No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations

Khaled Yakdan*, Sebastian Eschweiler†, Elmar Gerhards-Padilla†, Matthew Smith*
*University of Bonn, Germany
{*yakdan, smith*}*@cs.uni-bonn.de*
†Fraunhofer FKIE, Germany
{*sebastian.eschweiler, elmar.gerhards-padilla*}*@fkie.fraunhofer.de*

***Abstract*—Decompilation is important for many security applications; it facilitates the tedious task of manual malware reverse engineering and enables the use of source-based security tools on binary code. This includes tools to find vulnerabilities, discover bugs, and perform taint tracking. Recovering high-level control constructs is essential for decompilation in order to produce structured code that is suitable for human analysts and source-based program analysis techniques. State-of-the-art decompilers rely on structural analysis, a pattern-matching approach over the control flow graph, to recover control constructs from binary code. Whenever no match is found, they generate `goto` statements and thus produce unstructured decompiled output. Those statements are problematic because they make decompiled code harder to understand and less suitable for program analysis.**

In this paper, we present DREAM, the first decompiler to offer a `goto`-free output. DREAM uses a novel *pattern-independent* control-flow structuring algorithm that can recover all control constructs in binary programs and produce structured decompiled code without any `goto` statement. We also present *semantics-preserving transformations* that can transform unstructured control flow graphs into structured graphs. We demonstrate the correctness of our algorithms and show that we outperform both the leading industry and academic decompilers: Hex-Rays and Phoenix. We use the GNU `coreutils` suite of utilities as a benchmark. Apart from reducing the number of `goto` statements to zero, DREAM also produced more compact code (less lines of code) for 72.7% of decompiled functions compared to Hex-Rays and 98.8% compared to Phoenix. We also present a comparison of Hex-Rays and DREAM when decompiling three samples from Cridex, ZeusP2P, and SpyEye malware families.

I. INTRODUCTION

Malicious software (*malware*) is one of the most serious threats to the Internet security today. The level of sophistication employed by current malware continues to evolve significantly. For example, modern botnets use advanced cryptography, complex communication and protocols to make reverse engineering harder. These security measures employed by malware authors are seriously hampering the efforts by computer security researchers and law enforcement [4, 32] to understand and take down botnets and other types of malware. Developing effective countermeasures and mitigation strategies requires a thorough understanding of functionality and actions performed by the malware. Although many automated malware analysis techniques have been developed, security analysts often have to resort to manual reverse engineering, which is difficult and time-consuming. Decompilers that can reliably generate high-level code are very important tools in the fight against malware: they speed up the reverse engineering process by enabling malware analysts to reason about the high-level form of code instead of its low-level assembly form.

Decompilation is not only beneficial for manual analysis, but also enables the application of a wealth of source-based security techniques in cases where only binary code is available. This includes techniques to discover bugs [5], apply taint tracking [10], or find vulnerabilities such as RICH [7], KINT [38], Chucky [42], Dowser [24], and the property graph approach [41]. These techniques benefit from the high-level abstractions available in source code and therefore are faster and more efficient than their binary-based counterparts. For example, the average runtime overhead for the source-based taint tracking system developed by Chang *et al.* [10] is 0.65% for server programs and 12.93% for compute-bound applications, whereas the overhead of Minemu, the fastest binary-based taint tracker, is between 150% and 300% [6].

One of the essential steps in decompilation is control-flow structuring, which is a process that recovers the high-level control constructs (e.g., `if-then-else` or `while` loops) from the program's control flow graph (CFG) and thus plays a vital role in creating code which is readable by humans. State-of-the-art decompilers such as Hex-Rays [22] and Phoenix [33] employ structural analysis [31, 34] (§II-A3) for this step. At a high level, structural analysis is a pattern-matching approach that tries to find high-level control constructs by matching regions in the CFG against a predefined set of region schemas. When no match is found, structural analysis must use `goto` statements to encode the control flow inside the region. As a result, it is very common for the decompiled code to contain many `goto` statements. For instance, the *de facto* industry standard decompiler Hex-Rays (version v2.0.0.140605) produces 1,571 `goto` statements for a peer-to-peer Zeus sample (MD5 hash 49305d949fd7a2ac778407ae42c4d2ba) that consists of 997 nontrivial functions (functions with more than one basic block). The decompiled malware code consists of 49,514 lines of code. Thus, on average it contains one `goto` statement for each 32 lines of code. This high number of `goto` statements makes the decompiled code less suitable for both manual and automated program analyses. Structured code is easier to understand [16] and helps scale program analysis [31]. The

NDSS '15, 8-11 February 2015, San Diego, CA, USA
 ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23185

research community has developed several enhancements to structural analysis to recover control-flow abstractions. One of the most recent and advanced academic tools is the Phoenix decompiler [33]. The focus of Phoenix and this line of research in general is on correctly recovering more control structure and reducing the number of `goto` statements in the decompiled code. While significant advances are being made, whenever no pattern match is found, `goto` statements must be used and this is hampering the time-critical analysis of malware. This motivated us to develop a new control-flow structuring algorithm that relies on the semantics of high-level control constructs rather than the shape of the corresponding flow graphs.

In this paper, we overcome the limitations of structural analysis and improve the state of the art by presenting a novel approach to control-flow structuring that is able to recover *all* high-level control constructs and produce structured code without a single `goto` statement. To the best of our knowledge, this is the first control-flow structuring algorithm to offer a completely `goto`-free output. The key intuition behind our approach is based on two observations: (1) high-level control constructs have a single entry point and a single successor point, and (2) the type and nesting of high-level control constructs are reflected by the logical conditions that determine when CFG nodes are reached. Given the above intuition, we propose a technique, called *pattern-independent* control flow structuring, that can structure any region satisfying the above criteria without any assumptions regarding its shape. In case of cyclic regions with multiple entries or multiple successors, we propose *semantics-preserving* transformations to transform those regions into semantically equivalent single-entry single-successor regions that can be structured by our pattern-independent approach.

We have implemented our algorithm in a decompiler called DREAM (Decompiler for Reverse Engineering and Analysis of Malware). Based on the implementation, we measure our results with respect to correctness and compare DREAM to two state-of-the-art decompilers: Phoenix and Hex-Rays.

In summary, we make the following contributions:

- We present a novel *pattern-independent* control-flow structuring algorithm to recover *all* high-level control structures from binary programs without using any `goto` statements. Our algorithm can structure arbitrary control flow graphs without relying on a predefined set of *region schemas* or patterns.
- We present new *semantics-preserving graph restructuring* techniques that transform unstructured CFGs into a semantically equivalent form that can be structured without `goto` statements.
- We implement DREAM, a decompiler containing both the *pattern-independent* control-flow structuring algorithm and the *semantics-preserving graph restructuring* techniques.
- We demonstrate the correctness of our control-flow structuring algorithm using the joern C/C++ code parser and the GNU `coreutils`.
- We evaluate DREAM against the Hex-Rays and Phoenix decompilers based on the `coreutils` benchmark.

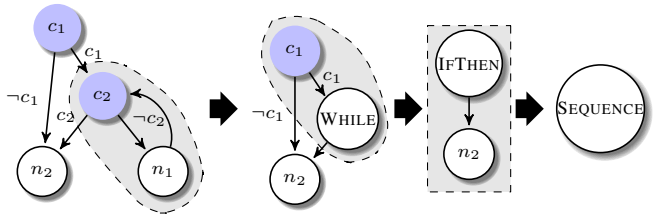


Fig. 2: Example of structural analysis.

- We use DREAM to decompile three malware samples from Cridex, ZeusP2P and SpyEye and compare the results with Hex-Rays.

II. BACKGROUND & PROBLEM DEFINITION

In this section, we introduce necessary background concepts, define the problem of control-flow structuring and present our running example.

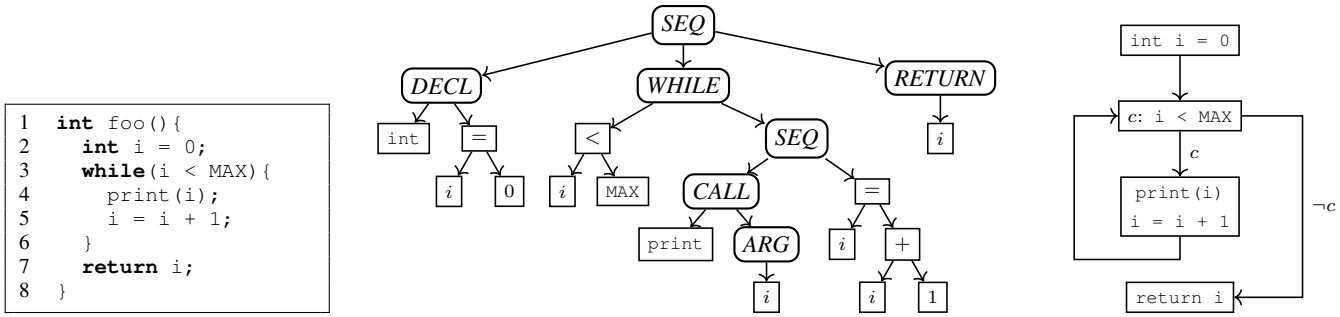
A. Background

We start by briefly discussing two classic representations of code used throughout the paper and provide a high-level overview of structural analysis. As a simple example illustrating the different representations, we consider the code sample shown in Figure 1a.

1) Abstract Syntax Tree (AST): Abstract syntax trees are ordered trees that represent the hierarchical syntactic structure of source code. In this tree, each interior node represents an *operator* (e.g., additions, assignments, or `if` statements). Each child of the node represents an *operand* of the operator (e.g., constants, identifiers, or nested operators). ASTs encode how statements and expressions are nested to produce a program. As an example, consider Figure 1b showing an abstract syntax tree for the code sample given in Figure 1a.

2) Control Flow Graph (CFG): A control flow graph of a program P is a directed graph $G = (N, E, n_h)$. Each node $n \in N$ represents a basic block, a sequence of statements that can be entered only at the beginning and exited only at the end. Header node $n_h \in N$ is P's entry. An edge $e = (n_s, n_t) \in E$ represents a possible control transfer from $n_s \in N$ to $n_t \in N$. A tag is assigned to each edge to represent the logical predicate that must be satisfied so that control is transferred along the edge. We distinguish between two types of nodes: *code nodes* represent basic blocks containing program statements executed as a unit, and *condition nodes* represent testing a condition based on which a control transfer is made. We also keep a mapping of tags to the corresponding logical expressions. Figure 1c shows the CFG for the code sample given in Figure 1a.

3) Structural Analysis: At a high level, the traditional approach of structural analysis relies on a predefined set of *patterns* or *region schemas* that describe the shape of high-level control structures (e.g., `while` loop, `if-then-else` construct). The algorithm iteratively visits all nodes of the CFG in post-order and locally compares subgraphs to its predefined patterns. When a match is found, the corresponding region is



(a) Exemplary code sample (b) Abstract Syntax Tree (c) Control Flow Graph

Fig. 1: Code representations.

collapsed to one node of corresponding type. If no match is found, `goto` statements are inserted to represent the control flow. In the literature, acyclic and cyclic subgraphs for which no match is found are called *proper* and *improper intervals,* respectively. For instance, Figure 2 shows the progression of structural analysis on a simple example from left to right. In the initial (leftmost) graph nodes n_1 and c_2 match the shape of a `while` loop. Therefore, the region is collapsed into one node that is labeled as a `while` region. The new node is then reduced with node c_1 into an `if-then` region and finally the resulting graph is reduced to a sequence. This series of reductions are used to represent the control flow as $\textbf{if}\,(c_1)\,\{\textbf{while}\,(\neg c_2)\,\{n_1\}\}\,;n_2$

B. Problem Definition

Given a program P in CFG form, the problem of *control-flow structuring* is to recover high-level, structured control constructs such as loops, `if-then` and `switch` constructs from the graph representation. An algorithm that solves the control-flow structuring problem is a program transformation function f_P that returns, for a program's control flow graph P_{CFG}, a semantically equivalent abstract syntax tree P_{AST}. Whenever f_P cannot find a high-level structured control construct it will resort to using `goto` statements. In the context of this paper, we denote code that does not use `goto` statements as structured code. The control-flow of P can be represented in several ways, i.e., several correct ASTs may exist. In its general form structural analysis can and usually does contain `goto` statements to represent the control flow. Our goal is to achieve fully structured code, i.e., code without any `goto`. For this, we restrict the solution space to *structured solutions*. That is, all nodes $n \in P_{AST}$ representing control constructs must belong to the set of structured constructs shown in Table I. The table does not contain `for` loops since these are not needed at this stage of the process. `for` loops are recovered during our post-structuring optimization step to enhance readability (§VI).

C. Running Example

As an example illustrating a sample control flow graph and running throughout this paper, we consider the CFG shown in Figure 3. In this graph, code nodes are denoted by n_i where i is an integer. Code nodes are represented in white. Condition nodes are represented in blue and labeled with the condition tested at that node. The example contains three regions that we use to illustrate different parts of our structuring algorithm. R_1 represents a loop that contains a `break` statement resulting in an exit from the middle of the loop to the successor node. R_2 is a proper interval (also called abnormal selection path). In this region, the subgraph headed at b_1 cannot be structured as an `if-then-else` region due to an abnormal exit caused by the edge (b_2, n_6). Similarly, the subgraph with the head at b_2 cannot be structured as `if-then-else` region due to an abnormal entry caused by the edge (n_4, n_5). Due to this, structural analysis represents at least one edge in this region as a `goto` statement. The third region, R_3, represents a loop with an unstructured condition, i.e., it cannot be structured by structural analysis. These three regions where chosen such that the difficulty for traditional structuring algorithms increases from R_1 to R_3. The right hand side of Figure 5 shows how the structuring algorithm of Hex-Rays structures this CFG. For comparison, the left hand side shows how the algorithms developed over the course of this paper structure the CFG. As can be seen for the three regions, the traditional approach produces `goto` statements and thus impacts readability. Even in this toy example a non-negligible amount of work needs to be invested to extract the semantics of region R_3. In contrast, using our approach, the entire region is represented by a single `while` loop with a single clear and understandable continuation condition.

TABLE I: AST nodes that represent high-level control constructs

AST Node	Description
$Seq\,[n_i]^{i\in 1..k}$	*Sequence* of nodes $[n_1,\ldots,n_k]$ executed in order. Sequences can also be represented as $Seq\,[n_1,\ldots,n_k]$.
$Cond\,[c,n_t,n_f]$	*If* construct with a condition c, a true branch n_t and a false branch n_f. It may have only one branch.
$Loop\,[\tau,c,n_b]$	*Loop* of type $\tau \in \{\tau_{\textbf{while}},\tau_{\textbf{dowhile}},\tau_{\textbf{endless}}\}$ with continuation condition c and body n_b.
$Switch\,[v,\mathcal{C},n_d]$	*Switch* construct consisting of a variable v, a list of cases $\mathcal{C}=[(V_1,n_1),\ldots,(V_k,n_k)]$, and a default node n_d. Each case (V_i,n_i) represents a node n_i that is executed when $v \in V_i$

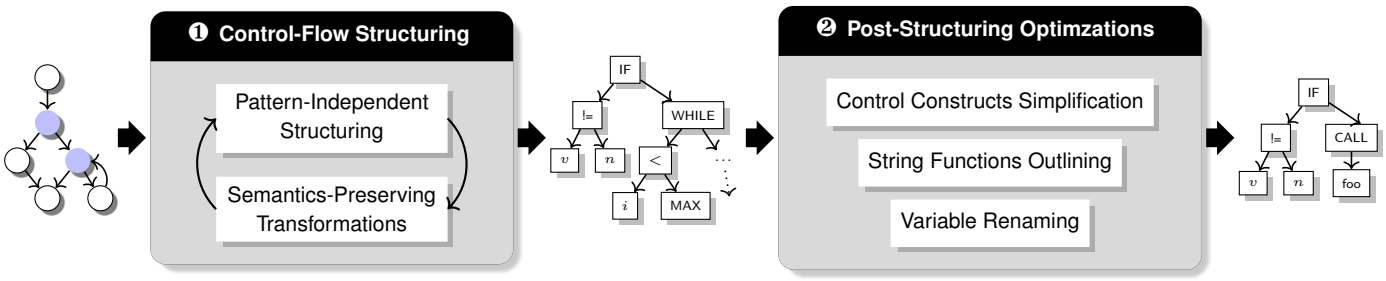


Fig. 4: Architecture of our approach. We first compute the abstract syntax tree using pattern-independent structuring and semantics-preserving transformations (❶). Then, we simplify the computed AST to improve readability (❷).

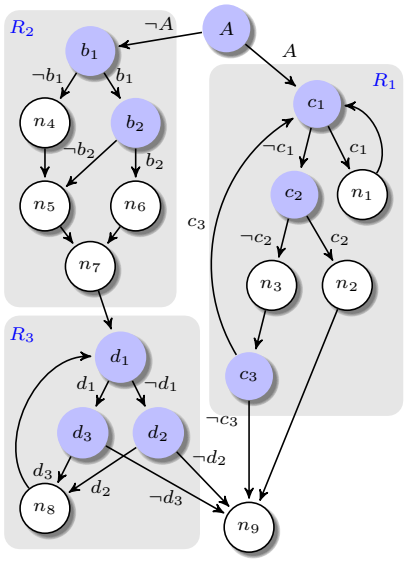


Fig. 3: Running example. Sample CFG that contains three regions: a `while` loop with a break statement (R_1), a proper interval (R_2), and a loop with unstructured condition (R_3).

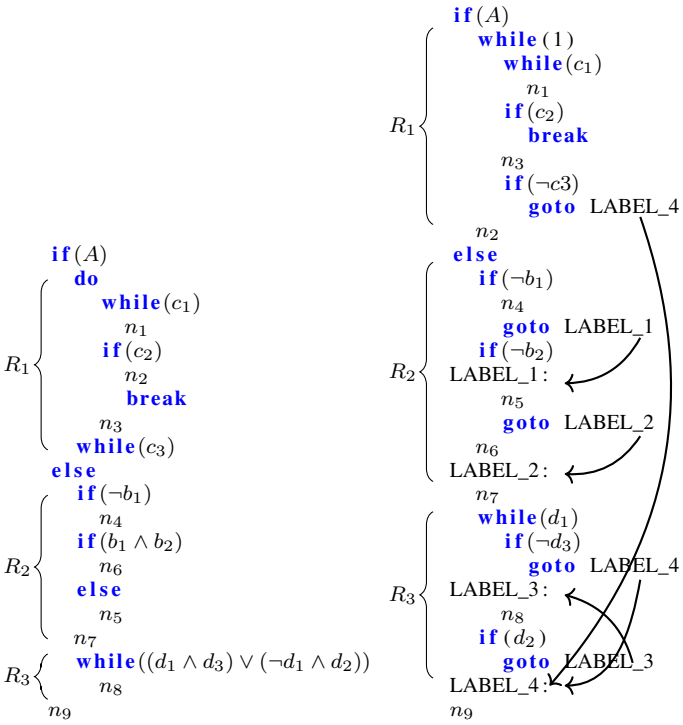


Fig. 5: Decompiled code generated by DREAM (left) and by Hex-Rays (right). The arrows represent the jumps realized by `goto` statements.

III. DREAM OVERVIEW

DREAM consists of several stages. First, the binary file is parsed and the code is disassembled. This stage builds the CFG for all binary functions and transforms the disassembled code into DREAM's intermediate representation (IR). There are several disassemblers and binary analysis frameworks that already implement this step. We use IDA Pro [2]. Should the binary be obfuscated tools such as [27] and [43] can be used to extract the binary code.

The second stage performs several data-flow analyses including constant propagation and dead code elimination. The third stage infers the types of variables. Our implementation relies on the concepts employed by TIE [29]. The forth and last phase is control-flow structuring that recovers high-level control constructs from the CFG representation. The first three phases rely on existing work and therefore will not be covered in details in this paper. The remainder of this paper focuses on the novel aspects of our research concerning the control-flow structuring algorithm.

A high-level overview of our approach is presented in Figure 4. It comprises two phases: *control-flow structuring*, and *post-structuring optimizations*. The first phase is our algorithm to recover control-flow abstractions and computes the corresponding AST. Next, we perform several optimization steps to improve readability.

Our control-flow structuring algorithm starts by performing a depth-first traversal (DFS) over the CFG to find *back edges* which identify cyclic regions. Then, it visits nodes in post-order and tries to structure the region headed by the visited node. Structuring a region is done by computing the AST of control flow inside the region and then reduce it into an *abstract* node. Post-order traversal guarantees that all descendants of a given node n are handled before n is visited.

When at node n, our algorithm proceeds as follows: if n is the head of an acyclic region, we compute the set of nodes dominated by n and structure the corresponding region if it has a single successor (§IV-B). If n is the head of a cyclic region, we compute loop nodes. If the corresponding region has multiple entry or successor nodes, we transform it into a semantically equivalent graph with a single entry and a single successor (§V) and structure the resulting region (§IV-C). The last iteration reduces the CFG to a single node with the program's AST.

Pattern-independent structuring. We use this approach to compute the AST of single-entry and single-successor regions in the CFG. The entry node is denoted as the region's header. Our approach to structuring acyclic regions proceeds as follows: first, we compute the lexical order in which code nodes should appear in the decompiled code. Then, for each node we compute the condition that determines when the node is reached from the region's header (§IV-A), denoted by *reaching condition*. In the second phase, we iteratively group nodes based on their reaching conditions and reachability relations into subsets that can be represented using `if` or `switch` constructs. In the case of cyclic regions, our algorithm first represents edges to the successor node by `break` statements. It then computes the AST of the loop body (acyclic region). In the third phase, the algorithm finds the loop type and condition by first assuming an endless loop and then reasoning about the whole structure. The intuition behind this approach is that any loop can be represented as endless loop with additional `break` statements. For example, a while loop **while** (c) {`body;`} can be represented by **while** (1) {**if** $(\neg c)$ {**break;**} `body;` }.

Semantics-preserving transformations. We transform cyclic regions with multiple entries or multiple successors into semantically equivalent single-entry single-successor regions. The key idea is to compute the unique condition `cond`(n) based on which the region is entered at or exited to a given node n, and then redirect corresponding edges into a unique header/successor where we add a series of checks that take control flow from the new header/successor to n if `cond`(n) is satisfied.

Post-structuring optimizations. After having recovered the control flow structure represented by the computed AST, we perform several optimization steps to improve readability. These optimizations include simplifying control constructs (e.g., transforming certain `while` loops into `for` loops), outlining common string functions, and giving meaningful names to variables based on the API calls.

IV. Pattern-Independent Control-Flow Structuring

In this section we describe our pattern-independent structuring algorithm to compute the AST of regions with a single entry (h) and single successor node, called region header and region successor. The first step necessary is to find the condition that determines when each node is reached from the header.

Algorithm 1: *Graph Slice*

Input : Graph $G = (N, E, h)$; source node n_s; sink node n_e
Output: $S_G(n_s, n_e)$

```
1  S_G ← ∅;
2  dfsStack ← {n_s};
3  while E has unexplored edges do
4  |  e := DFSNextEdge(G);
5  |  n_t := target(e);
6  |  if n_t is unvisited then
7  |  |  dfsStack.push(n_t);
8  |  |  if n_t = n_e then
9  |  |  |  AddPath(S_G, dfsStack)
10 |  |  end
11 |  else if n_t ∈ S_G ∧ n_t ∉ dfsStack
   |  then
12 |  |  AddPath(S_G, dfsStack)
13 |  end
14 |  RemoveVisitedNodes()
15 end
```

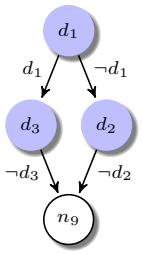


Fig. 6: $S_G(d_1, n_9)$ of running example

A. Reaching Condition

In this section, we discuss our algorithm to find the condition that takes the control flow from node n_s to node n_e, denoted by *reaching condition* $c_r(n_s, n_e)$. This step is essential for our pattern-independent structuring and guarantees the semantics-preserving property of our transformations (§V).

1) Graph Slice: We introduce the concept of the *graph slice* to compute the reaching condition between two nodes. We define the *graph slice* of graph $G(N, E, n_h)$ from a source node $n_s \in N$ to a sink node $n_e \in N$, denoted by $S_G(n_s, n_e)$, as the directed acyclic graph $G_s(N_s, E_s, n_s)$, where N_s is the set of nodes on simple paths from n_s to n_e in G and E_s is the set of edges on simple paths from n_s to n_e in G. We only consider simple paths since the existence of cycles on a path between two nodes does not affect the condition based on which one is reached from the other. Intuitively, we are only interested in the condition that causes control to leave the cycle and get closer to the target node. A path p that includes a cycle can be decomposed into two disjoint components: simple-path component p_s and cycle component p_c. The target node is reached if only p_s is followed (cycle is not executed) or if p_s and p_c are traversed (cycle is executed). Therefore, the condition represented by p is $\texttt{cond}(p) = \texttt{cond}(p_s) \vee [\texttt{cond}(p_s) \wedge \texttt{cond}(p_c)]$. The last logical expression can be rewritten as $\texttt{cond}(p_s) \wedge [1 \vee \texttt{cond}(p_c)]$ which finally evaluates to $\texttt{cond}(p_s)$.

Algorithm 1 computes the graph slice by performing depth-first traversal of the CFG starting from the source node. The slice is augmented whenever the traversal discovers a new simple path to the sink. The algorithm uses a stack data structure, denoted by `dfsStack`, to represent the currently explored simple path from the header node to the currently visited node. Nodes are pushed to `dfsStack` upon first-time visit (line 7) and popped when all their descendants have been discovered (line 14). In each iteration of edge exploration, the current path represented by `dfsStack` is added to the slice when traversal reaches the sink node (line 9) or when it discovers a simple path to a slice node (line 12). The last step

is justified by the fact that any slice node n has a simple path to the sink node. The path represented by `dfsStack` and the currently explored edge e is simple if the target node of e is not in `dfsStack`.

We extend Algorithm 1 to calculate the graph slice from a given node to a set of sink nodes. For this purpose, we first create a *virtual sink node* n_v, add edges from the sink set to n_v, compute $S_G(n_s, n_v)$, and finally remove n_v and its incoming edges. Figure 6 shows the computed graph slice between nodes d_1 and n_9 in our running example. The slice shows that n_9 is reached from d_1 if and only if the condition $(d_1 \wedge \neg d_3) \vee (\neg d_1 \wedge \neg d_2)$ is satisfied.

2) Deriving and Simplifying Conditions: After having computed the slice $S_G(n_s, n_e)$, the reaching conditions for all slice nodes can be computed by one traversal over the nodes in their topological order. This guarantees that all predecessors of a node n are handled before n. To compute the reaching condition of node n, we need the reaching conditions of its direct predecessors and the tags of incoming edges from these nodes. Specifically, we compute the reaching conditions using the formula:

$$c_r(n_s, n) = \bigvee_{v \in \texttt{Preds}(n)} (c_r(n_s, v) \wedge \tau(v, n))$$

where $\texttt{Preds}(n)$ returns the immediate predecessors of node n and $\tau(v, n)$ is the tag assigned to edge (v, n). Then, we simplify the logical expressions.

B. Structuring Acyclic Regions

The key idea behind our algorithm is that any directed acyclic graph has at least one topological ordering defined by its reverse postordering [14, p. 614]. That is, we can order its nodes linearly such that for any directed edge (u, v), u comes before v in the ordering. Our approach to structuring acyclic region proceeds as follows. First, we compute reaching conditions from the region header h to every node n in the region. Next, we construct the initial AST as sequence of code nodes in topological order associated with corresponding reaching conditions, i.e., it represents the control flow inside the region as **if** $(c_r(h, n_1))\{n_1\}; \ldots;$ **if** $(c_r(h, n_k))\{n_k\}$. Obviously, the initial AST is not optimal. For example, nodes with complementary conditions are represented as two `if-then` constructs **if** $(c)\{n_t\}$ **if** $(\neg c)\{n_f\}$ and not as one `if-then-else` construct **if** $(c)\{n_t\}$ **else** $\{n_f\}$. Therefore, in the second phase, we iteratively refine the initial AST to find a concise high-level representation of control flow inside the region.

1) Abstract Syntax Tree Refinement: We apply three refinement steps to AST sequence nodes. First, we check if there exist subsets of nodes that can be represented using `if-then-else`. We denote this step by *condition-based refinement* since it reasons about the logical expressions representing nodes' reaching conditions. Second, we search for nodes that can be represented by `switch` constructs. Here, we also look at the checks (comparisons) represented by each logical variable. Hence, we denote it by *condition-aware refinement*. Third, we additionally use the reachability relations among nodes to represent them as cascading `if-else` constructs. The third step is called *reachability-based refinement*.

At a high level, our refinement steps iterate over the children of each sequence node V and choose a subset $V_c \in V$ that satisfies a specific criterion. Then, we construct a new compound AST node v_c that represents control flow inside V_c and replaces it in a way that preserves the topological order of V. That is, v_c is placed after all nodes reaching it and before all nodes reached from it. Note that we define reachability between two AST nodes in terms of corresponding basic blocks in the CFG, i.e., let u, v be two AST nodes, u reaches v if u contains a basic block that reaches a basic block contained in v.

Condition-based Refinement. Here, we use the observation that nodes belonging to the true branch of an `if` construct with condition c is executed (reached) if and only if c is satisfied. That is, the reaching condition of corresponding node(s) is an AND expression of the form $c \wedge R$. Similarly, nodes belonging to the false branch have reaching conditions of the form $\neg c \wedge R$. This refinement step chooses a condition c and divides children nodes into three groups: true-branch candidates V_c, false-branch candidates $V_{\neg c}$, and remaining nodes. If the true-branch and false-branch candidates contain more than two nodes, i.e., $|V_c| + |V_{\neg c}| \geq 2$, we create a condition node v_c for c with children $\{V_c, V_{\neg c}\}$ whose conditions are replaced by terms R. Obviously, the second term of logical AND expressions (c or $\neg c$) is implied by the conditional node.

The conditions that we use in this refinement are chosen as follows: we first check for pairs of *code* nodes (n_i, n_j) that satisfy $c_r(h, n_i) = \neg c_r(h, n_j)$ and group according to $c_r(h, n_i)$. These conditions correspond to `if-then-else` constructs, and thus are given priority. When no such pairs can be found, we traverse *all* nodes in topological order (including conditional nodes) and check if nodes can be structured by the reaching condition of the currently visited node. Intuitively, this traversal mimics the nesting order by visiting the topmost nodes first. Clustering according to the corresponding conditions allows to structure inner nodes by removing common factors from logical expressions. Therefore, we iteratively repeat this step on all newly created sequence nodes to find further nodes with complementing conditions.

In our running example, when the algorithm structures the acyclic region headed at node b_1 (region R_2), it computes the initial AST as shown in Figure 7. Condition nodes are represented by white nodes with up to two outgoing edges that represent when the condition is satisfied (black arrowhead) or not (white arrowhead). Sequence nodes are depicted by blue nodes. Their children are ordered from left to right in topological order. Leaf nodes (rectangles) are the basic blocks. The algorithm performs a condition-based refinement wrt. condition $b_1 \wedge b_2$ since nodes n_5 and n_6 have complementary conditions. This results in three clusters $V_{b_1 \wedge b_2} = \{n_6\}$, $V_{\neg(b_1 \wedge b_2)} = \{n_5\}$, and $V_r = \{n_4\}$ and leads to creating a condition node. At this point, no further condition-based refinement is possible. Cifuentes proposed a method to structure compound conditions by defining four patterns that describe the shape of subgraphs resulting from short circuit evaluation of compound conditions [11]. Obviously, this method fails if no match to these patterns is found.

Condition-aware Refinement. This step checks if the child nodes, or a subset of them, can be structured as a `switch`

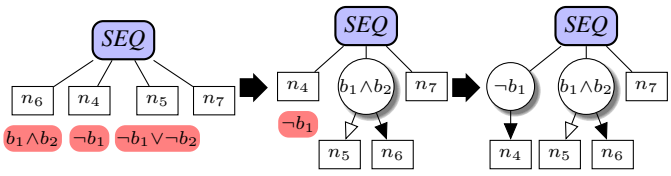


Fig. 7: Development of the initial AST when structuring the region R_2 in the running example. The initial AST (left) is refined by a condition-based refinement with respect to condition $b_1 \wedge b_2$ (middle). Finally, a condition node is created for n_4 (right).

construct. We apply this refinement when no further progress can be made by condition-based refinement and the AST has sequence nodes with more than two children. Here, we use the observation that in a `switch` construct with variable x, reaching conditions of case nodes are comparisons of x with scalar constants. A given case node is reached if x is equal to the case value or the preceding case node does not end with a `break` statement. As a result, the reaching condition is an equality check $x \stackrel{?}{=} c$ where c is a scalar constant or a logical OR expression of such checks. The reaching condition for the default case node, if it exists, can additionally contain checks for x such as $\geq$ with constants.

Our approach is to first search for a `switch` candidate node whose reaching condition is a comparison of a variable with a constant. We then cluster the remaining nodes in the sequence based on the type of their reaching conditions into three groups: case candidates V_c, default candidates V_d , and remaining items V_r. If at least two case nodes are found, i.e., $|V_c| + |V_d| \geq 3$, we construct a `switch` node v_s that replaces $V_c \cup V_d$ in the sequence. We compute the values associated with each case and determine whether the case ends with a `break` statement depending on the corresponding node's reaching condition. For this purpose, we traverse case candidate nodes in topological order which defines the lexical order of cases in the `switch` construct. When at node n, we check if the reaching condition of a subsequent case node v is a logical OR expression of the form $c_r(h, v) = c_r(h, n) \vee R_n$. This means that if n is reached, then v is also reached and thus n does not end with a `break` statement. The set of values associated to case node n is $V_n \setminus V_p$ where V_n is the set of constants checked in the reaching condition of node n and V_p is the set of values of previous cases.

Reachability-based Refinement. This is the last refinement that we apply when no further condition-based and condition-aware refinements are possible. Intuitively, a set of nodes $N = \{n_1, \ldots, n_k\}$ with nontrivial reaching conditions $\{c_1, \ldots, c_k\}$, i.e. $\forall i \in [1, k] : c_i \neq \texttt{true}$, can be represented as cascading `if-else` constructs if the following conditions are satisfied: First, there exists no path between any two nodes in N. Second, the OR expression of their reaching conditions evaluates to true, i.e., $\bigvee_{1 \leq i \leq k} c_i = \texttt{true}$. These nodes can be represented as **if** $(c_1)\,\{n_1\} \ldots$ **else if** $(c_{k-1})\,\{n_{k-1}\}$ **else** $\{n_k\}$. This eliminates the need to explicitly include condition c_k in the decompiled code as it is implied by the last `else`. The main idea is to group nodes that satisfy these conditions and construct cascading condition nodes to represent them. That is, for each node $n_i \in N$, we construct a condition node with condition c_i whose true branch is node n_i and the false branch is the next condition node for c_{i+1} (if $i < k - 1$) or n_k (if $i = k - 1$).

We iteratively process sequence nodes and construct clusters N_r that satisfy the above conditions. In each iteration, we initialize N_r to contain the last sequence node with a nontrivial reaching condition and traverse the remaining nodes backwards. A node u is added to N_r if $\forall n \in N_r : u \nrightarrow n$ since the topological order implies that no node in N_r has a path to n (this would cause this node to be before n in the order). We stop when the logical OR of reaching conditions evaluates to true. Since nodes in N_r are unreachable from each other, any ordering of them is a valid topological order. With the goal of producing well-readable code, we sort nodes in N_r by increasing complexity of the logical expressions representing their reaching conditions defined as the expression's number of terms. Finally, we build the corresponding cascading condition nodes.

C. Structuring Cyclic Regions

A loop is characterized by the existence of a back edge (n_l, n_h) from a latching node n_l into loop header node n_h. With the aim of structuring cyclic regions in a pattern-independent way, we first compute the set of loop nodes, restructure the cyclic region into a single-entry single-successor region if necessary, compute the AST of the loop body, and finally infer the loop type and condition by reasoning about the computed AST. Our CFG traversal guarantees that we handle inner loops before outer ones and thus we can assume that when structuring a cyclic region it does not contain nested loops.

1) Initial Loop Nodes and Successors: We first determine the set of *initial loop nodes* N_{loop}, i.e., nodes located on a path from the header node to a latching node. For this purpose, we compute the graph slice $S_G(n_h, N_l)$ where N_l is the set of latching nodes. This allows to compute loop nodes even if they are not dominated by the header node in the presence of abnormal entries. Abnormal entries are defined as $\exists n \in N_{loop} \setminus \{n_h\} : \texttt{Preds}(n) \not\subset N_{loop}$. If the cyclic region has abnormal entries, we transform it into a single-entry region (§V-A). We then identify the set of *initial exit nodes* N_{succ}, i.e., targets of outgoing edges from loop nodes not contained in N_{loop}. These sets are denoted as initial because they are refined by the next step to the final sets.

2) Successor Refinement and Loop Membership: In order to compute the final sets of loop nodes and successor nodes, we perform a *successor node refinement* step. The idea is that certain initial successor nodes can be considered as loop nodes, and thus we can avoid prematurely considering them as final successor nodes and avoid unnecessary restructuring. For example, a `while` loop containing `break` statements proceeded by some code results in multiple exits from the loop that converge to the unique loop successor. This step provides a precise *loop membership* definition that avoids prematurely analyzing the loop type and identifying the successor node based on initial loop nodes which may lead to suboptimal structuring. Algorithm 2 provides an overview of the successor

refinement step. The algorithm iteratively extends the current set of loop nodes by looking for successor nodes that have all their immediate predecessors in the loop and are dominated by the header node. When a successor node is identified as loop node, its immediate successors that are not currently loop nodes are added to the set of successor nodes. The algorithm stops when the set of successor nodes contains at most one node, i.e., the final unique loop successor is identified, or when the previous iteration did not find new successor nodes. If the loop still has multiple successors after refinement, we select from them the successor of the loop node with smallest post-order as the loop final successor. The remaining successors are classified as abnormal exit nodes. We then transform the region into a single-successor region as will be described in Section V-B. For instance, when structuring region R_1 in our running example (Figure 3), the algorithm identifies the following initial loop and successor nodes $N_{loop} = \{c_1, n_1, c_2, n_3, c_3\}$, $N_{succ} = \{n_2, n_9\}$. Next, node n_2 is added to the set of loop nodes since all its predecessors are loop nodes. This results in a unique loop node and the final sets $N_{loop} = \{c_1, n_1, c_2, n_3, c_3, n_2\}$, $N_{succ} = \{n_9\}$.

Algorithm 2: *Loop Successor Refinement*

Input : Initial sets of loop nodes N_{loop} and successor nodes N_{succ}; loop header n_h
Output: Refined N_{loop} and N_{succ}

```
1  N_new ← N_succ;
2  while |N_succ| > 1 ∧ N_new ≠ ∅ do
3      N_new ← ∅;
4      forall the n ∈ N_succ do
5          if preds(n) ⊆ N_loop then
6              N_loop ← N_loop ∪ {n};
7              N_succ ← N_succ \ {n};
8              N_new ← N_new ∪
               {u : u ∈ [succs(n) \ N_loop] ∧ dom(n_h,u)};
9          end
10     end
11     N_succ ← N_succ ∪ N_new
12 end
```

Phoenix [33] employs a similar approach to define loop membership. The key difference to our approach is that Phoenix assumes that the loop successor is either the immediate successor of the header or latching node. For example, in case of endless loops with multiple `break` statements or loops with unstructured continuation condition (e.g., region R_3), the simple assumption that loop successor is directly reached from loop header or latching nodes fails. In these cases Phoenix generates an endless loop and represents exits using `goto` statements. In contrast, our successor refinement technique described above does not suffer from this problem and generates structured code without needing to use `goto` statements.

3) Loop Type and Condition: In order to identify loop type and condition, we first represent each edge to the successor node as a `break` statement and compute the AST of the loop body after refinement n_b. Note that the loop body is an acyclic region that we structure as explained in §IV-B. Next, we represent the loop as endless loop with the computed

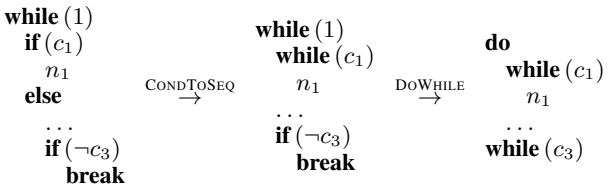


Fig. 9: Example of loop type inference of region R_1.

body's AST, i.e., $n_\ell = Loop\,[\tau_{\mathsf{endless}}, -, n_b]$. Our assumption is justified since all exits from the loop are represented by `break` statements. Finally, we infer the loop type and continuation condition by reasoning about the structure of loop n_ℓ.

Inference rules. We specify loop structuring rules as inference rules of the form:

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{C}$$

The top of the inference rule bar contains the premises $P_1, P_2, \dots, P_n$. If all premises are satisfied, then we can conclude the statement below the bar C. Figure 8 presents our loop structuring rules. The first premise in our rules describes the input loop structure, i.e., loop type and body structure. The remaining premises describe additional properties of loop body. The conclusion is described as a *transformation rule* of the form $n \rightsquigarrow \acute{n}$. Inference rules provide a formal compact notation for single-step inference and implicitly specify an inference algorithm by recursively applying rules on premises until a fixed point is reached. We denote by $\mathcal{B}_r$ a `break` statement, and by $\mathcal{B}_r^c$ a condition node that represents the statement **if** (c) {**break**}, i.e., $\mathcal{B}_r^c = Cond\,[c, Seq\,[\mathcal{B}_r], -]$. We represent by $n \Downarrow \mathcal{B}_r$ the fact that a `break` statement is attached to each exit from the control construct represented by node n. The operator $\sum$ returns the list of statements in a given node.

In our running example, computing the initial loop structure for region R_1 results in the first (leftmost) code in Figure 9. The loop body consists of an `if` statement with `break` statements only in its false branch. This matches the CondToSeq rule, which transforms the loop body into a sequence of a `while` loop and the false branch of the `if` statement. The rule states that in this case the true branch of the `if` statement (n_1) is continuously executed as long as the condition c_1 is satisfied. Then, control flows to the false branch. This is repeated until the execution reaches a `break` statement. The resulting loop body is a sequence that ends with a conditional break $\mathcal{B}_r^{\neg c_3}$ that matches the DoWhile rule. The second transformation results in the third (rightmost) loop structure. At this point the inference algorithm reaches a fixed point and terminates.

To give an intuition of the unstructured code produced by structural analysis when a region in the CFG does not match its predefined region schemas, we consider the region R_3 in our running example. Computing the body's AST of the loop in region R_3 and assuming an endless loop results in the loop represented as **while** (1) {**if** $((\neg d_1 \wedge \neg d_2) \vee (d_1 \wedge \neg d_3))$ {**break;**} $\dots$}. The loop's body starts with a conditional break and hence is structured according to the While rule into **while** $((d_1 \wedge d_3) \vee (\neg d_1 \wedge d_2))$ {$\dots$}. We wrote a small function that produces the same CFG as the region R_3 and decompiled it with Dream and Hex-Rays. Figure 11 shows that our

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad n_1 = \mathcal{B}_r^c}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{while}}, \neg c, Seq\left[n_i\right]^{i\in 2..k}\right]}\text{WHILE} \qquad \frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad n_k = \mathcal{B}_r^c}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{dowhile}}, \neg c, Seq\left[n_i\right]^{i\in 1..k-1}\right]}\text{DOWHILE}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad \forall i \in 1..k-1 : \mathcal{B}_r \notin \sum[n_i] \quad n_k = Cond\left[c, n_t, -\right]}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{endless}}, -, Seq\left[Loop\left[\tau_{\textbf{dowhile}}, \neg c, Seq\left[n_i\right]^{i\in 1..k-1}\right], n_t\right]\right]}\text{NESTEDDOWHILE}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad n_k = n'_k \Downarrow \mathcal{B}_r}{n_\ell \rightsquigarrow Seq\left[n_1, \ldots, n_{k-1}, n'_k\right]}\text{LOOPTOSEQ}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Cond\left[c, n_t, n_f\right]\right] \quad \mathcal{B}_r \notin \sum[n_t] \quad \mathcal{B}_r \in \sum[n_f]}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{endless}}, -, Seq\left[Loop\left[\tau_{\textbf{while}}, c, n_t\right], n_f\right]\right]}\text{CONDTOSEQ}$$

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Cond\left[c, n_t, n_f\right]\right] \quad \mathcal{B}_r \in \sum[n_t] \quad \mathcal{B}_r \notin \sum[n_f]}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{endless}}, -, Seq\left[Loop\left[\tau_{\textbf{while}}, \neg c, n_f\right], n_t\right]\right]}\text{CONDTOSEQNEG}$$

Fig. 8: Loop structuring rules. The input to the rules is a loop node n_ℓ.

```
1  signed int __cdecl loop(signed int a1)
2  {
3    signed int v2; // [sp+1Ch] [bp-Ch]@1
4
5    v2 = 0;
6    while ( a1 > 1 ){
7      if ( v2 > 10 )
8        goto LABEL_7;
9  LABEL_6:
10     printf("inside_loop");
11     ++v2;
12     --a1;
13   }
14   if ( v2 <= 100 )
15     goto LABEL_6;
16 LABEL_7:
17   printf("loop_terminated");
18   return v2;
19 }
```

Fig. 10: Decompiled code generated by Hex-Rays.

```
1  int loop(int a){
2    int b = 0;
3    while((a <= 1 && b <= 100)||(a > 1 && b <= 10)){
4       printf("inside_loop");
5       ++b;
6       --a;
7    }
8    printf("loop_terminated");
9    return b;
10 }
```

Fig. 11: Decompiled code generated by DREAM.

approach correctly found the loop type and continuation condition. In comparison, Hex-Rays produced unstructured code with two `goto` statements as shown in Figure 10; one `goto` statement jumps outside the loop and the other one jumps back in the loop.

D. Side Effects

Our structuring algorithm may result in the same condition appearing multiple times in the computed AST. For example, structuring region R_2 in the running example leads to the AST shown in Figure 7 where condition b_1 is tested twice. If the variables tested by condition b_1 are modified in block n_4, the second check of b_1 in the AST would not be the same as the first check. As a result, the code represented by the computed AST would not be semantically equivalent to the CFG representation.

To guarantee the semantics-preserving property of our algorithm, we first check if any condition is used multiple times in the computed AST. If this is the case, we check if any of the variables used in the test is changed on an execution path between any two uses. This includes if the variable is assigned a new value, used in a call expression, or used in reference expression (its address is read). If a possible change is detected, we insert a Boolean variable to store the initial value of the condition. All subsequent uses of the condition are replaced by the inserted Boolean variable.

E. Summary

In this section, we have discussed our approach to creating an AST for single-entry and single-successor CFG regions. The above algorithm can structure every CFG except cyclic regions with multiple entries and/or multiple successors. The following section discusses how we handle these problematic regions.

V. SEMANTICS-PRESERVING CONTROL-FLOW TRANSFORMATIONS

In this section, we describe our method to transform cyclic regions into semantically equivalent single-entry single-successor regions. As the only type of regions that cannot be structured by our pattern-independent structuring algorithm are cyclic regions with multiple entries or multiple successors, we

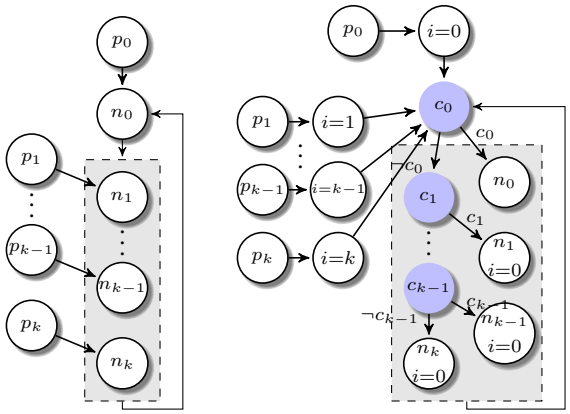


Fig. 12: Transforming abnormal entries: multi-entry loops (left) are transformed into semantically equivalent single-entry loops (right). Tags c_n represent the logical predicates $i = n$.

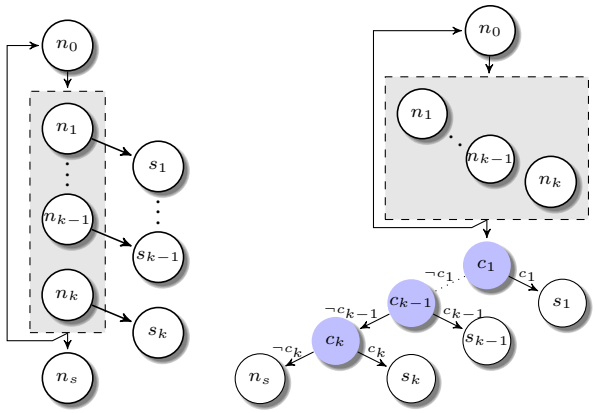


Fig. 13: Transforming abnormal exits: loops with multiple successors (left) are transformed into semantically equivalent single-successor loops (right).

apply the proposed transformations on those regions. Based on the previous steps we know the following information about the cyclic region: *a*) region nodes N_{loop}, *b*) normal entry n_h, and *c*) successor node n_s.

A. Restructuring Abnormal Entries

The high-level approach to structuring abnormal entries (cf. IV-C1) is illustrated in Figure 12. The underlying idea is to insert a *structuring* variable (i in Figure 12) that takes different values based on the node at which the loop is entered. We then redirect all loop entries to a new header node (c_0) where we insert cascading condition nodes that test equality of the structuring variable to the values representing the different entries. Each condition node transfers control to the corresponding entry node if the check is satisfied and to the next check (or the last entry node) otherwise. All incoming edges to the original header n_0 are directed to the new header c_0. We preserve semantics by inserting assignments of zero to the structuring variable at the end of each abnormal entry so that the next loop iteration is executed normally.

For each loop node $n \in N_{loop}$ with incoming edges from outside the loop, we first compute the set of corresponding abnormal entries $E_n = \{(p, n) \in E : p \notin N_{loop}\}$. Then, we create a new code node consisting of assignment of the structuring variable to a unique value and redirect edges in E_n into the newly created node. Finally, we add an edge from the new code node to the new loop header. We represent the normal entry to the loop by assigning zero to the structuring variable. In order to produce well-readable decompiled code, we strive to keep the changes caused by our transformations minimal. For this reason, the first check we make at the new loop header is whether the loop is entered normally. In this case, we transfer control to the original header. This has the advantage of preserving loop type and minimally modifying the original condition. For example, restructuring a `while` loop **while** $(c)\,\{\ldots\}$ with abnormal entries results in a `while` loop whose condition contains additional term representing the abnormal entries **while** $(c \vee i \neq 0)\,\{\ldots\}$.

B. Restructuring Abnormal Exits

The high-level approach to structuring abnormal exits (cf. IV-C2) is illustrated in Figure 13. Our approach computes for each exit the unique condition that causes the control-flow to choose that exit and redirects all exit edges to a new successor node. Here, we insert cascading condition nodes that successively check the exit conditions and transfer control to the original exit if the corresponding condition is satisfied or to the next check (or the last exit node) otherwise. We restructure abnormal exits after restructuring abnormal entries. Therefore, at this stage the loop successor is known and the loop has a unique entry node dominating all loop nodes.

We start by computing the set of edges that exit the cyclic region to a node other than the successor node $E_{out} = \{(n, u) \in E : n \in N_{loop} \wedge u \notin N_{loop} \cup \{n_s\}\}$. Then, we compute *nearest common dominator* (NCD) for the set of source nodes for edges in E_{out}, denoted n_{ncd}. In a graph $G\,(N, E)$, a node $d \in N$ is the *nearest common dominator* of a set of nodes $U \subseteq N$ if d dominates all nodes of U and there exists no node $\acute{d} \neq d$ that dominates all nodes of U and is strictly dominated by d. Since the loop header dominates all loop nodes (after restructuring abnormal entries), the NCD of any subset of loop nodes is also a loop node. The basic idea here is that any change in the control flow to a given exit does not happen before n_{ncd}. Thus, we need to compute the set of reaching conditions starting from n_{ncd}, i.e., we compute reaching conditions $c_r\,(n_{\mathrm{ncd}}, u)$ to the target nodes of edges in E_{out}.

C. Summary

At this point we transformed the CFG to an AST that contains only high-level control constructs and no `goto` statements. As a final step, we introduce several optimizations to improve the readability of the decompiled output.

VI. Post-Structuring Optimizations

After having computed the abstract syntax tree, we perform several optimizations to improve code readability. Specifically, we perform three optimizations: control constructs simplification, outlining certain string functions, and variable renaming.

We implement several transformations that find simpler forms for certain control constructs. For instance, we transform `if` statements that assign different values to the same variable into a ternary operator. That is, code such as **if** $(c)\,\{x = v_t\}$ **else** $\{x = v_f\}$ is transformed into the equivalent form $x = c\,?\,v_t : v_f$. Also, we identify `while` loops that can be represented as `for` loops. `for` loop candidates are `while` loops that have a variable x used both in their continuation condition and the last statement in their body. We then check if a single definition of x reaches the loop entry that is only used in the loop body. We transform the loop into a `for` loop if the variables used in the definition are not used on the way from the definition to the loop entry. These checks allow us to identify `for` loops even if their initialization statements are not directly before the loop.

Several functions such as `strcpy`, `strlen`, or `strcmp` are often inlined by compilers. That is, a function call is replaced by the called function body. Having several duplicates of the same function results in larger code and is detrimental to manual analysis. DREAM recognizes and outlines several functions. That is, it replaces the corresponding code by the equivalent function call.

For the third optimization, we leverage API calls to assign meaningful names to variables. API functions have known signatures including the types and names of parameters. If a variable is used in an API call, we give it the name of corresponding parameter if that name is not already used.

VII. EVALUATION

In this section, we describe the results of the experiments we have performed to evaluate DREAM. We base our evaluation on the technique used to evaluate Phoenix by Schwartz *et al.* [33]. This evaluation used the GNU `coreutils` to evaluate the quality of the decompilation results. We compared our results with Phoenix [33] and Hex-Rays [22]. We included Hex-Rays because it is the leading commercial decompiler and the *de facto* industry standard. We tested the latest version of Hex-Rays at the time of writing, which is v2.0.0.140605. We picked Phoenix because it is the most recent and advanced academic decompiler. We did not include *dcc* [11], DISC [28], REC [1], and Boomerang [17] in our evaluation. The reason is that these projects are either no longer actively maintained (e.g., Boomerang) or do not support x86 (e.g., *dcc*). However, most importantly, they are outperformed by Phoenix. The implementation of Phoenix is not publicly available yet. However, the authors kindly agreed to share both the `coreutils` binaries used in their experiments and the raw decompiled source code produced by Phoenix to enable us to compute our metrics and compare our results with theirs. We very much appreciate this good scientific practice. This way, we could ensure that all three decompilers are tested on the same binary code base. We also had the raw source code produced by all three decompilers as well, so we can compare them fairly. In addition to the GNU `coreutils` benchmark we also evaluated our approach using real-world malware samples. Specifically, we decompiled and analyzed ZeusP2P, SpyEye, Cridex. For this part of our evaluation we could only compare our approach to Hex-Rays since Phoenix is not yet released.

A. Metrics

We evaluate our approach with respect to the following quantitative metrics.

- **Correctness.** Correctness measures the functional equivalence between the decompiled output and the input code. More specifically, two functions are semantically equivalent if they follow the same behavior and produce the same results when they are executed using the same set of parameters. Correctness is a crucial criterion to ensure that the decompiled output is a faithful representation of the corresponding binary code.
- **Structuredness.** Structuredness measures the ability of a decompiler to recover high-level control flow structure and produce structured decompiled code. Structuredness is measured by the number of generated `goto` statements in the output. Structured code is easier to understand [16] and helps scale program analysis [31]. For this reason, it is desired to have as few `goto` statements in the decompiled code as possible. These statements indicate the failure to find a better representation of control flow.
- **Compactness.** For compactness we perform two measurements: first, we measure the total lines of code generated by each decompiler. This gives a global picture on the compactness of decompiled output. Second, we count for how many functions each decompiler generated the fewest lines of code compared to the others. If multiple decompilers generate the same (minimal) number of lines of code, that is counted towards the total of each of them.

B. Experiment Setup & Results

To evaluate our algorithm on the mentioned metrics, we conducted two experiments.

1) Correctness Experiment: We evaluated the correctness of our algorithm on the GNU `coreutils` 8.22 suite of utilities. `coreutils` consist of a collection of mature programs and come with a suite of high-coverage tests. We followed a similar approach to that proposed in [33] where the `coreutils` tests were used to measure correctness. Also, since the `coreutils` source code contains `goto` statements, this means that both parts of our algorithm are invoked; the pattern-independent structuring part and the semantics-preserving transformations part. Our goal is to evaluate the control-flow structuring component. For this, we computed the CFG for each function in the `coreutils` source code and provided it as input to our algorithm. Then, we replaced the original functions with the generated algorithm output, compiled the restructured `coreutils` source code, and finally executed the tests. We used *joern* [41] to compute the CFGs. Joern is a state-of-the-art platform for analysis of C/C++ code. It generates *code property graphs*, a novel graph representation of code that combines three classic code representations; ASTs, CFGs, and Program Dependence Graphs (PDG). Code property graphs are stored in a Neo4J graph database. Moreover, a thin python interface for joern and a set of useful utility traversals are provided to ease interfacing with the graph database. We iterated over all parsed functions in the database and extracted the CFGs. We then transformed statements in the CFG nodes into DREAM's intermediate representation. The extracted graph representation was then provided to our structuring algorithm. Under the assumption of correct parsing, we

Considered Functions F	$\|F\|$	**Number of** gotos
Functions after preprocessor	1,738	219
Functions correctly parsed by *joern*	1,530	129
Functions passed tests after structuring	1,530	0

TABLE II: Correctness results.

can attribute the failure of any test on the restructured functions to the structuring algorithm. To make the evaluation tougher, we used the source files produced by the C-preprocessor, since depending on the operating system and installed software, some functions or parts of functions may be removed by the preprocessor before passing them to the compiler. That in turn would lead to potential structuring errors to go unnoticed if the corresponding function is removed by the preprocessor. We got the preprocessed files by passing the `--save-temps` to `CFLAGS` in the configure script. The preprocessed source code contains 219 `goto` statements.

2) Correctness Results: Table II shows statistics about the functions included in our correctness experiments. The preprocessed `coreutils` source code contains 1,738 functions. We encountered parsing errors for 208 functions. We excluded these functions from our tests. The 1,530 correctly parsed functions were fed to our structuring algorithm. Next, we replaced the original functions in `coreutils` by the structured code produced by our algorithm. The new version of the source code passed all `coreutils` tests. This shows that our algorithm correctly recovered control-flow abstractions from the input CFGs. More importantly, `goto` statements in the original source code are transformed into semantically equivalent structured forms.

The original Phoenix evaluation shows that their control-flow structuring algorithm is correct. Thus, both tools correctly structure the input CFG.

3) Structuredness and Compactness Experiment: We tested and compareed DREAM to Phoenix and Hex-Rays. In this experiment we used the same GNU `coreutils` 8.17 binaries used in Phoenix evaluation. Structuredness is measured by the number of `goto` statements in code. These statements indicate that the structuring algorithm was unable to find a structured representation of the control flow. Therefore, structuredness is inversely proportional to the number of `goto` statements in the decompiled output. To measure compactness, we followed a straightforward approach. We used David A. Wheeler's SLOCCount utility to measure the lines of code in each decompiled function. To ensure fair comparison, the Phoenix evaluation only considered functions that were decompiled by both Phoenix and Hex-Rays. We extend this principle to only consider functions that were decompiled by all the three decompilers. If this was not done, a decompiler that failed to decompile functions would have an unfair advantage. Beyond that, we extend the evaluation performed by Schwartz *et al.* [33] in several ways.

- *Duplicate functions*. In the original Phoenix evaluation all functions were considered, i.e., including duplicate functions. It is common to have duplicate functions as the result of the same library function being statically linked to several binaries, i.e., its code is copied into the binary. Depending on the duplicate functions this can skew the results. Thus, we wrote a small IDAPython script that extracts the assembly listings of all functions and then computed the SHA-512 hash for the resulting files. We found that of the 14,747 functions contained in the `coreutils` binaries, only 3,141 functions are unique, i.e., 78.7% of the functions are duplicates. For better comparability, we report the results both on the filtered and unfiltered function lists. However, for future comparisons we would argue that filtering duplicate functions before comparison avoids skewing the results based on the same code being included multiple times.
- Also in the original Phoenix evaluation only *recompilable functions* were considered in the `goto` test. In the context of `coreutils`, this meant that only 39% of the unique functions decompiled by Phoenix were considered in the `goto` experiment. We extend these tests to consider the intersection of all functions produced by the decompilers, since even non-recompilable functions are valuable and important to look at, especially for malware and security analysis. For instance, the property graph approach [41] to find vulnerabilities in source code does not assume that the input source code is compilable. Also, understanding the functionality of a sample is the main goal of manual malware analysis. Hence, the quality of *all* decompiled code is highly relevant and thus included in our evaluation. For completeness, we also present the results based on the functions used in the original evaluation done by Schwartz *et al*.

4) Structuredness & Compactness Results: Table III summarizes the results of our second experiment. For the sake of completeness, we report our results in two settings. First, we consider all functions without filtering duplicates as was done in the original Phoenix evaluation. We report our results for the functions considered in the original Phoenix evaluation (i.e., only recompilable functions) (T1) and for the intersection of all functions decompiled by the three decompilers (T2). In the second setting we only consider unique functions and again report the results only for the functions used in the original Phoenix study (T3) and for all functions (T4). In the table $|F|$ denotes the number of functions considered. The following three columns report on the metrics defined above. First, the number of `goto` statements in the functions is presented. This is the main contribution of our paper. While both state-of-the-art decompilers produced thousands of `goto` statements for the full list of functions, DREAM produced none. We believe this is a major step forward for decompilation. Next, we present total lines of code generated by each decompiler in the four settings. DREAM generated more compact code overall than Phoenix and Hex-Rays. When considering all unique functions, DREAM's decompiled output consists of 107k lines of code in comparison to 164k LoC in Phoenix output and 135k LoC produced by Hex-Rays. Finally, the percentage of functions for which a given decompiler generated the most compact function is depicted. In the most relevant test setting T4, DREAM produced the minimum lines of code for 75.2% of the functions. For 31.3% of the functions, Hex-Rays generated the most compact code. Phoenix achieved the best compactness in 0.7% of the cases. Note that the three percentages exceed

Considered Functions F	$\|F\|$	Number of goto Statements			Lines of Code			Compact Functions		
		DREAM	Phoenix	Hex-Rays	DREAM	Phoenix	Hex-Rays	DREAM	Phoenix	Hex-Rays
`coreutils` functions with duplicates										
$T_1 : F_p^r \cap F_h^r$	8,676	0	40	47	93k	243k	120k	81.3%	0.3%	32.1%
$T_2 : F_d \cap F_p \cap F_h$	10,983	0	4,505	3,166	196k	422k	264k	81%	0.2%	30.4%
`coreutils` functions without duplicates										
$T_3 : F_p^r \cap F_h^r$	785	0	31	28	15k	30k	18k	74.9%	1.1%	36.2%
$T_4 : F_d \cap F_p \cap F_h$	1,821	0	4,231	2,949	107k	164k	135k	75.2%	0.7%	31.3%
Malware Samples										
ZeusP2P	1,021	0	N/A	1,571	42k	N/A	53k	82.9%	N/A	14.5%
SpyEye	442	0	N/A	446	24k	N/A	28k	69.9%	N/A	25.7%
Cridex	167	0	N/A	144	7k	N/A	9k	84.8%	N/A	12.3%

TABLE III: Structuredness and compactness results. For the `coreutiles` benchmark, we denote by F_x the set of functions decompiled by compiler x. F_x^r is the set of recompilable functions decompiled by compiler x. d represents DREAM, p represents Phoenix, and h represents Hex-Rays.

100% due to the fact that multiple decompilers could generate the same minimal number of lines of code. In a one on one comparison between DREAM and Phoenix, DREAM scored 98.8% for the compactness of the decompiled functions. In a one on one comparison with Hex-Rays, DREAM produced more compact code for 72.7% of decompiled functions.

5) Malware Analysis: For our malware analysis, we picked three malware samples from three families: ZeusP2P, Cridex, and SpyEye. The results for the malware samples shown in Table III are similarly clear. DREAM produces `goto`-free and compact code. As can be seen in the Zeus sample, Hex-Rays produces 1,571 `goto` statements. These statements make analyzing these pieces of malware very time-consuming and difficult. While further studies are needed to evaluate if compactness is always an advantage, the total elimination of `goto` statements from the decompiled code is a major step forward and has already been of great benefit to us in our work analyzing malware samples.

Due to space constraints, we cannot present a comparison of the decompiled malware source code in this paper. For this reason, we have created a supplemental document which can be accessed under the following URL: https://net.cs.uni-bonn.de/fileadmin/ag/martini/Staff/yakdan/code_snippets_ndss_2015.pdf. Here we present listings of selected malware functions so that the reader can get a personal impression on the readability improvements offered by DREAM compared to Hex-Rays.

VIII. RELATED WORK

There has been much work done in the field of decompilation and abstraction recovery from binary code. In this section, we review related work and place DREAM in the context of existing approaches. We start by reviewing control-flow structuring algorithms. Next, we discuss work in decompilation, binary code extraction and analysis. Finally, techniques to recover type abstractions from binary code are discussed.

Control-flow structuring. There exist two main approaches used by modern decompilers to recover control-flow structure from the CFG representation, namely *interval analysis* and *structural analysis*. Originally, these techniques were developed to assist data flow analysis in optimizing compilers. Interval analysis [3, 13] deconstructs the CFG into nested regions called *intervals*. The nesting structure of these regions helps to speed up data-flow analysis. Structural analysis [34] is a refined form of interval analysis that is developed to enable the syntax-directed method of data-flow analysis designed for ASTs to be applicable on low-level intermediate code. These algorithms are also used in the context of decompilation to recover high-level control constructs from the CFG.

Prior work on control-flow structuring proposed several enhancement to vanilla structural analysis. The goal is to recover more control structure and minimize the number of `goto` statements in the decompiled code. Engel *et. al.* [18] extended structural analysis to handle C-specific control statements. They proposed a Single Entry Single Successor (SESS) analysis as an extension to structural analysis to handle the case of statements that exist before `break` and `continue` statements in the loop body.

These approaches share a common property; they rely on a predefined set of region patterns to structure the CFG. For this reason, they cannot structure arbitrary graphs without using `goto` statements. Our approach is fundamentally different in that it does not rely on any patterns.

Another related line of research lies in the area of eliminating `goto` statements at the source code level such as [19] and [39]. These approaches define transformations at the AST level to replace `goto` statements by equivalent constructs. In some cases, several transformations are necessary to remove a single `goto` statement. These approaches increase the code size and miss opportunities to find more concise forms to represent the control-flow. Moreover, they may insert unnecessary Boolean variables. For example, these approaches cannot find the concise form found by DREAM for region R_3 in our running example. These algorithms do not solve the control-flow structuring problem as defined in Section II-B.

Decompilers. Cifuentes laid the foundations for modern decompilers. In her PhD thesis [11], she presented several techniques to decompile binary code into a high-level language.

These techniques were implemented in *dcc*, a decompiler for Intel 80286/DOS to C. The structuring algorithm in *dcc* [12] is based on interval analysis. She also presented four region patterns to structure regions resulted from the short-circuit evaluation of compound conditional expressions, e.g., $x \vee y$.

Van Emmerik proposed to use the Static Single Assignment (SSA) form for decompilation in his PhD thesis [17]. His work demonstrates the advantages of the SSA form for several data flow components of decompilers, such as expression propagation, identifying function signatures, and eliminating dead code. His approach is implemented in Boomerang, an open-source decompiler. Boomerang's structuring algorithm is based on *parenthesis theory* [35]. Although faster than interval analysis, it recovers less structure.

Chang *et. el.* [9] demonstrated the possibility of applying source-level tools to assembly code using decompilation. For this goal, they proposed a modular decompilation architecture. Their architecture consists of a series of decompilers connected by intermediate languages. For their applications, no control-flow structuring is performed.

Hex-Rays is the *de facto* industry standard decompiler. It is built as plugin for the Interactive Disassembler Pro (IDA). Hex-Rays is closed source, and thus little is known about its inner workings. It uses structural analysis [22]. As noted by Schwartz *et el.* in [33], Hex-Rays seems to use an improved version of vanilla structural analyses.

Yakdan *et al.* [40] developed REcompile, a decompiler that employs interval analysis to recover control structure. The authors also proposed node splitting to reduce the number of `goto` statements. Here, nodes are split into several copies. While this reduces the amount of `goto` statements, it increases the size of decompiled output.

Phoenix is the state-of-the-art academic decompiler [33]. It is built on top of the CMU Binary Analysis Platform (BAP) [8]. BAP lifts sequential x86 assembly instructions into an intermediate language called BIL. It also uses TIE [29] to recover types from binary code. Phoenix enhances structural analysis by employing two techniques: first, *iterative refinement* chooses an edge and represents it using a `goto` statement when the algorithm cannot make further progress. This allows the algorithm to find more structure. Second, *semantics-preserving* ensures correct control structure recovery. The authors proposed correctness as an important metric to measure the performance of a decompiler.

The key property that all structuring algorithms presented above share is the reliance on pattern matching, i.e, they use a predefined set of region schemas that are matched against regions in the CFG. This is a key issue that prevents these algorithms from structuring arbitrary CFGs. This leads to unstructured decompiled output with `goto` statements. Our algorithm does not rely on such patterns and is therefore able to produce well-structured code without a single `goto` statement.

Binary code extraction. Correctly extracting binary code is essential for correct decompilation. Research in this field is indispensable for decompilation. Kruegel *et al.* presented a method [27] to disassemble x86 obfuscated code. Jakstab [26] is a static analysis framework for binaries that follows the paradigm of *iterative disassembly*. That is, it interleaves multiple disassembly rounds with data-flow analysis to achieve accurate and complete CFG extraction. Zeng *et el.* presented *trace-oriented programming* (TOP) [43] to reconstruct program source code from execution traces. The executed instructions are translated into a high-level program representation using C with templates and inlined assembly. TOP relies on dynamic analysis and is therefore able to cope with obfuscated binaries. With the goal of achieving high coverage, an *offline combination* component combines multiple runs of the binary. BitBlaze [37] is a binary analysis platform. The CMU Binary Analysis Platform (BAP) [8] is successor to the binary analysis techniques developed for Vine in the BitBlaze project.

Type recovery. Reconstructing type abstractions from binary code is important for decompilation to produce correct and high-quality code. This includes both elementary and complex types. Several prominent approaches have been developed in this field including Howard [36], REWARDS [30], TIE [29], and [23]. Other work [15, 20, 21, 25] focused on C++ specific issues, such as recovering C++ objects, reconstructing class hierarchy, and resolving indirect calls resulting from virtual inheritance. Since our work focuses on the control flow structuring we do not make a contribution to type recovery but we based our type recovery on TIE [29].

IX. Conclusion

In this paper we presented the first control-flow structuring algorithm that is capable of recovering all control structure and thus does not generate any `goto` statements. Our novel algorithm combines two techniques: pattern-independent structuring and semantics-preserving transformations. The key property of our approach is that it does not rely on any patterns (region schemas). We implemented these techniques in our DREAM decompiler and evaluated the correctness of our control-flow structuring algorithm. We also evaluated our approach against the *de facto* industry standard decompiler, Hex-Rays, and the state-of-the-art academic decompiler, Phoenix. Our evaluation shows that DREAM outperforms both decompilers; it produced more compact code and recovered the control structure of all the functions in the test without any `goto` statements. We also decompiled and analyzed a number of real-world malware samples and compared the results to Hex-Rays. Again, DREAM performed very well, producing `goto`-free and compact code compared to Hex-Rays, which had one `goto` for every 32 lines of code. This represents a significant step forward for decompilation and malware analysis. In future work, we will further examine the quality of the code produced by DREAM specifically concerning the compactness. Our experience based on the malware samples we analyzed during the course of this paper suggests that more compact code is better for human understanding. However, it is conceivable that in some cases less compact code is easier to understand. This will require further research and potential optimization of the post-processing step.

Acknowledgements

We are grateful to Fabian Yamaguchi, the author of joern, who was very helpful and created several patches to improve the parsing of the `coreutils`. We sincerely thank Edward J. Schwartz for sharing the Phoenix experiments results. We

would also like to thank the anonymous reviewers for their valuable feedback.

REFERENCES

[1] REC Studio 4 - Reverse Engineering Compiler. http://www.backerstreet.com/rec/rec.htm. Page checked 7/20/2014.

[2] The IDA Pro disassembler and debuger. http://www.hex-rays.com/idapro/.

[3] F. E. Allen, "Control Flow Analysis," in *Proceedings of ACM Symposium on Compiler Optimization*, 1970.

[4] D. Andriesse, C. Rossow, B. Stone-Gross, D. Plohmann, and H. Bos, "Highly Resilient Peer-to-Peer Botnets Are Here: An Analysis of Gameover Zeus," in *Proceedings of the 8th IEEE International Conference on Malicious and Unwanted Software (MALWARE)*, 2013.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World," *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[6] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The World's Fastest Taint Tracker," in *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2011.

[7] D. Brumley, T. Chiueh, R. Johnson, H. Lin, and D. Song, "RICH: Automatically Protecting Against Integer-Based Vulnerabilities," in *Proceedings of the 14th Network and Distributed System Security Symposium (NDSS)*, 2007.

[8] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011.

[9] B.-Y. E. Chang, M. Harren, and G. C. Necula, "Analysis of Low-level Code Using Cooperating Decompilers," in *Proceedings of the 13th International Conference on Static Analysis (SAS)*, 2006.

[10] W. Chang, B. Streiff, and C. Lin, "Efficient and Extensible Security Enforcement Using Dynamic Data Flow Analysis," in *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS)*, 2008.

[11] C. Cifuentes, "Reverse Compilation Techniques," Ph.D. dissertation, Queensland University of Technology, 1994.

[12] ——, "Structuring Decompiled Graphs," in *Proceedings of the 6th International Conference on Compiler Construction (CC)*, 1996.

[13] J. Cocke, "Global Common Subexpression Elimination," in *Proceedings of the ACM Symposium on Compiler Optimization*, 1970.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. The MIT Press, 2009.

[15] D. Dewey and J. T. Giffin, "Static detection of C++ vtable escape vulnerabilities in binary code," in *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, 2012.

[16] E. W. Dijkstra, "Letters to the Editor: Go to Statement Considered Harmful," *Communications of the ACM*, vol. 11, no. 3, pp. 147–148, Mar. 1968.

[17] M. J. V. Emmerik, "Static Single Assignment for Decompilation," Ph.D. dissertation, University of Queensland, 2007.

[18] F. Engel, R. Leupers, G. Ascheid, M. Ferger, and M. Beemster, "Enhanced Structural Analysis for C Code Reconstruction from IR Code," in *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2011.

[19] A. Erosa and L. J. Hendren, "Taming Control Flow: A Structured Approach to Eliminating Goto Statements," in *Proceedings of 1994 IEEE International Conference on Computer Languages*, 1994.

[20] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina, "SmartDec: Approaching C++ Decompilation," in *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE)*, 2011.

[21] A. Fokin, K. Troshina, and A. Chernov, "Reconstruction of Class Hierarchies for Decompilation of C++ Programs," in *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[22] I. Guilfanov, "Decompilers and Beyond," in *Black Hat, USA*, 2008.

[23] I. Haller, A. Slowinska, and H. Bos, "MemPick: High-Level Data Structure Detection in C/C++ Binaries," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, 2013.

[24] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, "Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[25] W. Jin, C. Cohen, J. Gennari, C. Hines, S. Chaki, A. Gurfinkel, J. Havrilla, and P. Narasimhan, "Recovering C++ Objects From Binaries Using Inter-Procedural Data-Flow Analysis," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[26] J. Kinder and H. Veith, "Jakstab: A Static Analysis Platform for Binaries," in *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, 2008.

[27] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna, "Static Disassembly of Obfuscated Binaries," in *Proceedings of the 13th Conference on USENIX Security Symposium*, 2004.

[28] S. Kumar. DISC: Decompiler for TurboC. http://www.debugmode.com/dcompile/disc.htm. Page checked 7/20/2014.

[29] J. Lee, T. Avgerinos, and D. Brumley, "TIE: Principled Reverse Engineering of Types in Binary Programs," in *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*, 2011.

[30] Z. Lin, X. Zhang, and D. Xu, "Automatic Reverse Engineering of Data Structures from Binary Execution," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[31] S. S. Muchnick, *Advanced Compiler Design and Implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[32] C. Rossow, D. Andriesse, T. Werner, B. Stone-Gross, D. Plohmann, C. J. Dietrich, and H. Bos, "P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P)*, 2013.

[33] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[34] M. Sharir, "Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers," *Computer Languages*, vol. 5, no. 3-4, pp. 141–153, Jan. 1980.

[35] D. Simon, "Structuring Assembly Programs," Honours thesis, University of Queensland, 1997.

[36] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A Dynamic Excavator for Reverse Engineering Data Structures," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[37] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, 2008.

[38] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving Integer Security for Systems with KINT," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[39] M. H. Williams and G. Chen, "Restructuring Pascal Programs Containing Goto Statements," *The Computer Journal*, 1985.

[40] K. Yakdan, S. Eschweiler, and E. Gerhards-Padilla, "REcompile: A Decompilation Framework for Static Analysis of Binaries," in *Proceedings of the 8th IEEE International Conference on Malicious and Unwanted Software (MALWARE)*, 2013.

[41] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and Discovering Vulnerabilities with Code Property Graphs," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[42] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing Missing Checks in Source Code for Vulnerability Discovery," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

[43] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.