frontiers*, and it also does not break any of the previously enforced invariants.

First Iteration Outlining. This step enforces the *Single Entry* property on *Regions*, removing potential multiple entry points that at this stage are still possible by means of *abnormal entries*. An *abnormal entry* is an edge $e = \langle s,t \rangle$ such that s is not in the *Region*, t is in the *Region*, and t is not the *head*.

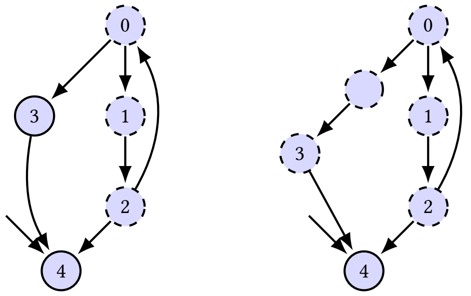
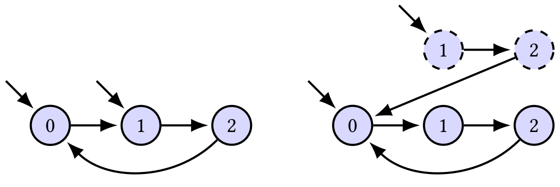


Figure 5: Absorbing Successors. Left – The *Region* with the nodes with dashed border has two successors: 3 and 4. Right – Node 3 has been absorbed in the *Region*, which now has a single successor.



before outlining after outlining

Figure 6: First iteration outlining. Dashed nodes are the outlined.

Abnormal entries are removed based on the observation that each of them generates a set of paths that: enter the *Region*, execute some parts of the loop and at some point reach the proper head of the loop and proceed with regular iterations.

Thanks to this observation, the nodes and edges that compose the first iteration can be duplicated and moved out of the *Region*, since once they are outlined they have no *retreating edges* and bear no signs of being loops.

Note that it would be possible to leave the first iteration inside the loop, but it requires guarding each statement with conditional constructs, an approach adopted by previous works [21, 22]. However, we deem that choice to be suboptimal since it generates decompiled code where paths are entangled together and artificially guarded by conditional constructs. Moving the first iteration outside the *Region* makes it easier to reason about, since it can be analyzed in isolation, while also leading to more idiomatic C code.

Exit Dispatcher Creation. Symmetrically to the creation of entry dispatchers, this step normalizes the *Regions* to completely enforce the *Single Successor* loop property. The *successors absorption* step was an optional step to get the low-hanging fruit in this direction, while gracefully handling common cases, but not all scenarios.

If the *Single Successor* loop property does not hold after the *successors absorption* step, this step injects an *exit dispatcher*, that is built and acts similarly to the *entry dispatcher*, changing the control flow without altering the semantic of the program thanks to a *state variable*. Again, each of the edges $e = \langle s,t \rangle$ with s in the *Region* and t outside is substituted with two edges. The first starts from s, sets the state variable, and jumps to the *exit dispatcher*. The second starts from the *exit dispatcher* and goes to the t. In this way, the first node of the *exit dispatcher* becomes the single successor of the *Region*. Notice that this means that the *exit dispatcher* itself is not part of the *Region* but is part of its parent *Region* in the *Region Tree*.

Figure 7 shows a case where the creation of the *exit dispatcher* is necessary.

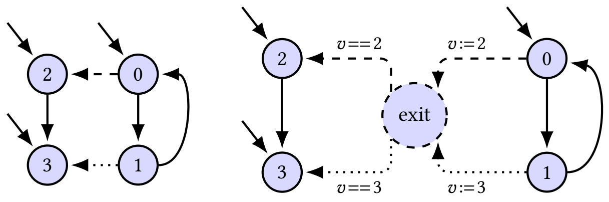


Figure 7: Creating exit dispatcher. Left – The *Region* composed by nodes 0 and 1 has two successors (2 and 3). Right – Creation of the *exit dispatcher*, making it the target of the outgoing edges. The edges also carry the values assigned to the *state variable*, later used dispatch the execution to the real successors.

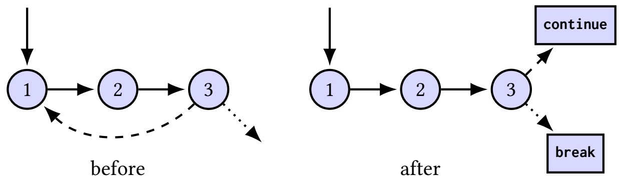


Figure 8: Creation of `break` and `continue` nodes.

Note that this step does not alter the program semantics and does not break any of the previously enforced invariants, since the *exit dispatcher* is built of conditional statements. The *Single Successor* loop property is enforced for all *Regions* and will be preserved by all the following transformations.

Again, this does not alter the program semantics and does not break any of the previously enforced invariants, since the *exit dispatcher* is built of conditional statements. The *Single Successor* loop property is enforced for all *Regions* and will be preserved by all following transformations.

`break` and `continue` Emission. This step transforms each *Region* in a DAG that is then ready to be fed into the *Combing* stage.

After the previous steps, the execution of a program in a given *Region* can either take an exit edge and jump to the single successor, or take a *retreating edge* and jump to the *head* to execute another iteration. At this point, all the properties introduced in Section 4.2 have been enforced, with two exceptions: *Diamond Shape*, that will be enforced later by *Combing* and *Two Predecessors*, that was enforced at the beginning of Section 4.4.1, but that might have been broken during *Retreating Edges Normalization*, to enforce the *Single Retreating Target* and *Single Exit*. As a matter of fact, if all the *retreating edges* in a *Region* point to the *head*, *head* might have more than two predecessors. This step re-enforces the *Two Predecessors* while transforming the *Region* in a DAG.

It starts by removing all the *retreating edges*, and substituting them with jumps to a newly created `continue` node. This naturally conveys the same semantic, that a retreating edge jumps to the *head* to start another iteration of the loop.

Then, all the edges jumping out of the region to the single successor are substituted them with jumps to a newly created `break` node. This also conveys the same semantic, that an exit edge jumps straight out of the loop to its single successor.

Figure 8 shows an example of these transformations.

Collapsing *Regions*. At this point, a *Region* that has been transformed by all the previous steps of *Preprocessing* is finally a DAG. As mentioned at the beginning of Section 4.4.2, only one *Region*

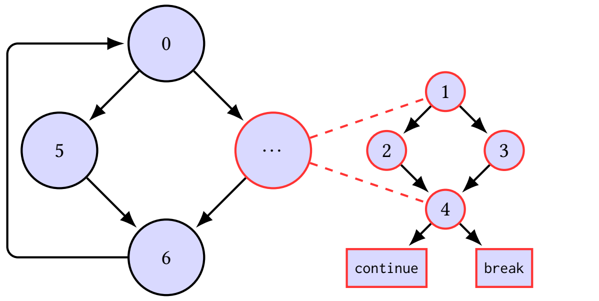


Figure 9: Collapsing nested DAG *Region*. The *Region* with red nodes on the left (composed of 1,2,3,4,`break`, and `continue`) can be collapsed in a virtual node from the point of view of its parent.

at a time is turned into a DAG, working on the *Region Tree* from the leaves to the root. After a *Region* has been transformed into a DAG, this step collapses it into a single virtual node in its parent's representation.

This is possible since each DAG *Region* has a *Single Entry* (part of the DAG) and a *Single Successor* (not part of the DAG). *Retreating edges* have been removed from the DAG, and substituted by `continue` nodes, that represent jumps to the *Single Entry*. Paths that exit from the DAG have been substituted with `break` nodes jumping to the *Single Successor*.

Hence in the parent's representation, a DAG *Region* is collapsed into a single virtual node V as follows. Given a *Region* P and a nested DAG *Region* C with *Single Entry* $E \in C$ and *Single Successor* $S \in P \setminus C$.

First, all the nodes in C are moved into the virtual node V_C.

Then, each edge $e = \langle X,E \rangle$ jumping from $P \setminus C$ to E is substituted with an edge $e_{V_C} = \langle X, V_C \rangle$. These represent all the entry paths to V_C (hence to the collapsed *Region* R), since the *Diamond Shape* property guarantees that there are no edges in the form $\langle X,Y \rangle$ with $X \in P \setminus C$, $Y \in C$ and $Y \neq E$. From a semantic standpoint, every new edge e_v jumps from X to the head of the *Region* C collapsed into V_C.

Finally, a new edge $e_S = \langle V_C, S \rangle$ is added to represent the fact that `break` nodes inside *Region* C collapsed into V_C can jump straight to the successor S.

This step concludes the collapsing of a single *Region*. An example can be seen in Figure 9.

Once all the children of a *Region* have been collapsed, the *Region* can be processed, until all the *Regions* in the tree become DAGs. These DAGs contain, among others, virtual nodes that represent nested collapsed DAG *Regions*. The *Region Tree* is now ready to be processed by the *Combing* stage.

4.5 Combing

This is the core of the *Control Flow Combing* algorithm. It enforces the *Diamond Shape* property, on all the DAGs in the *Region Tree*.

Enforcing this property reshapes the DAG so that it is only composed by nested *diamond-shaped* regions. These regions have only a single entry and a single exit node. They have no branches that jump directly in the middle of the region or jumping out from the middle of the region. All the paths incoming into a diamond-shaped region pass by the entry, and all the paths outgoing from the region pass by the exit. A simple example is visible in Figure 10.a. *Diamond-shaped* regions are easily convertible to C `if-else` constructs, with *then* and *else* branches, and with a single common successor that is the code emitted in C after both *then* and *else*.

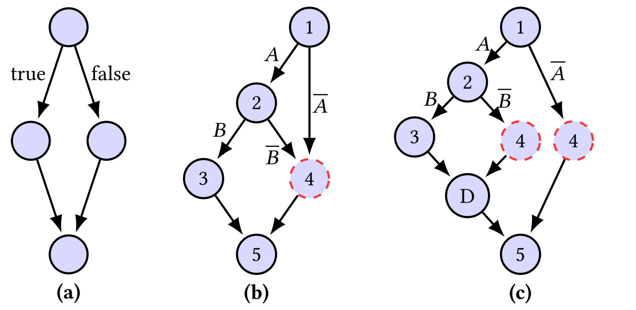


Figure 10: (a) A *diamond* shaped region. (b) A region which is not diamond-shaped. The arc between 1 and 4 breaks the assumption of not having edges incoming from outside the region. (c) The same region after the *Combing* has two nested diamond-shaped regions. D is a dummy node, i.e., an empty node useful only to highlight the diamond-shape.

Informally, the key idea of the *Combing* step, is to take all the regions that are not *diamond-shaped* (as the one in Figure 10.b) and restructure them to be *diamond-shaped* (as the one in Figure 10.c). In order to achieve this goal, it is necessary to duplicate some nodes in the graph. Node duplication can increase the size of the final generated C code. However, we deem that this increases clarity since it disentangles complex overlapping paths in the control flow, linearizing them and making them easier to reason about for an analyst, that can consider them one at a time. Moreover, in most cases the duplicated nodes introduced by the *Combing* can be deduplicated by the *Matching* stage, that uses them to emit idiomatic C code such as short-circuited `if`s as explained in Section 4.6.2.

The Combing Algorithm. As all previous steps, *Combing* is done on a single *Region* at a time. Thanks to the previous steps, *Regions* at this point are DAGs. These two properties greatly reduce the complexity, thanks to the shift of the problem from a global to a local perspective, and since DAGs are acyclic.

For each *Region* DAG the comb works as follows. *First*, it collects all the conditional nodes on the DAG. The *Diamond Shape* property states that every conditional node must dominate all the nodes between itself and its immediate post-dominator. Hence, for each of these nodes, it identifies the immediate post-dominator. This is always possible since the DAG has a single exit, thanks to the *sink* node injected at the beginning of *Preprocessing*. In this way, for each conditional node C, the algorithm identifies as set of nodes $\mathbb{D}(C)$ between C and its immediate post-dominator.

Second, for each node N in $\mathbb{D}(C)$ that is not dominated by C there is some incoming edge $e = \langle X, N \rangle$ such that source node X is not dominated by C. To enforce the *Diamond Shape* property, N should be dominated by C. Hence the node N is duplicated, creating a basic block node N' that contains the same instructions as N. Initially N' has no incoming nor outgoing edges. Then, for every outgoing edge $e_S = \langle N, S \rangle$ from N, an outgoing edge $e'_S = \langle N', S \rangle$ is created from N'. This ensures that N' jumps in the same places where N jumped, preserving the semantic of the program after N. Then, each incoming edge $e_P = \langle P, N \rangle$ into N such that C dominates P is *substituted* with an edge $e'_P = \langle P, N' \rangle$ incoming into N'. This means that after this transformation the node N' is dominated by C, and the node N is not reachable from C anymore. See Figure 10.b and Figure 10.c for an example of this transformation.

Basically, the underlying idea is to group the incoming edges in node N in two sets: one composed by the edges dominated by conditional node C, that will be moved to node N', and the other one composed by the edges not dominated by C, that will remain attached to node N.

This is sufficient to enforce the *Diamond Shape* property for C and N, but there might be other nodes in $\mathbb{D}(C)$. Repeating this on each $N \in \mathbb{D}(C)$ fully enforces the *Diamond Shape* property for the conditional node C. In turn, repeating the process in post-order on all conditional nodes in the DAG enforces the property on the whole *Region*. Notice that the process either never touches a node N (since it already fulfills the *Diamond Shape* property for all the conditional nodes from which it is reachable) or it splits the incoming edges of N into two sets.

At the end of the procedure, the *Region* DAG fulfills the *Diamond Shape* property and is said to be *combed*.

Note that, as shown in Figure 10, the *Combing* can insert dummy nodes (i.e., empty nodes) to reinstate the *two predecessor* property, and highlight the diamond-shape.

Improved Combing Algorithm: Untangling Return Paths. The *Combing Algorithm* as described above still has a drawback in some common cases: it duplicates code very aggressively which can lead to a big increase in code size if not controlled.

Consider Figure 11. The source code in the figure is very simple, and represents a pretty common case where some checks are performed on the arguments (`A` and `B`), a complex computation composes the body of the function (`C`), and some final error check is performed going on.

As we can see in Figure 11(a), one of the typical optimizations performed by compilers even at lower optimization levels is to coalesce all the `return`s in a single node (`R`). This is intended to reduce code size in the binaries, but from an analysis standpoint it "entangles" different execution paths that were originally separate (the two `return` statements at lines 4 and 10).

If the vanilla *Combing Algorithm* is applied on the graph in Figure 11(a), both nodes `C` and `R` would be duplicated, like shown in Figure 11(b). This would be very detrimental, because it would end up duplicating the whole bulk of the computation (`C`), unnecessarily inflating the size of the decompiled source code.

In order to cope with this cases we devised an improved combing algorithm: the *Untangling Algorithm*. The improved version of the algorithm, the *Untangling* is focused on handling these cases, and is performed just before the vanilla *Combing Algorithm*. After the *Untangling*, the *Combing* is executed on the untangled graphs, so that it can iron out all the situations left behind from the *Untangling* because they were not beneficial to untangle.

The *Untangling* is applied on each conditional node in a *Region* DAG, and only if beneficial. Its benefits are evaluated with an heuristic that determines, for each conditional node, if the duplication induced by untangling the return path is significantly lower than the duplication that the *Combing* pass would introduce if the *Untangling* is not performed. To do this, the heuristic assigns a weight to each node in the graph, to evaluate the consequences of applying the *Untangling* compared to the vanilla *Combing*. The weight of each node is proportional to the number of instructions each node

```
if (arg0) {    // A
  fun_call(); // B
  if (arg1)    // B
    return;    // R
}
// complex // C
// code    // C
// here    // C
if (err()) // C
  return; // R
```

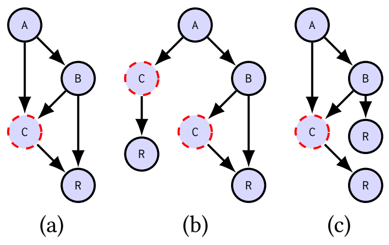


Figure 11: Situation in which the baseline *Combing* would be very costly in terms of duplicated code size. The graph in (a) is the CFG of the snippet of code on the left. The red dashed node (`C`) represent a big and messy portion of the CFG that would greatly increase code size if duplicated. With the baseline *Combing* both `C` and `R` would be duplicated, like shown in (b). With the *Untangling*, only `R` is duplicated instead, like shown in (c).

contains, and for collapsed *Regions* this number if computed cumulatively on all the nodes they contain. If, according to these weights, *Untangling* would duplicate more code than vanilla *Combing*, the graph is not untangled and only combed.

Whenever triggered on a conditional node *N* (`B` in Figure 11), the *Untangling* duplicates all the blocks from the post-dominator of *N* to the exit of the graph (only `R` in Figure 11, but potentially any other node after `R`). This transformation allows the *Combing* step to keep duplication under control.

Going back to Figure 11, we can see in Figure 11(c) how the *Untangling* would transform the graph. Only `R` is duplicated, saving a huge amount of unnecessary duplication if `C` is big. This is also an example where, after the *Untangling*, the plain *Combing Algorithm* does not have anything to do, because all nodes in Figure 11(c) are dominated by all the conditional nodes from which they are reachable. This means that the *Diamond Shape* property already holds after *Untangling*, saving the work that would have been necessary to comb the graph.

Finally, notice how the graph in Figure 11(c) is much more structurally close to the original source code than what plain *Combing* would obtain, i.e. Figure 11(b).

4.6 Matching C Constructs

This phase builds the initial Abstract Syntax Tree (AST) representation of each of the combed *Regions*, and then manipulates it to emit idiomatic C code.

4.6.1 *Building the AST.*

Thanks to the *Preprocessing* and *Combing* stages and all the enforced properties, building an AST is straightforward. The *Two Successors* rule ensures that each conditional node can be emitted as an `if`, and since all the DAGs are diamond-shaped regions, the DAG naturally represents a program with perfectly nested scopes (each diamond-shaped part represents a scope). Moreover, all *retreating edges* have already been removed and converted to `break` and `continue` nodes.

All these properties imply that the dominator tree of each DAG *Region* is a tree where each node can have at most three children. Exploiting this property, the algorithm works on the dominator tree (from root to leaves) to emit the AST. If a node *A* in the dominator tree has only a single child *B*, *A* and *B* are emitted as subsequent statements in a single scope in C. If a node *A* in the dominator tree has two or three children, then *A* is an `if` statement. Depending on how *A* is connected to its children, they can represent the *then* branch of *A*, the *else* branch of *A*, and the code that is emitted in the AST after both the *then* and the *else*. This allows to represent all the conditional nodes as well-structured `if` constructs.

A special treatment is reserved to nodes in a DAG *Region* that represent another nested DAG *Region*, that was collapsed by the *Collapsing Regions* step. Whenever one of such nodes is encountered, it is emitted in the AST as a `while(1) {...}` construct. The AST representing the body of the loop is then generated iteratively from the DAG of the collapsed *Region*. In general, this representation is not optimal for any loop, but it's only preliminary AST form that will be made more idiomatic as described in the next section.

4.6.2 *Matching Idiomatic C Constructs.*

The preliminary AST is now post-processed to match idiomatic C constructs, striving to emit even more readable code, while, at the same time, reducing the duplication introduced by *Combing*, when this is possible without sacrificing readability.

This post-processing is modular and extensible. We only report some basic matching steps leading to significant improvements with a reduced effort. Additional matching criteria can be devised and added to the pipeline, to emit even better code.

Each matching criterion listed in the following is basically structured as a top-down visit' on the AST, which recognizes certain patterns and transforms the AST to more idiomatic, but semantically equivalent, C constructs.

Short-Circuit Reduction. This criterion recognizes and reconstructs short-circuited `if` statements in C. In fact, the *Combing* step breaks these constructs as shown in Figure 10.c. This matching criterion reverts that choice when possible, allowing the *Combing* to handle general situations, while also emitting idiomatic short-circuited `if`s whenever possible.

Figure 10.c shows two nested `if` statements that have the same duplicated node in their `else` branches. This criterion matches that pattern. Whenever two nested `if` nodes on the AST have the same code on one of their branches, they are transformed into a single `if` node, short-circuiting their conditions with the appropriate combination of `&&`, `||`, and `!` operators.

Note that the previous works [22] did not perform short-circuited `if` matching, often leading to suboptimal results.

Switch Reconstruction. This criterion recognizes and builds `switch` statements. As mentioned in Section 4.4.1, to enforce the *Two Successors* property, `switch`es are decomposed in nested `if`s in the preprocessing phase of the *Restructuring*.

This criterion looks in the AST for nested `if`s whose conditions compare a variable for equality with different constants. Matched sequences of `if`s are transformed into `switch`es.

Loop Promotion. Similarly to what is done in Yakdan *et al.*[22], this criterion manipulates loops, initially emitted as `while(1) {...}`, to transform them into more idiomatic loops, with complex exit conditions and various shapes such as `while(...) {...}` and `do {...} while(...)`.

To match `while` loops, the AST is scanned looking for loops whose body starts with a statement in the form of `if(`*X*`) break;`. Any such cycle can be converted into a `while(!`*X*`) {...}`, leaving the rest of the loop body untouched. To match `do-while` loops, instead, the AST is scanned looking for loops whose body's final statement is in the form `if(X) continue; else break;`. These loops are transformed into `do {...} while(X)`. The same is done for loops where the `continue` and `break` statements are inverted, simply negating the condition.

5 EXPERIMENTAL RESULTS

This section evaluates the proposed approach. Section 5.1 describes the experimental setup, while Section 5.2 compares our implementation with state-of-the-art commercial and open-source decompilers.

5.1 Experimental Setup

To evaluate the performance of the Control Flow Combing described in the previous section, the algorithm has been implemented on top of the `rev.ng` static binary analysis framework [8, 9], based on QEMU [3] and LLVM [16]. `rev.ng` is capable of generating CFGs from binary programs for various CPU architectures. The *Preprocessing* and *Combing* stages of the algorithm has been implemented on top of the LLVM IR. After these phases, the *Matching* stage has been implemented on a simple custom AST for C, that is then translated into the AST employed by clang (LLVM's C/C++ compiler) and finally serialized to C in textual form.

The resulting decompiler is called `revng-c`. The quality of the code generated by `revng-c` is compared with two other well-known decompilers: IDA Pro's Hex-Rays Decompiler, the leading commercial decompiler developed by Hex-Rays [13], and Ghidra, developed by the National Security Agency (NSA) of the USA for internal use and recently open-sourced [2].

For a more thorough comparison, we tried to reach the authors of two other recent academic contributions in the control flow restructuring research area, namely [6] and [22]. Unfortunately, the authors of Brumley *et al.* [6] were not able to retrieve the artifacts to reproduce the results, while the main authors of DREAM [22] have left academia to focus on different topics, and therefore were not able to answer our inquiry. Given that DREAM [22] is the only other approach to generate `goto`-free C code, it would have been the perfect candidate to compare our approach with. This comparison would have enabled an evaluation of the main novelty of our approach: allowing duplication of code in order to reduce the cyclomatic complexity of the decompiled code, which is a measure of the mental load required to an analyst to understand the program. DREAM tries not to resort to duplication, while we accept small to moderate duplication because it reduces the cyclomatic complexity of the generated code. Lacking reproducible results to compare directly with DREAM, we decided to focus on a restricted number of case-studies, that show how the code generated by `revng-c` compares with their results. Given the limited reach this manual comparison with DREAM, we have left it in Appendix B.

The remainder of this section provides comparisons between `revng-c`, Ghidra, and Hex-Rays Decompiler. For these decompilers, the quality of the decompiled code was evaluated on the GNU Coreutils. These are the basic file, shell and text manipulation command line utilities of the GNU operating system. These benchmarks have been used in the past in related works on control flow restructuring [6, 22], since they implement a large set of utilities, hence producing a wide range of real-world different CFGs that do not emerge in toy examples. For the benchmarks, the GNU Coreutils 9.29 have been compiled with GCC 4.9.3 targeting the x86-64 architecture, without debug symbols and dynamic linking. We evaluated the performances of 4 optimizations levels, `O0`, `O1`, `O2` and `O3`.

At this point, all the generated binaries have been decompiled with all the decompilers, to generate C code. It is worth nothing that, on the basis of the data collected during our evaluation, `revng-c` is the only decompiler that produces valid C code as output. The decompiled code generated by Hex-rays Decompiler and Ghidra cannot be parsed as-is by a standard-conforming C parser. In order to do this, which was a requirement to collect our evaluation metrics on the decompiled code, we had to perform ad-hoc changes on the decompiled sources, such as declaring missing variables and types, correcting the number of parameters in function calls, and others.

To ensure a fair comparison, with the help of IDAPython for the Hex-Rays Decompiler and Java scripts for Ghidra, we extracted some information on the decompiled functions, such as their entry point and their size. We then proceeded to compare only the functions for which the three tools gave identical information about entry point and size. Later, in Table 1, we report the percentage of functions which matched in dimension, and that we used in our evaluation.

There is another aspect to keep in mind about using Coreutils as benchmarks. All these programs share a core library, called `gnulib`, whose code is statically linked with all binaries. This means that the functions in `gnulib` are duplicated many times. This problem has already been pointed out in the past, by the authors of DREAM [22], who also designed a strategy to overcome it. The idea is simple: all the decompiled functions across all Coreutils need to be deduplicated before the final comparison, to avoid overrepresenting duplicated functions. We adopted the same strategy for the comparison of our results.

5.2 Evaluation of the Results

The quality of the generated code has been measured according to the two following metrics.

gotos. The number of the emitted `goto` statements. `goto` statements are very detrimental to the readability of the decompiled code since they can arbitrarily divert the control flow, and keeping track of the execution becomes significantly more difficult [10].

Cyclomatic Complexity. The increment in cyclomatic complexity [18] of the decompiled code, using the one of the original code as baseline. This measures the mental effort required to understand the decompiled code.

We evaluate the code generated by the three decompilers from binaries with different optimization levels according to these metrics. The evaluation is limited to the functions that all the decompilers were able to correctly identify. The results are reported in Table 1. By construction, `revng-c` zeros out the `gotos` metric, generating 0 `gotos`, over the entire GNU Coreutils suite. We can also see how `revng-c` generates decompiled code with a reduced cyclomatic complexity with respect to the Hex-Ray Decompiler and Ghidra. Note that the cyclomatic complexity of decompiled code of the tools is expressed with respect to this baseline. In fact, we assume that this complexity is intrinsic in the code, and the objective of the decompilation is to introduce as little additional complexity as

	-O0			-O1			-O2			-O3		
	revng-c	IDA	Ghidra	revng-c	IDA	Ghidra	revng-c	IDA	Ghidra	revng-c	IDA	Ghidra
Cyclomatic Complexity	+11%	+12%	+16%	+13%	+17%	+20%	+36%	+60%	+72%	+78%	+86%	+94%
Gotos	0	1010	1370	0	2370	2622	0	2082	2062	0	2119	2282
Matched functions		93%			91%			89%			81%	

Table 1: Comparison between revng-c, IDA, and Ghidra. Results are aggregated for the optimization level that was used to obtain the binaries that were then decompiled (-O0, -O1, -O2, -O3). For each optimization level, the first row shows percentage of functions that were matched by all the decompilers (percentage of the binary code size). The second and third row show respectively the number of gotos produced by each decompiler, and the additional cyclomatic complexity introduce by the decompilation process with respect to the baseline cyclomatic complexity of the original source code.

possible. If we observe the -O2 optimization level, the one typically adopted in release builds, we can notice that `revng-c` is able to reduce the additional cyclomatic complexity by 40% with respect to IDA, and by almost 50% with respect to Ghidra.

For what concerns the metrics for the Hex-Rays Decompiler and Ghidra, we can see that Ghidra performs slightly better in terms of `goto` statements emitted, emitting less `gotos` when compared to the Hex-Rays Decompiler. Overall, as previously stated, we think that these metrics shows that the two decompilers adopt a similar approach to decompilation.

In Table 2 we provide an overview of the increase in terms of size of the decompiled code due to the duplication introduce by our approach. In Figure 12 we also show an estimate of the probability distribution function (using the KDE method) of the increase in size for all the optimizations levels. This metric, has not been computed for the other tools, since they do not introduce duplication.

Note also that the effects of duplication could be significantly mitigated by performing regular optimizations on the generated code, such as dead code elimination. In fact, the optimizer might be able to prove that, for instance, part of the code duplicated outside of a loop due to the outlining of the first iteration will never be executed and can therefore be dropped. However, due to timing constraints, we have not been able to assess the impact of such optimizations.

We also produced a pair of heat maps Figure 13 that helps visualizing how the relationship between duplication and decrement in cyclomatic complexity evolves. We plotted the values for the -O2 optimization level, comparing with both Hex-Rays and Ghidra. In particular, apart from the bright spots in correspondence of a low duplication level which are positive, we can see some reddish clouds towards the center of the heat maps, which represents a class of functions for which the duplication is significant, but for which the cyclomatic complexity is reduced with respect to IDA and Ghidra. This represent the fact that even when a cost in terms of duplication is payed, we have a gain in terms of reduction of cyclomatic complexity.

6 CONCLUSION

In this work we presented a novel approach to control flow restructuring and to decompilation, by introducing new techniques for transforming any given CFG into a DAG form, which we called *Preprocessing*, to which we later apply our *Combing* algorithm. Thanks to *Combing*, we are able to build a C AST from the input code, which is then transformed by the *Matching* phase to emit idiomatic C. We implemented our solution on top of the `rev.ng` framework, building a decompiler tool called `revng-c`. In the evaluation, we performed compared our results against both academic and commercial state-of-the-art decompilers.

The experimental results show that our solution is able to avoid the emission of `goto` statements, which is an improvement over the Hex-Rays Decompiler and Ghidra, but at the same time does not resorts to predicated execution, which on the other hand affects DREAM. In future work, we will to improve the quality of the decompiled code by focusing on the recovery of more idiomatic C constructs. This type of work will be greatly simplified by the already modular nature of the *Matching* phase of our algorithm.

	-O0	-O1	-O2	-O3
Goto	1.07×	1.10×	1.15×	1.32×
No-Goto	1.04×	1.08×	1.12×	1.25×

Table 2: Size increment metrics (over the original size) for the functions over different optimization levels. For each optimization level, we also provide the duplication factor metric computed only on functions which do not have goto statements in the original source code. While our algorithm is able to completely eliminate gotos in the decompiled code, we can see that in case we approach decompilation of code with gotos our duplication factor is penalized. Indeed, our algorithm makes the assumption that we are trying to decompile well-structured code, therefore gotos in the original source code make this assumption to fail.

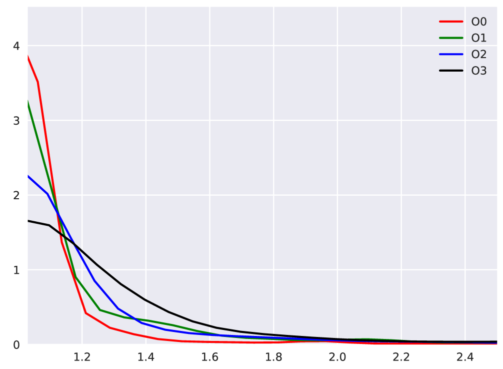
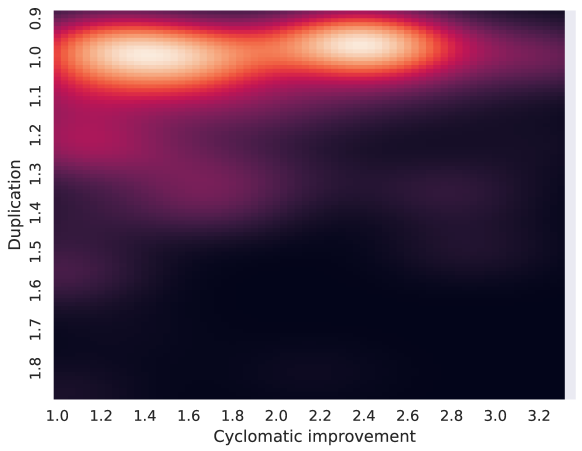
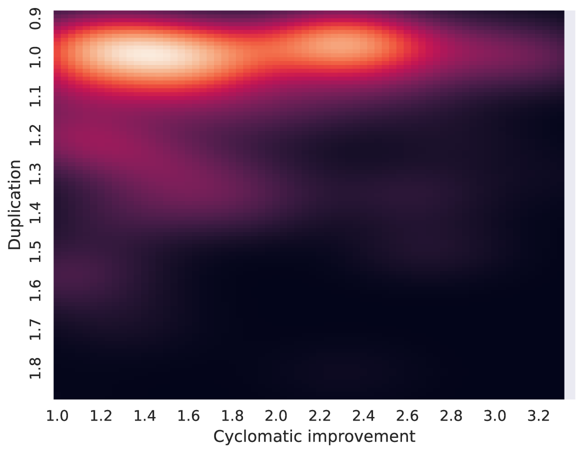


Figure 12: Plot showing the Probability distribution functions of the duplication introduced by revng-c. On the x axis, we have the amount of duplication introduced (measured in terms of code size increase over the original value). We can notice that as the optimization level increases, revng-c introduces a little bit more duplication in order to be able to be able to emit goto-free decompiled code.



(a) Cyclomatic complexity improvement w.r.t. IDA



(b) Cyclomatic complexity improvement w.r.t. Ghidra

Figure 13: Cyclomatic Complexity improvements of `revng-c` at O2. This heat maps helps us visualizing where the cyclomatic complexity improvement gain obtained by `revng-c` is introduced. The cyclomatic improvement is represented on the x axis, while the duplication factor introduced by `revng-c` is represented on the y axis (higher means less duplication). To color of a cell of the heat map is computed by performing a sum of bivariate distributions for each data point in our dataset (every function). This means that a cell will assume a brighter color as more data points showing values of duplication and decrease in cyclomatic complexity in its surrounding are present.

More in general, in the future we aim to further improve the quality of the decompiled code in other areas, such as arguments and return values detection and advanced type recognition techniques.

In addition, we are also considering the possibility for our decompiler to support the emission of some `goto` statements in a very limited and controlled setting, i.e., where they may be considered idiomatic and legitimate, e.g., in the *goto cleanup pattern*. The goal of this would be to trade the introduction of a `goto` in order to further reduce the duplication introduced by our combing algorithm.

We also want to address the verification of the semantics preservation of the control flow restructuring transformation we introduce. We deem this goal achievable thanks to the very nature of the `rev.ng` framework. The idea is to enforce back the modifications done by the control flow restructuring algorithm at the level of the LLVM IR lifted by `rev.ng`, and to use the recompilation features of the framework to prove the behavioural equivalence between the original binary and the one generated after the restructuring.

REFERENCES

[1] Hex-rays decompiler. https://www.hex-rays.com/products/decompiler/.
[2] National Security Agency. Ghidra. https://ghidra-sre.org/.
[3] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the FREENIX Track: 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*, 2005.
[4] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*. Springer, 2011.
[5] David Brumley, JongHyup Lee, Edward J Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, 2013.
[6] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013.
[7] Cristina Cifuentes. *Reverse compilation techniques*. Queensland University of Technology, Brisbane, 1994.
[8] Alessandro Di Federico and Giovanni Agosta. A jump-target identification method for multi-architecture static binary translation. In *Compliers, Architectures, and Sythesis of Embedded Systems (CASES), 2016 International Conference on*, 2016.
[9] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. rev. ng: a unified binary analysis framework to recover cfgs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction*, 2017.
[10] Edsger W Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3), 1968.
[11] Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. `rev.ng`: A multi-architecture framework for reverse engineering and vulnerability discovery. In *International Carnahan Conference on Security Technology, ICCST 2018, Montréal, Canada, October 22-25, 2018*. IEEE, 2018.
[12] Ilfak Guilfanov. Decompilers and beyond. *Black Hat USA*, 2008.
[13] Hex-Rays. Ida pro. https://www.hex-rays.com/products/ida/.
[14] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.
[15] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, 2004.
[16] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO 2004*.
[17] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1979.
[18] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4), Dec 1976.
[19] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *International Conference on Information Systems Security*. Springer, 2008.
[20] Khaled Yakdan. Dream code snippets. https://net.cs.uni-bonn.de/fileadmin/ag/martini/Staff/yakdan/code_snippets_ndss_2015.pdf.
[21] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016.
[22] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*, 2015.

A GRAPHS BASICS

This section introduces the fundamental concepts to understand the design of the *Control Flow Combing* algorithm, described in Section 4.

Graphs. A *directed graph* is a pair $G = \langle V,E \rangle$, where V is a set, and $E \subset V \times V$ is a set of pairs of element of V. Each $v \in V$ is called a *node*, and each $e = \langle v_1,v_2 \rangle$ is called an *edge*. Given e as defined above, v_1 is said to be a *predecessor* of v_2, while v_2 is said to be a *successor* of v_1. e is said to be *outgoing* from v_1 and *incoming* in v_2. v_1 is called the *source* of e and v_2 is called the *target* of v_1. A sequence of edges $e_1 = \langle v_{1,1},v_{1,2} \rangle,...,e_n = \langle v_{n,1},v_{n,2} \rangle$, is called a *path* if $\forall k = 1,...,n-1$ holds $v_{k,2} = v_{k+1,1}$.

Control Flow Graphs. A directed graph used to represent the control flow of a function in a program.

Each node of a CFG is called a *basic block* and represents a sequence of instructions in the program that are executed sequentially, without any branch with the exception of the last instruction in the basic block.

Each edge in a CFG is called a *branch.* A *branch* $b = \langle BB_1,BB_2 \rangle$ in a CFG, represents the notion that the execution of the program at the end of BB_1 might jump to the beginning of BB_2. Branches can be *conditional* or *unconditional.* A branch b is *unconditional* if it is always taken, independently of the specific values of conditions in the program at runtime. The source s of an unconditional branch b has no other outgoing edges. Conversely, a branch b is called conditional if it might be taken by the execution at runtime, depending on the runtime value of specific conditions in the program. The source of a conditional branch always has multiple outgoing edges and the conditions associated to each outgoing edge are always strictly exclusive.

Finally, a CFG representing a function has a special basic block, called *entry node*, *entry basic block*, or simply *entry*, that represents the point in the CFG where the execution of the function starts.

In the remainder of this work, where not specified otherwise, we will refer to CFGs.

Depth First Search. The concept of *Depth First Search* (DFS) [14] is very important for the rest of this work. Briefly, DFS is a search algorithm over a graph, which starts from a root node, and explores as far as possible following a branch (going deep in the graph, hence the name), before backtracking and following the other branches. When the algorithm explores a new node it is pushed on a stack, and, when the visit of the subtree starting in that node is completed, it is popped from the exploration stack. Such traversal can be used both for inducing an ordering on the nodes of a graph, and to calculate the set of *retreating edges* (informally, edges that jump back in the control flow). Directed graphs without retreating edges are called *Directed Acyclic Graphs* (DAG). A Depth First Search induces the following orderings of the nodes of a graph:

Preorder. Ordering of the nodes according to when they were first visited by the DFS and, therefore, pushed on the exploration stack.
Postorder. Ordering of the nodes according to when their subtree has been visited completely and, therefore, popped from the exploration stack.
Reverse Postorder. The reverse of postorder.

Dominance and Post-Dominance. Other two fundamental concepts in program analysis are *dominance* and *post-dominance.*

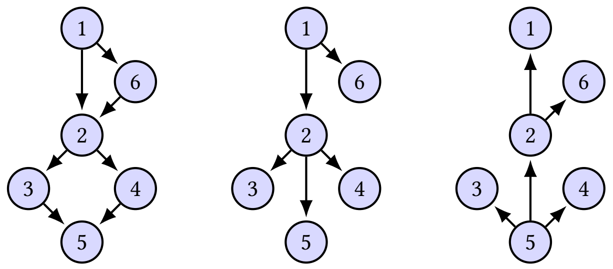


Figure 14: Left – An example graph. Node 1 is the *entry* and 5 is the *exit.* Middle – Dominator tree of the graph on the left. Edges in this tree go from immediate dominator to the dominated node. Right – Post-dominator tree of the graph on the left. Edges in this tree go from immediate post-dominator to post-dominated node.

Informally, *dominance* describes, given a node, which nodes have to be traversed on all paths from the *entry*, which must be present and unique, to that node. Given a graph and two nodes A and B, A *dominates* B iff every path from the *entry* to B contains A. A *properly dominates* B if A dominates B and $A \neq B$. A *immediately dominates* B if A properly dominates B and $A \neq B$ and it does not exist a node C such that A properly dominates C and C properly dominates B.

Conversely, *post-dominance* is related to which nodes must be traversed on paths from a given node to the *exit*, if this is present and unique. In cases where there is not a single *exit* node post-dominance is not defined. The node A *post-dominates* B if every path from B to the *exit* contains A. A *properly post-dominates* B if A post-dominates B and $A \neq B$. A *immediately post-dominates* B if A properly post-dominates B and $A \neq B$ and it does not exist a node C such that A properly post-dominates C and C properly post-dominates B.

Dominator and Post-Dominator Tree. The *dominator tree* (and the *post-dominator tree*) are a compact representation of the dominance (and post-dominance) relationship withing a graph. The *dominator tree* (DT) contains a node for each node of the input graph, and an edge from node A to node B iff A is the immediate dominator of B. The resulting graph is guaranteed to be a tree since each node except the *entry* node has a unique immediate dominator. As a consequence, the *entry* node is the root of the dominator tree. Whenever the *exit* of a graph is unique, it is possible to build an analogous data structure for the post-dominance relationship, called *post-dominator tree* (PDT). A well known and widely used algorithm for calculating a dominator tree is the *Lengauer-Tarjan's* algorithm, that has the peculiarity of having an almost linear complexity [17].

Examples of dominator and post-dominator tree for a CFG are represented in Figure 14.

```
void sub43E100(void *a1, int a2) {
...
  v4 = *(result + 0xc);
  if(a2 >= v4)
    v5 = *(result + 8) + v4;
  if(a2 < v5 && a2 >= v4)
    break;
  i++;
  result += 0x28;
...
}
```

Figure 15: Snippets which shows the reuse of the condition `a2 >= 4` on the same execution path

B CASE STUDIES

This section is devoted to a comparison of our solution with DREAM. We need a special section for this since the only artifacts of decompilation available from DREAM are the ones included in a whitepaper [20] cited in [22]. In this document, for every sample of code decompile by DREAM is present the corresponding code decompiled by Hex-Rays. Unfortunately, we were not able to recover the original functions (in terms of binary code) used for the evaluation of DREAM. This is due to the fact that the presented snippets belongs to malware samples for which a lot of different variants are available.

We observe that the provided Hex-Rays Decompiler decompiled source resembles very closely the original assembly representation (e.g., due to the abundance of `goto` statements). Therefore, in order to be able to compare our results with DREAM's, we decided to use the Hex-Rays Decompiler decompiled sources as a starting point for obtaining the CFG of the functions. Then, in turn, apply our algorithm, and obtain the `revng-c` decompiled sources.

To assess on a large scale how `revng-c` performed compared to DREAM, we collected the mentioned metrics on both the DREAM decompiled sources provided in the whitepaper, and on the sources produced by `revng-c`.

Table 3 presents the results we obtained. The higher cyclomatic complexity in code produced by DREAM is due to the predicated execution-like code. In fact, in this cases, the same condition will be employed multiple times as a state variable that enables or disables certain portions of the code. This approach forces the analyst to keep track of the state of the variables, increasing its mental load in a non-negligible way. As we can see, both DREAM and `revng-c` provide decompiled sources without `goto` statements, but DREAM presents reuse of the conditions as expected and informally explained throughout the paper. As a concrete example of conditional reuse, consider the snippet in Figure 15 (lines from 11 to 17 extracted from the snippet 1.5 in DREAM whitepaper [20]), we can see how the condition `a2 >= v4` is reused twice on the same execution path.

As an additional example, Figure 16 compares how a situation that DREAM (on the left of the listing) handles through predicated execution is handled with duplication in `revng-c` (on the right of the listing). The `revng-c` listing has been manually modified to reflect the same variable names used by DREAM. Also some optimizations in terms of code readability have been performed, but these changes do not concern the control flow, but are simple aesthetic

	DREAM	`revng-c`
Cridex4	9	5
ZeusP2P	9	5
SpyEye	19	15
OverlappingLoop	4	3

Table 3: This table presents the cyclomatic complexity of the code produced by DREAM and `revng-c`. As we can see, DREAM consistently presents higher figures compared to `revng-c` due to the high number situations in which conditions are reused multiple times.

improvements. We can see that, while DREAM uses a predicated execution approach, guarding the statement at line 11 with a complicated condition. On the other hand, `revng-c` duplicates some code, in this case the assignment, directly where the conditions to evaluate if the assignment needs to be performed are available, specifically at line 5. The idea is to *inline* the portion of code, paying a cost in terms of duplication, instead of deferring it, but paying a cost in terms of mental load necessary to understand when this assignment is actually executed. In the example, the cost is visible in the DREAM snippet as the convoluted condition of the `if` statement at lines 9 and 10. In this case, duplication also highlights immediately what value is assigned to `v2`, that is the return value of the function.

In this section we illustrated why we think that predicated execution is suboptimal in terms of mental load for the analyst that reads the decompiled code. The point is that in presence of predicated execution, different parts of the code in different conditional constructs are executed on the basis of the state of the conditional variable. This causes a mix of control flow and information on the state of the conditional variable, which causes the heavy mental load. Of course this is something that can be present in C code in principle, but the predicated execution introduced by DREAM push this to the limit where it becomes an impediment for the analyst. We have done this by highlighting in a couple of examples where this happens and how we approach instead the decompilation of the same snippet of code, and by showing that the predicated execution approach increases the cyclomatic complexity of the code.

During the design of the validation of our work, we also evaluated the possibility of conducting an user study to evaluate the performance of different decompilers, as done in [21]. However, we deemed that such kind of user study is really helpful to evaluate the overall performance of a decompiler tool only once aspects orthogonal to what presented in this paper are developed, such as the identification of library functions and type identification techniques. In this paper instead, we focused on the control flow recovery portion of the decompilation task, and this led us to set up the experimental evaluation in the way we did. Anyway, we do not exclude to conduct an user study, once the other mentioned aspects of the decompiler have matured.

```
if (!cond1 && !cond2) {
  v4 = sub4634E2(a1+a2*4, a7, 0, ...);
  v2 = v4;
  if (v4) {
    cond3 = v4 == -4;
    ...
  }
}
if((cond1 || v4) && (cond1 || !cond2)
   && (cond3 || !v3) && (!cond1 || !v3))
  v2 = -3;
if(!HeapValidate(GetProcessHeap(), 0, lpMem))
  return v2;
HeapFree(GetProcessHeap(), 0, lpMem);
return v2;
```

```
if (var_4) {
  v4 = sub4634E2(a1+a2*4, a7, 0, ...);
  if (v4) {
    if (v4 == -4) {
      v2 = -3;
    } else {
      v2 = v4
    }
    if(!HeapValidate(GetProcessHeap(), 0, lpMem))
      return v2;
    HeapFree(GetProcessHeap(), 0, lpMem);
  }
  ...
}
return v2;
```

Figure 16: Side by side SpyeEye listings of the decompiled source by DREAM (on the left) and `revng-c` (on the right).