A Human-Centric Approach For Binary Code Decompilation

Dissertation

zur

Erlangung des Doktorgrades (Dr. rer. nat.)

der

Mathematisch-Naturwissenschaftlichen Fakultät

der

Rheinischen Friedrich-Wilhelms-Universität Bonn

vorgelegt von

Khaled Yakdan

aus

Souieda, Syrien

Bonn, 2017

Angefertigt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Rheinischen Friedrich-Wilhelms-Universität Bonn

1. Gutachter: Prof. Dr. Matthew Smith
2. Gutachter: Prof. Dr. Eric Bodden

Tag der Promotion: 15.02.2018
Erscheinungsjahr: 2018

Abstract

Many security techniques have been developed both in academia and industry to analyze source code, including methods to discover bugs, apply taint tracking, or find vulnerabilities. These source-based techniques leverage the wealth of high-level abstractions available in the source code to achieve good precision and efficiency. Unfortunately, these methods cannot be applied directly on binary code which lacks such abstractions. In security, there are many scenarios where analysts only have access to the compiled version of a program. When compiled, all high-level abstractions, such as variables, types, and functions, are removed from the final version of the program that security analysts have access to.

This dissertation investigates novel methods to recover abstractions from binary code. First, a novel pattern-independent control flow structuring algorithm is presented to recover high-level control-flow abstractions from binary code. Unlike existing structural analysis algorithms which produce unstructured code with many `goto` statements, our algorithm produces fully-structured `goto`-free decompiled code. We implemented this algorithm in a decompiler called DREAM. Second, we develop three categories of code optimizations in order to simplify the decompiled code and increase readability. These categories are expression simplification, control-flow simplification and semantics-aware naming. We have implemented our usability extensions on top of DREAM and call this extended version DREAM^{++}.

We conducted the first user study to evaluate the quality of decompilers for malware analysis. We have chosen malware since it represents one of the most challenging cases for binary code analysis. The study included six reverse engineering tasks of real malware samples that we obtained from independent malware experts. We evaluated three decompilers: the leading industry decompiler Hex-Rays and both versions of our decompiler DREAM and DREAM^{++}. The results of our study show that our improved decompiler DREAM^{++} produced significantly more understandable code that outperforms both Hex-Rays and DREAM. Using DREAM^{++} participants solved 3× more tasks than when using Hex-Rays and 2× more tasks than when using DREAM. Moreover, participants rated DREAM^{++} significantly higher than the competition.

Contents

Contents iii

Publications vii

1 Introduction 1

1.1 Research Questions . . . 4
1.2 Thesis Contributions . . . 5
1.3 Thesis Outline . . . 6

2 The Dream^{++} Decompiler 9

2.1 Overview . . . 10
2.2 Naming Conventions . . . 12
2.3 Decompiler Design . . . 12
2.4 Static Single Assignment . . . 13
2.5 Type Analysis . . . 15
2.6 Data Flow Analysis . . . 16
2.7 Summary . . . 19

3 Control-Flow Structuring 21

3.1 Introduction . . . 22
3.2 Background & Problem Definition . . . 24

3.3 Approach Overview 29
3.4 Pattern-Independent Control-Flow Structuring 31
3.5 Semantics-Preserving Control-Flow Transformations 43
3.6 `goto`-Free Output 47
3.7 Evaluation 49
3.8 Related Work 56
3.9 Summary 59

4 Usability Optimizations **61**
4.1 Introduction 62
4.2 Problem Statement & Overview 63
4.3 Expression Simplification 68
4.4 Code Query and Transformation 74
4.5 Control-Flow Simplification 79
4.6 Semantics-Aware Naming 83
4.7 Related Work 85
4.8 Summary 87

5 Malware Analysis User Study **89**
5.1 User Study Design 90
5.2 User Study 99
5.3 Related Work 105
5.4 Summary 106

6 Conclusion and Future Work **109**
6.1 Conclusion 109
6.2 Future Work 110

Bibliography **113**

A Code Snippets in the User Study **125**
A.1 Task 1 125
A.2 Task 2 127

A.3 Task 3 130
A.4 Task 4 135
A.5 Task 5 139
A.6 Task 6 146

List of Figures **155**

List of Tables **159**

Publications

The research presented in this thesis was also published in the following peer-reviewed conference proceedings:

- 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016

- Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the 22nd Network and Distributed System Security (NDSS) Symposium*, 2015. **Distinguished Paper Award**

- Khaled Yakdan, Sebastian Eschweiler, and Elmar 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

During the work on this thesis, the author has also participated in other lines of research, such as botnet tracking, bug search in binary code, and malware analysis. This resulted in the following publications:

- Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A Comprehensive Measurement Study of Domain Generating Malware. In *Proceedings of the 25th USENIX Security Symposium*, 2016

- Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016

- Thomas Barabosch, Adrian Dombeck, Khaled Yakdan, and Elmar Gerhards-Padilla. Bot-Watcher: Transparent and Generic Botnet Tracking. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015

1 Introduction

Computers are ubiquitous in our modern society, and they affect almost every aspect of our life. Software is what really turns computers into very powerful and smart devices that are capable of performing useful tasks. Today, normal users install many programs on their computers such as web browsers, multimedia apps, text editors, games, etc. These programs sometimes come from unknown or untrusted sources. This is also the case for companies that use software from third-parties, which they do not control. The strong reliance on third-party software creates a wide spectrum of serious security risks.

Unfortunately, some programs are buggy or may even contain deliberately inserted backdoors. This enables attackers to exploit these vulnerabilities in order to gain access to the systems and install malicious software (*malware*). The installed malware can then steal sensitive information, manipulate private data, and prevent access of legitimate users. For this reason, it is a extremely important to secure the systems we depend on. Since we cannot prevent developers from making mistakes and we do not control the majority of software we use, we need effective techniques to quickly analyze and understand the functionality of software. Therefore, code analysis is an essential step in order to find vulnerabilities and analyze malware that exploits them.

Code analysis is becoming increasingly difficult due to the high complexity of modern software. Malware, which is one of the most serious threats to the Internet security today, is a striking example of that. The level of sophistication employed by current malware continues to evolve significantly. For example, modern botnets use advanced cryptography, complex communication 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, 74] 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.

When dealing with third-party software or malware, security experts usually only have access to the compiled binary version of the code. Even when the source code of a program is available, analyzing the corresponding compiled code is important. This is mainly due to two reasons: First, the optimizations performed by compilers may alter the semantics of the source code, which creates a discrepancy between the source code of a program and its executable code. This phenomenon is referred to in the literature as *What You See Is Not What You eXecute (WYSINWYX)* [6]. The binary form of the program produced by the compiler is what actually gets executed by the processor. This means that it provides the actual ground truth about the program's functionality. Second, some vulnerabilities are specific to certain platforms [96]. That is, when the same code is compiled into two different platforms, the compiled binary code might be secure for one platform but vulnerable for the other one. This clearly shows the need for techniques to analyze binary code directly and also to be able to support multiple platforms.

Analyzing binary code is extremely challenging and time consuming. This mainly stems from the fact that during compilation almost all high-level abstractions available in the source code are removed by the compiler. These include abstractions like functions, variable names, date types, and control-flow constructs. The presence of these abstractions in the source code makes it more easily understandable by humans. However, they are not needed by the processor to correctly execute the code. *Decompilation* offers an attractive method to tackle this issue and assist malware analysis by enabling analyses to be performed on a high-level, more abstract form of the binary code. At a high level, decompilation consists of a collection of *abstraction recovery* mechanisms to recover high-level abstractions that are not readily available in the binary code. Both manual and automated analyses can then be performed on the decompiled program code, reducing both the time and effort required. Towards this goal, the research community has addressed principled methods for recovering high-level abstrac-

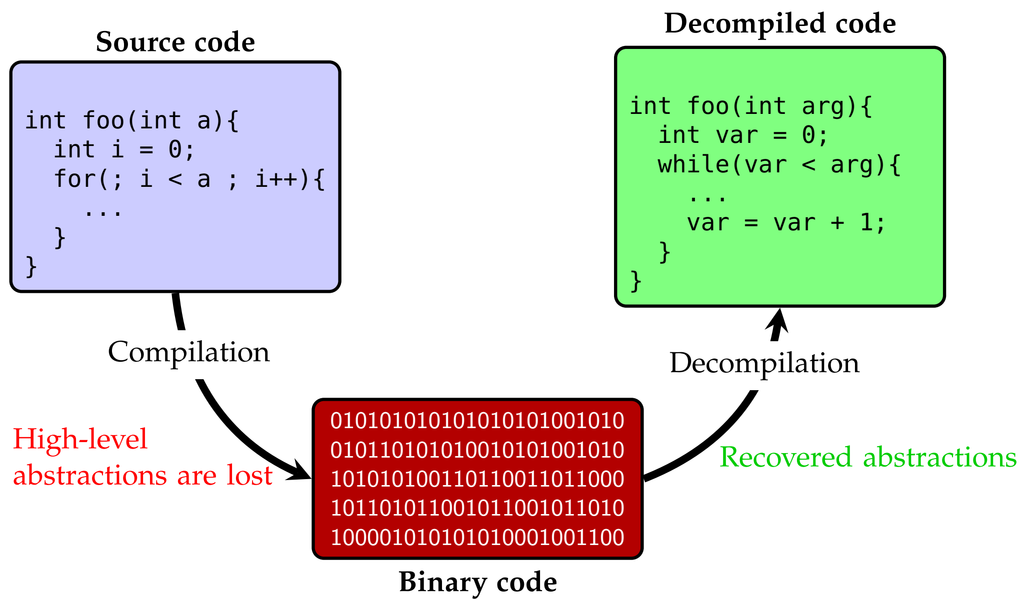


Figure 1.1: Compilation vs Decompilation

tions required for source code reconstruction. This includes approaches for recovering data types [59, 82, 58] and high-level control-flow structure (e.g., `if-then-else` constructs and `while` loops) from binary code [76, 103].

Decompilers that can reliably generate high-level readable 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. The faster and better the functionality and inner workings of a piece of malware is understood the faster effective detection techniques and countermeasures can be devised. This largely depends on the quality of the decompiled code, which can range from a very well readable code to a very poor version that still looks more like assembly code.

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 [9], apply taint tracking [23], or find vulnerabilities such as RICH [15], KINT [91], Chucky [106], Dowser [49], and the property graph approach [104, 105]. 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. [23] 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% [13]. This clearly illustrates the benefits of decompilation: it can bridge the gap between source code analyses which are efficient but rely on high-level abstractions and binary code where these abstractions are not available.

Binary code decompilation has a long history that dates back to the 1960's. A very good survey on the history of decompilation and several related areas can be found in Van Emmerik's PhD thesis [39, Chapter 5]. Another in-depth overview is available online [33]. Unfortunately, while significant advances have been made, state-of-the-art decompilers still create very complex code and do not focus on readability. The decompiled code can be so difficult to understand that security experts resort to analyzing the assembly code directly. Moreover, the evaluations of decompilers in previous works have never considered the human factor. That is, these evaluations have never performed user studies to test whether and to what extent the proposed decompilation techniques actually help human analysts. This is surprising since human analysts are a very important target for decompilation research.

Decompilation is a very wide topic and in this thesis we focus on improving the start of the art in certain dimensions. More specifically, we focus on improving the readability of the decompiled code in order to make it easier to understand by human analysts. In the following, we clearly state the research questions and the contributions of the thesis.

1.1 Research Questions

The focus of this work is on binary code decompilation. We seek to explore ways to improve the state of the art by improving the readability of the decompiled code in order to facilitate the process of manual analysis of binary code. Also, we target designing new methods to quantitatively and qualitatively evaluate the quality of the decompiled code and the benefits of decompilers for human analysts.

Research Question 1. *How can we produce structured code?*

Structured code uses high-level control constructs such as `if-then-else` and `while` loops to express the control flow inside a program. These constructs are easy for humans to understand and used by developers when writing code. A big issue with state-of-the-art

decompilers is that they produce code that contains a lot of `goto` statements representing arbitrary jumps in the code. These statements result in unstructured code that is hard to understand [37]. This research question seeks to find methods to reliably recover control-flow abstractions so that the decompiled code is structured and does not contain `goto` statements.

Research Question 2. *How can the decompiled code be put in a readable format to facilitate manual reverse engineering?*

Decompiled code is easier to understand if it can be written in a way that is similar to the manner a human developer would write code. However, during compilation the program structure is transformed into a more efficient but less readable form. This negatively impacts the decompiled version of the program recovered by the decompiler. While some compiler optimizations are not reversible, this research question involves following a human-centric approach to devise optimizations that transform the decompiled code into a more readable form.

Research Question 3. *How can we evaluate the effectiveness of decompiler for manual reverse engineering?*

Properly evaluating the quality of decompilers is essential to verify whether and to what extent the developed techniques can actually help in the analysis of binary code. Given that manual reverse engineering is one of the main motivations for decompiler research, it is surprising that previous works has never considered the human factor in their evaluation. This research question involves including the human factor in the evaluation of decompilation techniques to test how useful these techniques are for human analysts.

1.2 Thesis Contributions

The techniques presented in thesis are implemented in a academic decompiler called DREAM^{++} (Decompiler for Reverse Engineering and Analysis of Malware). In summary, the contributions of this thesis can be summarized as follows.

- **New control-flow structuring algorithm**. 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, as done by state-of-the-art decompilers. 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 refer to the version of our decompiler that implements our new control-flow structuring algorithm as DREAM.

- **Usability extensions to decompiler**. We present several semantics-preserving code transformations to simplify and improve the readability of decompiled code. The key insight of our approach is that the abstractions recovered during previous decompilation stages can be leveraged to devise powerful optimizations. To this end, we devise optimizations to simplify expressions and control-flow structure, remove redundancy, and give meaningful names to variables based on how they are used in code. We have implemented our techniques as extensions to our decompiler DREAM. The extended version is called DREAM^{++}.

- **Evaluation with malware analysis user study**. We include the *human factor* in a metric to evaluate how useful a decompiler is for manual analysis of binary code. Based on that, we conduct the *first* user study to evaluate the quality and usefulness of our approach for malware analysis. We conduct our study both with students trained in malware analysis as well as professional malware analysts. The results provide a statistically significant evidence that DREAM^{++} outperforms both the leading industry decompiler Hex-Rays and the original DREAM decompiler in the amount of tasks successfully analyzed.

1.3 Thesis Outline

This thesis consists of six chapters. The following four chapters describe the decompilation techniques developed during the work on this thesis. These chapters are based on papers published at peer-reviewed conferences. In the following, we describe the remaining chapters of the thesis.

Chapter 2. This chapter provides a high-level overview of the structure of our decompiler. The decompiler is based on a combination of existing works and novel techniques developed for

this thesis. Here, we discuss the existing techniques that we used and mention our extensions to them.

Chapter 3. This chapter describes our novel control-flow algorithm to produce fully-structured decompiled code. Here, we discuss our *pattern-independent structuring* and *semantics-preserving transformations* techniques designed to produce a `goto`-free output.

Chapter 4. This chapter describes our usability optimizations to make the decompiled code more readable and easier to understand. Here, we describe a combination of semantic-preserving transformations to simplify the code and increase readability. These optimizations are divided into three categories: expression simplification, control-flow simplification and semantics-aware naming.

Chapter 5. This chapter presents the evaluation of our techniques for malware analysis. Here, we describe the design of the first malware analysis user study and present the results of comparing our approach with the leading industry decompiler Hex-Rays.

Chapter 6. This chapter concludes the thesis by summarizing the main contributions and mentioning open directions for future research.

2 The DREAM^{++} Decompiler

Authors' Contributions

The work presented in this chapter is based on our paper published at the 8th IEEE International Conference on Malicious and Unwanted Software (MALWARE 2013) [101]. The chapter text is taken and adapted from this paper. The authors' contributions that are relevant to the contents of this chapter are as follows:

- **Khaled Yakdan** designed and implemented the system. Khaled also designed the main part of the evaluation and performed the evaluation.
- **Sebastian Eschweiler** provided valuable feedback during all phases of the work, and participated in designing the evaluation.
- **Elmar Padilla** was very helpful in discussing the work and provided tips for structuring the paper.

This chapter describes the overall architecture of the DREAM^{++} decompiler and discusses the design decisions. Designing an end-to-end decompiler is a large and challenging project. This stems from the fact that a wealth of high-level abstractions are removed by the compiler since they are not needed to correctly execute the code. As a result, several decompilation steps are needed to reconstruct those abstractions from binary code. As discussed in the previous Chapter, we focus in this work on improving the state of the art of binary code decompilation by focusing on certain decompilation steps. While building our decompiler, we rely on existing

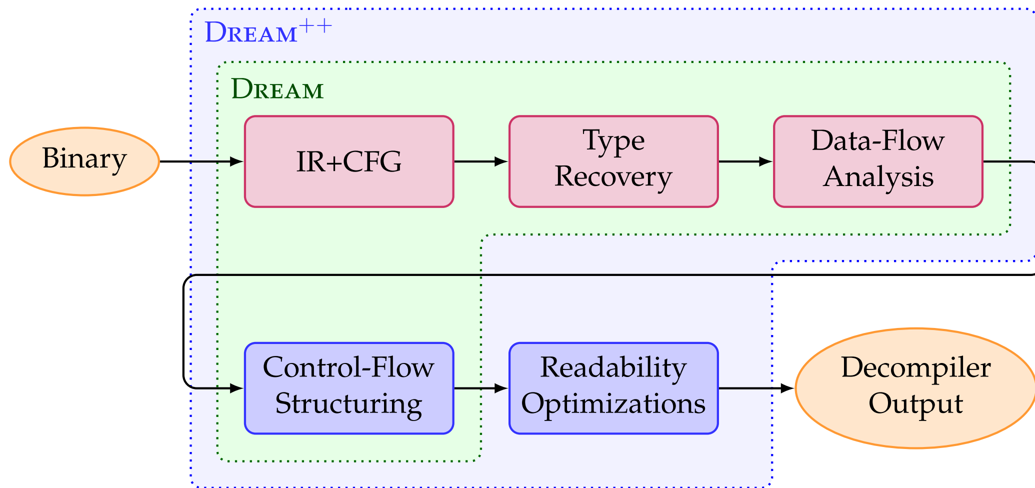


Figure 2.1: Overview of the DREAM^{++} decompiler. In the remainder of this thesis, we refer by DREAM to the version of the decompiler consisting of the first four steps (highlighted in green). The name DREAM^{++} refers to the complete decompiler (highlighted in blue).

tools and techniques for those steps where we don't make new contributions. In the following, we discuss our design and elaborate on the choices we made. We also give an overview on the existing techniques we used and the extensions we introduced to these techniques.

2.1 Overview

A high-level overview of the architecture of DREAM^{++} is given in Figure 2.1. The decompiler consists of several stages. First, the executable file is parsed, the program is loaded, and the code is disassembled. This stage builds the control-flow graph for all binary functions. For this step, we use the IDA Pro [51]. We also rely on IDA for the function interface recovery step, i.e., recovering the parameters and return values of functions. After that, the disassembled code is lifted into DREAM's intermediate representation (IR), which enables the subsequent analysis steps to be implemented independently from the input architecture. The used IR is based on the intermediate representation presented by Van Emmerik in his PhD thesis [39]. Should the binary be obfuscated tools such as [55] and [107] can be used to extract the binary code. We also rely on IDA for variable recovery. Currently, we only support translating x86 into the intermediate representation.

The second stage reconstruct the data types of recovered variables. Our implementation for this step is based on the concepts employed by TIE [58]. Recently, several more advanced approaches have been proposed to perform type analysis on binary code [73, 64]. Employing these approaches to DREAM^{++} is left for future work.

The third stage performs several data-flow analyses to remove several low-level details from the code and replace them with corresponding high-level representations. This stage consists of several standard code optimizations such as expression propagation and dead code elimination. We based our data-flow analysis on the work of Van Emmerik on his PhD thesis [39]. The main idea of Van Emmerik's thesis is that decompilation is easier on the Single Static Assignment (SSA) form of a program. Transforming our IR into SSA enables an efficient implementation of several data-flow analysis algorithms. These stages (marked in red in Figure 2.1) rely on existing work and will not be discussed in detail in the next chapters.

The four and fifths steps (marked in blue in Figure 2.1) are the core contributions of this thesis. The fourth stage is our new control-flow structuring algorithm to recover high-level control constructs from the CFG representation. The main idea of this algorithm is to be pattern independent. That is, unlike existing approaches, it does not rely on any predefined patterns that describe the shape of graphs corresponding to high-level control constructs. Rather, it relies on the semantics of those control constructs and can therefore produce structured `goto`-free code.

The fifth stage performs several code optimizations to improve the readability of the decompiled code. The main focus of these optimization is to transform the decompiled code into a semantically-equivalent representation that is easier to understand. We develop three categories of semantics-preserving code transformations to simplify the code and increase readability. These categories are expression simplification, control-flow simplification and semantics-aware naming.

We design our transformations to be semantics-preserving. However, some of these transformations rely the results of other analyses such as type analysis and function interface recovery. Errors in these analyses can lead to incorrect transformations. For example, if the function interface recovery step fails to detect that a called function has a parameter that is passed in a register, an assignment to this register before the function call may appear as dead code. This

will be the case if the assignment is only used by the caller to initialize the argument of the called function. Consequently, the assignment will be deleted by dead code elimination.

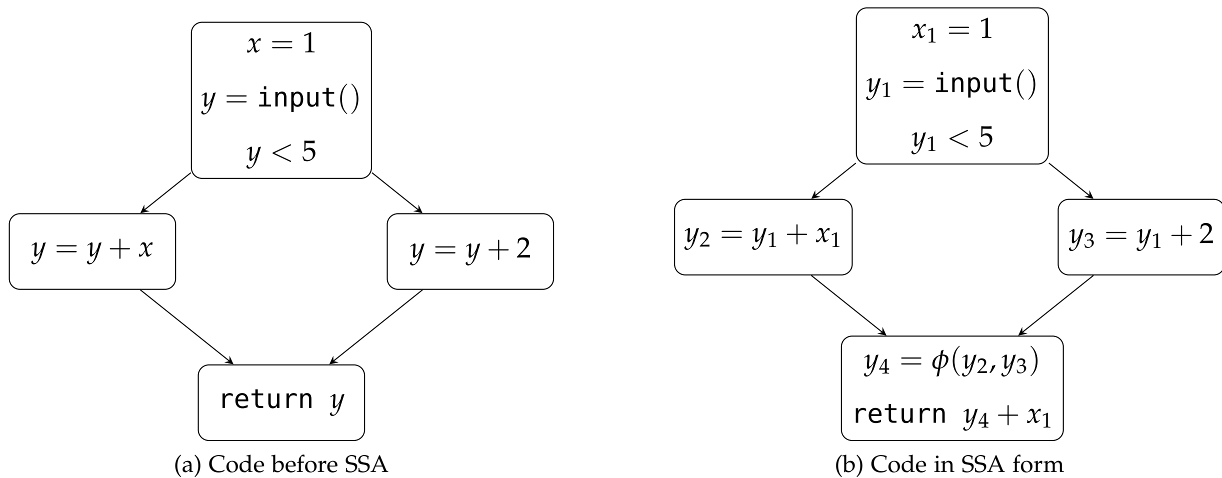
2.2 Naming Conventions

In Chapter 5, we conduct a user study to evaluate the impact of our readability optimizations (i.e., the fifth and last stage in Figure 2.1) on the quality of the decompiled code. For this, we compare the readability of the decompiled code produced by our decompiler with and without these optimizations. For easier readability, we gave the corresponding versions of our decompiler distinguishing names. More specifically, we refer to the version of the decompiler consisting of the first four steps by DREAM. This corresponds to the area highlighted in green in Figure 2.1. The enhanced and complete version of the decompiler that adds the readability optimizations is referred to in the thesis by DREAM^{++}.

2.3 Decompiler Design

An end-to-end decompiler is a complex project. Therefore, we opted for a modular design for DREAM^{++}. As can be seen in Figure 2.1, we split the abstraction recovery process into multiple steps where each step recovers a specific high-level abstraction. The output of each step is provided as an input to the next step. This modular design enables us to implement each step as an independent module that can be later used and updated independently from the rest of the system.

One important design decision was to lift the binary code into an *intermediate representation* (IR) before applying our analyses. This step is essential to easily support multiple architectures by providing an abstraction layer between the underlying architecture of the binary code and the analysis logic. CISC architectures are very complex and contain hundreds of instructions. For example, the x86 instruction set including all of its modern extensions contains more than 600 instructions. This makes it extremely challenging to correctly model and test the effects of all instructions in the analysis logic. Moreover, working directly with the binary code makes the analysis logic tightly coupled with the corresponding architecture. Consequently, a complete rewrite of the analysis code is required when wanting to support a new architecture.



(a) Code before SSA

(b) Code in SSA form

Figure 2.2: SSA Form.

2.4 Static Single Assignment

DREAM^{++} transforms the IR code into static single assignment (SSA) form before analyzing it. The SSA form is a code representation of code where each variable is only defined once in the program text. Figure 2.2 illustrates this idea by showing the control flow graph of a sample program before (Figure 2.2a) and the result of transforming it into the SSA form (Figure 2.2b). At a high level, each variable is assigned a index that is incremented with each new definition of the variable. To represent the different versions of a variable reaching join points in the control flow graph, the so called ϕ-functions are inserted. In SSA form, use-def chains are explicit and each contains a single element.

The SSA form makes it easier to write efficient code optimizations. Thus, it is used as the internal code representation such as the LLVM compiler infrastructure [87]. Van Emmerik has shown in his PhD thesis [39] that several data-flow analyses for decompilation can be better implemented with SSA form.

2.4.1 Transforming code into SSA

There exist several algorithms to transform code into SSA form. These algorithms differ in the number of ϕ-functions they insert into the code. We use the SSA generation algorithm proposed by Cytron et al. [31]. The algorithm efficiently constructs the SSA form based on

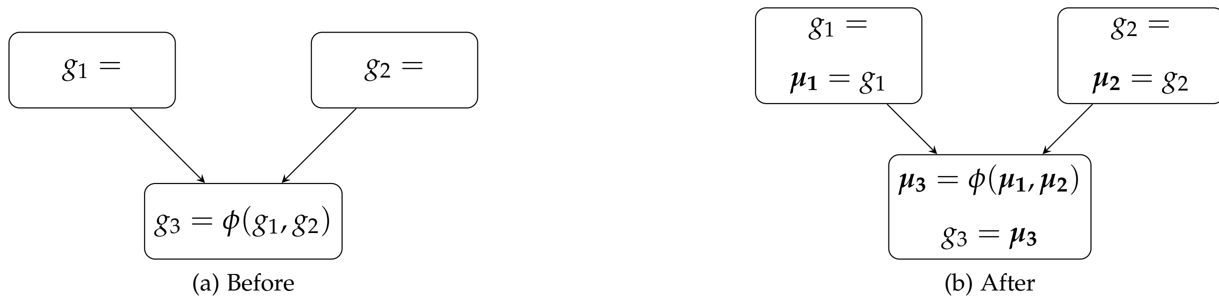


Figure 2.3: Handling global variables for SSA back translation.

the *dominance frontiers* graph property [3] and computes the minimal SSA form in terms of inserted ϕ-functions. As a by-product of applying this algorithm, we construct two data sets linking the definitions and uses of all variables in the program.

- **definitionsMap** is a hash table that allows fast access to the instruction that defines each variable.
- **usesMap** is a hash table that allows fast access to the set of instructions that use each variable.

2.4.2 Applying SSA to Memory

Pointer aliasing complicates the construction of SSA. The fact that different memory expressions may refer to a single location effectively means that the same memory location can be accessed using different *names*. Reasoning about aliases is important to correctly add indices to memory expressions. To this end, we apply the approach proposed by Van Emmerik [39]. At a high level, this approach consists of two steps: first, non-memory locations are translated into the SSA form and expression propagation is only applied to these locations. Second, subscripting and propagating memory locations is delayed until after propagation of non-memory locations is done.

2.4.3 SSA Back Translation

The optimized IR is transformed out of SSA before code generation. This involves removing ϕ-functions since they do not belong to any high-level language. Originally, all variables in

a ϕ-function stem from the same variable, and removing the ϕ-function means choosing one representative for them. This is only possible if the live ranges of variables in the ϕ-function do not interfere. That is, two variables x_1, x_2 can be represented in the program text using one representative x if they are not mutually live at any point in the program. A variable x is live at a point p of the program if there exists an execution path from the definition of x to p and a path from p to a use of x. Several approaches propose removing interferences by inserting copy statements. DREAM^{++} uses Sreedhar's algorithm [84] since it produces fewer copies in general [75]. However, the algorithm has the drawback that it does not distinguish between global and local variables when inserting copy statements. This may lead to renaming some global variables participating in a ϕ-function which changes the semantics of input code.

We solve this problem by breaking the live ranges of interfering global variables participating in a ϕ-function. Figure 2.3 shows an example of this case. If the live ranges of global variables g_1, g_2 and g_3 shown in Figure 2.3a interfere, copy instructions using local variables μ_1, μ_2 and μ_3 are inserted as illustrated in Figure 2.3b. This breaks the live ranges of global variables and the ϕ function now contains only local variables that can be renamed without any constraints. At this point the subscripts of global variables can be safely removed.

2.5 Type Analysis

Type analysis addresses the problem of assigning types to variables. For this, we base our type analysis on TIE [58] where we start from a set of *type sinks*, i.e., locations in code where the types of variables are directly known. Then, the types of remaining variables are resolved using a set of type inference rules. Binary code contains instructions that take operands of fixed and known types. For example, in the x86 instruction set these instructions include:

- *string instructions* which deal with pointers. This set includes `movs, lods, stos, cmps`.
- *integer instructions* which deal with integer values. This set includes `mul, div` , etc.
- *floating-point instructions* which operate on floating-point numbers. This set includes `fadd, fdiv` , etc.

- *Standard library calls* which have a well-defined and publicly known API. Here, the types of parameters and return values can be easily acquired from the definition of the function interface. For example, the single argument of `strlen` must be of the `char*` type.

DREAM^{++} uses these instructions as reliable starting points for performing type unification. That is, it uses a set of inference rules to deduce the types of remaining variables based on how they are used in code. For simple types, an assignment of the form $x = y$ reveals that both variables have compatible types. For addition of the form $x = y + z$, knowing the type of two operands leads to identifying the type of the third operand. For example, if y and z are integers, then x is also an integer. Recognized types are propagated using the properties of SSA which allows to efficiently get, for each variable, the defining instruction and the list of using instructions.

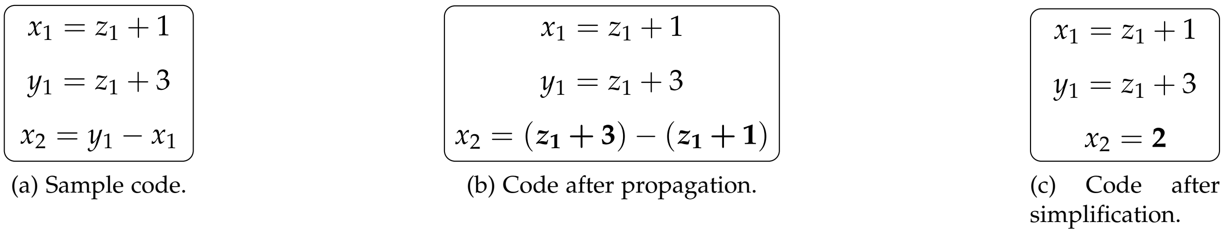
2.6 Data Flow Analysis

In this section, we describe and discuss the third analysis phase, as depicted in Figure 2.1. Here, we perform several data-flow analyses to reconstruct high-level statements corresponding to the input code.

2.6.1 Expression Propagation

Machine code instructions can only represent simple expressions directly. Moreover, instruction sets impose restrictions on the number and type of operands that can be used in these instructions. Therefore, compilers break high-level expressions into a sequence of simpler subexpressions that can be represented by machine instructions. Expression propagation reverses this process by propagating variable definitions into the instructions using them. Figure 2.4a shows a sample code of three instructions. Propagating the value of variables b_1 and c_1 into the third instruction results in the code in Figure 2.4b.

This propagation may result in superfluously complex expressions. After propagation, DREAM^{++} performs a mathematical simplification phase in order to transform expressions into equivalent but simpler forms. This phase is analogous to that of common compilers and its effect is illustrated in Figure 2.4c where the third instruction is simplified.



(a) Sample code.

(b) Code after propagation.

(c) Code after simplification.

Figure 2.4: Expression Propagation.

2.6.2 Dead Code Elimination

A variable is *dead* if it is defined by a given instruction but not used afterwards. If the defining instruction only defines the dead variable, it can be safely removed. Dead code is common after expression propagation as illustrated in Figure 2.4c where variables b_1 and c_1 become dead. Checking if a variable v is dead can be performed in constant time using the **usesMap** data structure.

$$v \text{ is dead} \iff \textbf{usesMap}\,[v.name]\,[v.subscript] = \emptyset \tag{2.1}$$

Certain types of variables cannot be removed even if they satisfy statement 2.1. This particularly concerns global variables, i.e., memory locations in data section. Such variables can be accessed and modified by all functions of the program. Therefore, DREAM^{++} does not eliminate global variables. Combining expression propagation and dead code elimination enables DREAM^{++} to overcome obfuscation techniques that insert junk code and semantic NOPs.

Trivial ϕ chains. Expression propagation may result in situations where some variables are not effectively used but cannot be deleted because they do not satisfy the condition in statement 2.1. This is particularly relevant for variables participating in ϕ functions. Figure 2.5 shows an example consisting of variables x_1, x_2 and x_3. None of these variables is dead because there exists a circular dependence between them. Moreover, translating this code out of the SSA form will result in useless assignments of the form $x = x$. We call such a set *trivial ϕ chain* and denote it by ϕ_t. It is defined as the set of variables that are only either used in

1. a ϕ function of variables in ϕ_t; or

2. a copy assignment of the form $a_i = a_j$ defining a variable contained in ϕ_t.

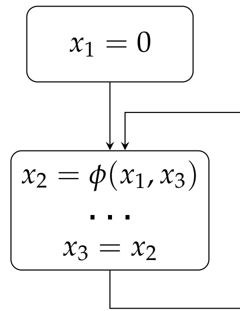


Figure 2.5: A trivial ϕ function

All variables in ϕ_t can be safely removed without changing the semantics of code. The scope of these chains may cover several ϕ functions. Removing trivial ϕ chains may lead to other variables becoming dead. Therefore, the dead code elimination algorithm is applied iteratively until no trivial ϕ chain is found.

2.6.3 Detection of Function Parameters

Function parameters are those variables used before being defined in the body of the function. They are defined by a former function in the call chain. Therefore, a parameter is live at the function's entry. Global memory locations can be directly accessed by all functions, hence, they do not conform to the notion of parameters being locally defined in the body of the caller. DREAM^{++} constructs function parameters based on the following equation

$$Parameters\,(f) = \{p \mid p \in LiveIn\,(B_0) \text{ and } p \in Candidates\}$$

$LiveIn\,(B_0)$ is the set of live variables on the function's entry and *Candidates* is the set of non-global variables.

After the data-flow analysis phase, most machine-specific details are replaced by high-level representations. Tested flags are replaced by equivalent conditions. Functions calls are presented with their actual parameters. The optimized IR contains high-level expressions and is smaller than the input code because dead code resulted from expression propagation or semantic NOPs is removed.

2.7 Summary

In this chapter, we described the overall architecture of our decompiler and discussed the main design decisions we made. We also described the decompilation steps in DREAM^{++} that are based on existing works. For these steps, we also described the extensions and improvements we made. The next chapter describes our first main contribution: a novel control-flow structuring algorithm that produces fully structured decompiled code without `goto` statements.

3 Control-Flow Structuring

Authors' Contributions

The work presented in this chapter is based on our paper published at the 22nd Network and Distributed System Security Symposium (NDSS 2015) [103]. The chapter text is taken and adapted from this paper. The authors' contributions that are relevant to the contents of this chapter are as follows:

- **Khaled Yakdan** had the main idea, designed and implemented the system, designed and conducted the evaluation.
- **Sebastian Eschweiler** was very helpful in discussing the idea.
- **Elmar Padilla** provided valuable feedback to the idea and evaluation.
- **Matthew Smith** participated in designing the part of the evaluation regarding comparing DREAM with other decompilers. Matthew also gave valuable insights and guidance to the structure of the paper.

This chapter of the thesis focuses on the recovery of control-flow abstractions from binary code. This process, denoted in the literature as *control-flow structuring*, means taking the control-flow graph of a binary function and recovering the corresponding high-level control flow constructs (e.g., `if-then-else` or `while` loops) from the graph representation. 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 binary code decompilers such as Hex-Rays [47] and Phoenix [76] rely on structural analysis for this step, which is a pattern-matching approach over the control flow graph, to recover high-level 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 chapter, we present 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. These techniques make DREAM the first decompiler to offer a `goto`-free output. 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.

3.1 Introduction

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 [47] and Phoenix [76] employ structural analysis [62, 77] (§3.2.1) 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 [37] and helps scale program analysis [62]. The 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 [76]. 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 chapter, 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[1]. 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. To avoid unnecessarily increasing the size of the decompiled code and thus negatively impacting its readability, we designed these transformations so that they do not involve duplicating any code blocks. This is an important feature distinguishing our algorithm from other approaches that use node splitting to handle unstructured control flow [1].

[1] This is the case even when the original source code contains `goto` statements.

We have implemented our algorithm in a decompiler called DREAM [2] (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.

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

3.2 Background & Problem Definition

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

[2]Check Section 2.2 and Figure 2.1 for information about the naming of the different versions of our decompiler.

```
1  int foo(){
2   int i = 0;
3    while(i < MAX){
4      print(i);
5      i = i + 1;
6    }
7    return i;
8  }
```

Figure 3.1: Exemplary code sample

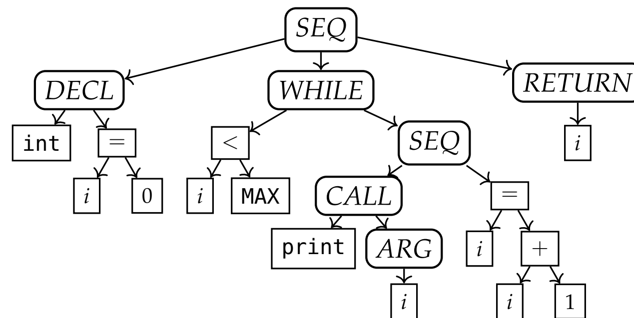


Figure 3.2: Abstract Syntax Tree

3.2.1 Background

We start by briefly discussing two classic representations of code used throughout the chapter 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 3.1.

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 3.2 showing an abstract syntax tree for the code sample given in Figure 3.1.

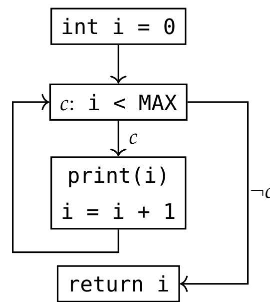


Figure 3.3: Control Flow Graph

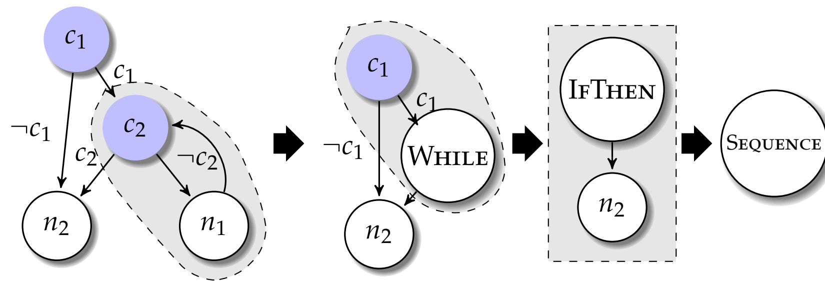


Figure 3.4: Example of structural analysis.

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, denoted by $\tau(n_s, n_t)$, is assigned to each edge $(n_s, n_t) \in E$ to represent the logical predicate that must be satisfied so that control is transferred along this 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 3.3 shows the CFG for the code sample given in Figure 3.1.

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`

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 $Seq\,[n_1, \ldots, n_k]$.
$Cond\left[c, n_t, n_f\right]$	*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 $\tau \in \{\tau_{\mathbf{while}}, \tau_{\mathbf{dowhile}}, \tau_{\mathbf{endless}}\}$ 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 $v \in V_i$

Table 3.1: AST nodes that represent high-level control constructs

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 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 3.4 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 $\textbf{if}\,(c_1)\,\{\textbf{while}\,(\neg c_2)\,\{n_1\}\}\,;n_2$

3.2.2 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 thesis, 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` statement. 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 3.1. The table does not contain `for` loops since these are not needed at this stage of the process. `for` loops are recovered during optimizations described in Chapter 4. We allow `break` statements to represent early exits from loops. Differently from `goto` statements, `break` statements cause control to be transferred to the loop successor and not to arbitrary locations in code.

3.2.3 Running Example

As an example illustrating a sample control flow graph and running throughout this chapter, we consider the CFG shown in Figure 3.5. 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 without `goto` statements. These three regions were chosen such that the difficulty for traditional structuring algorithms increases from R_1 to R_3. The right hand side of Figure 3.6 shows how the structuring algorithm of Hex-Rays structures this CFG. For comparison, the left hand side shows how our algorithm 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.

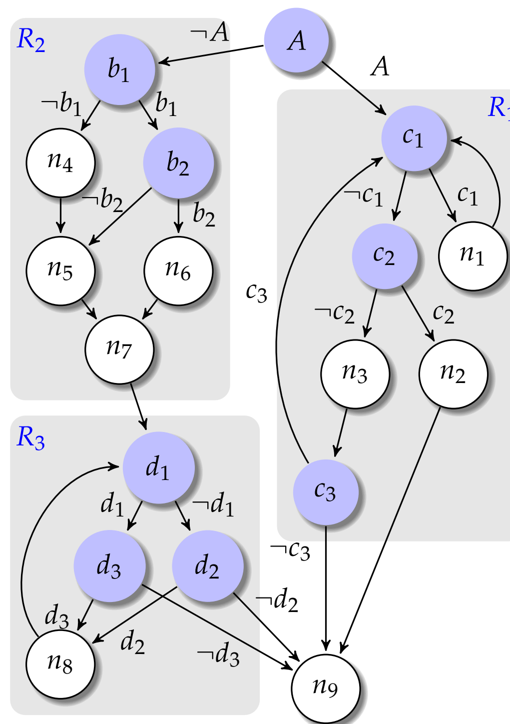


Figure 3.5: Running example. Sample CFG that contains three regions: a `while` loop with a break statement (R_1), a proper interval (R_2), and a loop with unstructured condition (R_3).

3.3 Approach Overview

At a high level, our approach comprises two phases: *pattern-independent structuring*, and *semantics-preserving transformations*. The algorithm recovers control-flow abstractions and computes the corresponding AST. 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 (§3.4.2). 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

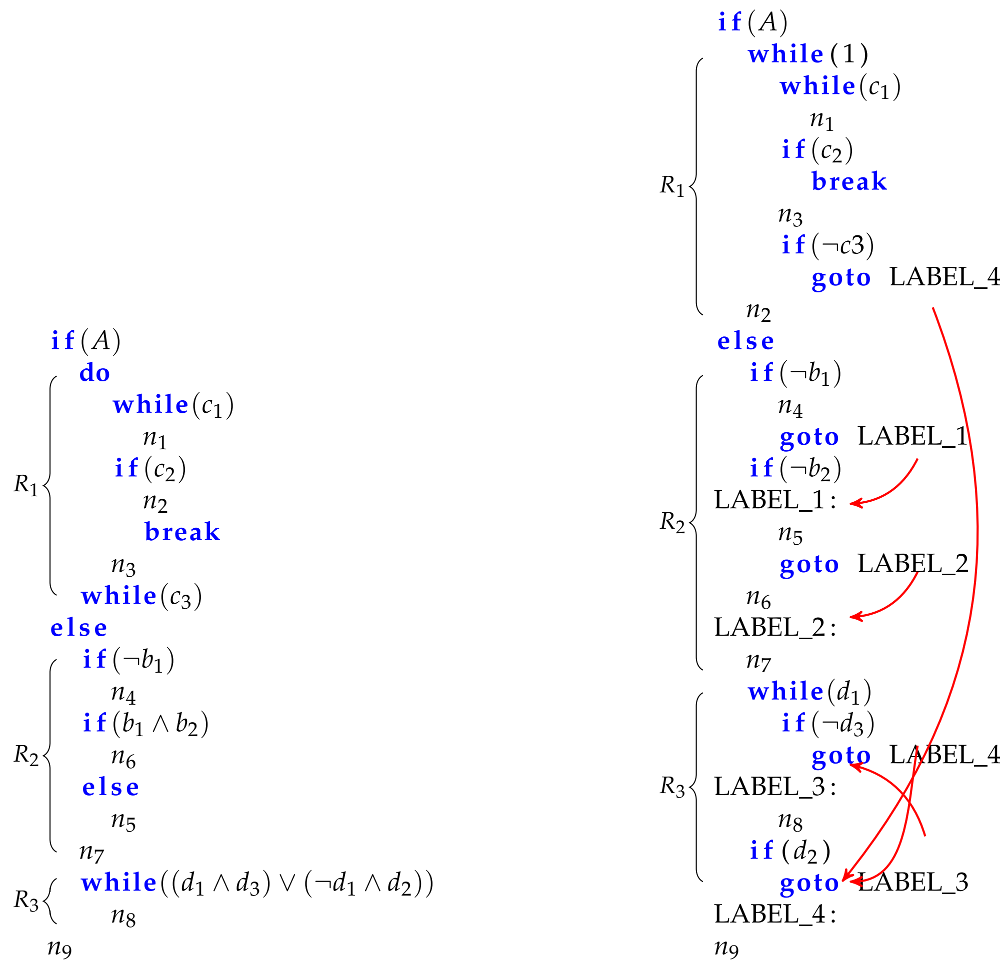


Figure 3.6: Decompiled code generated by DREAM (left) and by Hex-Rays (right). The arrows represent the jumps realized by `goto` statements.

(§3.5) and structure the resulting region (§3.4.3). 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 (§3.4.1), 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 infers 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, starting from the following initial loop structure **while** (1) $\{$**if** $(\neg c)$ $\{$**break;**$\}$ `body;` $\}$, we can refine this structure into a while loop **while** (c) $\{$`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.

3.4 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.

3.4.1 Reaching Condition

In this section, we discuss our algorithm to find the condition that takes the control flow from a given starting node n_s (also called source node) to a given end node n_e (also called sink node) in the CFG, denoted by *reaching condition* $c_r(n_s, n_e)$. This means that when at node n_s, control flow reaches n_e if and only if the reaching condition $c_r(n_s, n_e)$ is true. This step is essential for our pattern-independent structuring and guarantees the semantics-preserving property of our transformations (§3.5).

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:     else if n_t ∈ S_G ∧ n_t ∉ DfsStack then
11:         AddPath(S_G, DfsStack)
12:     RemoveVisitedNodes()
```

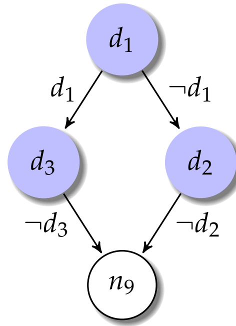


Figure 3.7: $S_G(d_1, n_9)$ of the running example

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. A simple path in the graph G is a path which does not have repeating vertices. 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 3.7 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.

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:

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

where `Preds` (n) returns the immediate predecessors of node n and $\tau(v, n)$ is the tag assigned to edge (v, n), which represents the logical predicate that must be satisfied so that control is transferred along this edge. Then, we simplify the logical expressions.

3.4.2 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 [30, 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 $\textbf{if}\,(c_r\,(h, n_1))\,\{n_1\}\,; \ldots ; \textbf{if}\,(c_r\,(h, n_k))\,\{n_k\}$. Obviously, the initial AST is not optimal. For example, nodes with complementary conditions are represented as two `if-then` constructs $\textbf{if}\,(c)\,\{n_t\}\,\textbf{if}\,(\neg c)\,\{n_f\}$ and not as one `if-then-else` construct $\textbf{if}\,(c)\,\{n_t\}\,\textbf{else}\,\{n_f\}$. Therefore, in the second phase, we iteratively refine the initial AST to find a concise high-level representation of control flow inside the region.

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 $c \land R$. Similarly, nodes belonging to the false branch have reaching conditions of the form $\neg c \land R$. 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 3.8. 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 $b_1 \land b_2$ since nodes n_5 and n_6 have complementary conditions. This results in three clusters $V_{b_1 \land b_2} = \{n_6\}$, $V_{\neg(b_1 \land b_2)} = \{n_5\}$, and $V_r = \{n_4\}$ and leads to creating a condition node. At

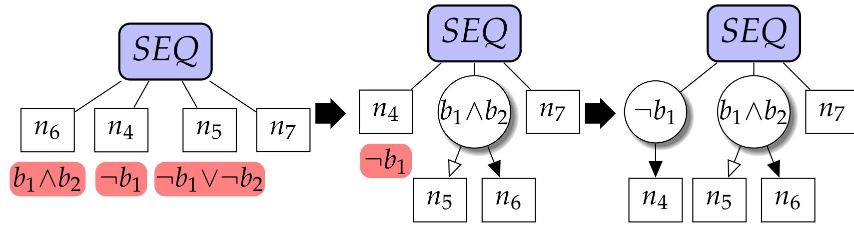


Figure 3.8: Development of the initial AST when structuring the region R_2 in the running example. The initial AST (left) is refined by a condition-based refinement with respect to condition $b_1 \wedge b_2$ (middle). Finally, a condition node is created for n_4 (right).

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 [24]. 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` 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 u (this would cause this node to be before u 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.

3.4.3 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.

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 (§3.5.1). 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.

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 3.5.2. For instance, when structuring region R_1 in our running example (Figure 3.5), 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: $N_{new} \leftarrow N_{succ}$
2: **while** $|N_{succ}| > 1 \land N_{new} \neq \emptyset$ **do**
3: $\quad N_{new} \leftarrow \emptyset$
4: $\quad$ **for all** $n \in N_{succ}$ **do**
5: $\quad\quad$ **if** $\textsc{Preds}(n) \subseteq N_{loop}$ **then**
6: $\quad\quad\quad N_{loop} \leftarrow N_{loop} \cup \{n\}$
7: $\quad\quad\quad N_{succ} \leftarrow N_{succ} \setminus \{n\}$
8: $\quad\quad\quad N_{new} \leftarrow N_{new} \cup \left\{u : u \in \left[\textsc{Succs}(n) \setminus N_{loop}\right] \land \textsc{Dom}(n_h, u)\right\}$
9: $\quad N_{succ} \leftarrow N_{succ} \cup N_{new}$

Phoenix [76] 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 the 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.

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

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

$$\frac{n_\ell = Loop\left[\tau_{\textbf{endless}}, -, Seq\left[n_i\right]^{i\in 1..k}\right] \quad n_k = \mathcal{B}_r^c}{n_\ell \rightsquigarrow Loop\left[\tau_{\textbf{dowhile}}, \neg c, Seq\left[n_i\right]^{i\in 1..k-1}\right]}\textsc{DoWhile}$$

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

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

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

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

Figure 3.9: Loop structuring rules. The input to the rules is a loop node n_ℓ.

the loop body is an acyclic region that we structure as explained in §3.4.2. Next, we represent the loop as endless loop with the computed body's AST, i.e., $n_\ell = Loop\left[\tau_{\mathsf{endless}}, -, n_b\right]$. 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:

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

The top of the inference rule bar contains the premises $P_1, P_2, \ldots, P_n$. If all premises are satisfied, then we can conclude the statement below the bar C. Figure 3.9 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

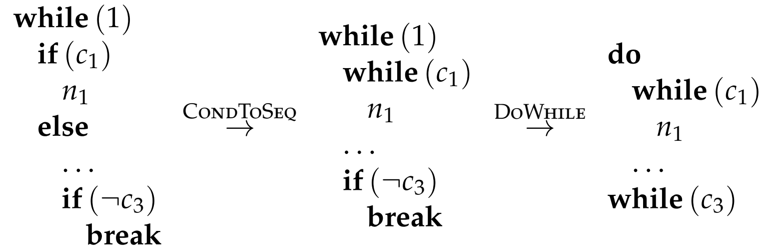


Figure 3.10: Example of loop type inference of region R_1.

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 3.10. 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;**} ...}. The loop's body starts with a conditional break and hence is structured according to the While rule into **while** $((d_1 \wedge d_3) \vee (\neg d_1 \wedge d_2))$ {...}. We wrote a small function that produces the same CFG as the region R_3 and decompiled it with Dream and Hex-Rays. Figure 3.12 shows that our approach correctly found the loop type and continuation condition. In comparison, Hex-Rays

```
signed int __cdecl loop(signed int a1)
{
  signed int v2; // [sp+1Ch] [bp-Ch]@1

  v2 = 0;
  while ( a1 > 1 ){
    if ( v2 > 10 )
      goto LABEL_7;
LABEL_6:
    printf("inside_loop");
    ++v2;
    --a1;
  }
  if ( v2 <= 100 )
    goto LABEL_6;
LABEL_7:
  printf("loop_terminated");
  return v2;
}
```

Figure 3.11: Decompiled code generated by Hex-Rays.

```
int loop(int a){
  int b = 0;
  while((a <= 1 && b <= 100)||(a > 1 && b <= 10)){
     printf("inside_loop");
     ++b;
     --a;
  }
  printf("loop_terminated");
  return b;
}
```

Figure 3.12: Decompiled code generated by Dream.

produced unstructured code with two `goto` statements as shown in Figure 3.11; one `goto` statement jumps outside the loop and the other one jumps back in the loop.

3.4.4 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 3.8 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.

```
1  int foo(){
2    ...
3    p = &v
4    ...
5    use1(v)
6    ...
7    *p = ...
8    ...
9    use2(v)
10   ...
11 }
```

Figure 3.13: Aliasing example

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 may be 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 in an expression such as `p = &v`). We do not limit the last check to a path between the uses. That is, we check if anywhere in the function the variable is involved in a reference expression. This is necessary to handle cases where the address of a variable taken before the uses and then its value is changed using the resulting pointer as illustrated by the example shown in Figure 3.13. 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.

3.4.5 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.

3.5 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 apply the proposed transformations on those regions. An important feature of these transformations is that they do not involve code duplication. This avoid increasing the size of decompiled output and making it harder to understand. 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.

3.5.1 Restructuring Abnormal Entries

The high-level approach to structuring abnormal entries (cf. 3.4.3) is illustrated in Figure 3.14. The underlying idea is to insert a *structuring* variable (i in Figure 3.14) 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 an additional term representing the abnormal entries **while** $(c \lor i \neq 0)$ $\{\ldots\}$.

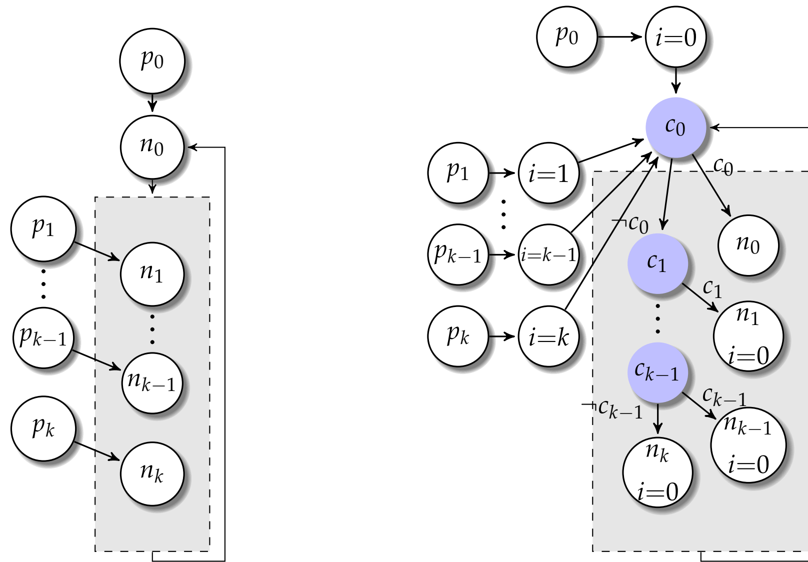


Figure 3.14: Transforming abnormal entries: multi-entry loops (left) are transformed into semantically equivalent single-entry loops (right). Tags c_n represent the logical predicates $i = n$.

3.5.2 Restructuring Abnormal Exits

The high-level approach to structuring abnormal exits (cf. 3.4.3) is illustrated in Figure 3.15. 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 (the red edges in Figure 3.15). 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. As an example illustrating the fact that control flow is preserved by our transformation, consider Figure 3.15 where control flow exits the original loop through edge (n_1, s_1) (left graph). In the transformed CFG, the control flow follows the redirected edge (n_1, c_1) where the condition c_1 is checked. This condition distinguishes this exit and only evaluates to `true` when the loop is exited though (n_1, s_1). As a result, control flow then follows the `true` branch (c_1, s_1) leading to the same target as in the original CFG. A more concrete example is provided in Section 3.6 provides a concrete example on the result of applying this transformation to the case of overlapping loops. This transformation does not change the internal loop structure and thus keep the loop exit condition intact. It merely makes sure that the restructured loop has a single successor so that it can be structured as discussed in Section 3.4.3. Note that the added conditions are computed inside the loop and

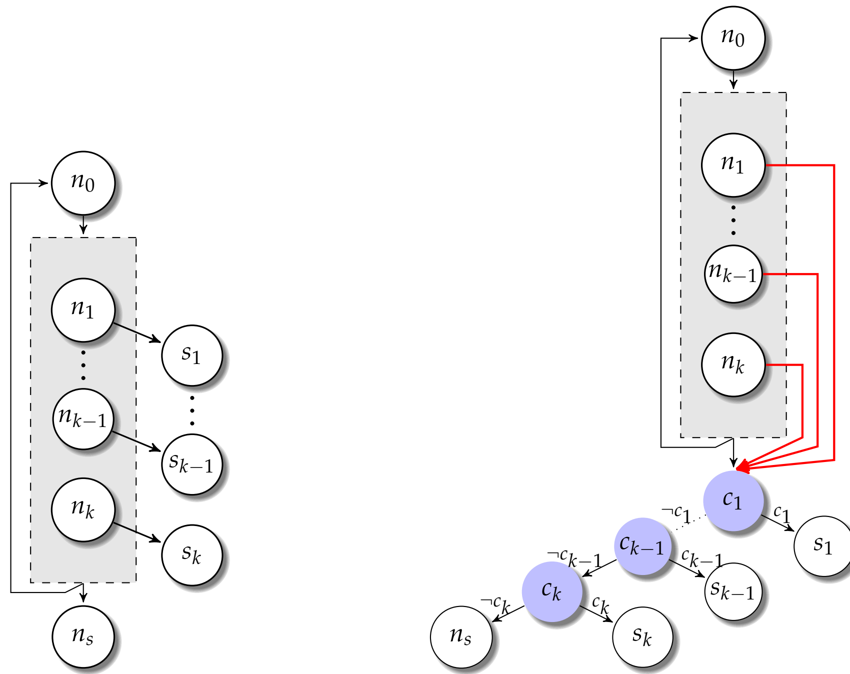


Figure 3.15: Transforming abnormal exits: loops with multiple successors (left) are transformed into semantically equivalent single-successor loops (right).

thus when evaluated after the loop, they correspond to the last loop iteration representing the state that caused the loop to be exited along one of the exit edges.

Computing exit conditions. 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 $c_r(n_{\mathsf{ncd}}, u)$ to the target nodes of edges in E_{out}.

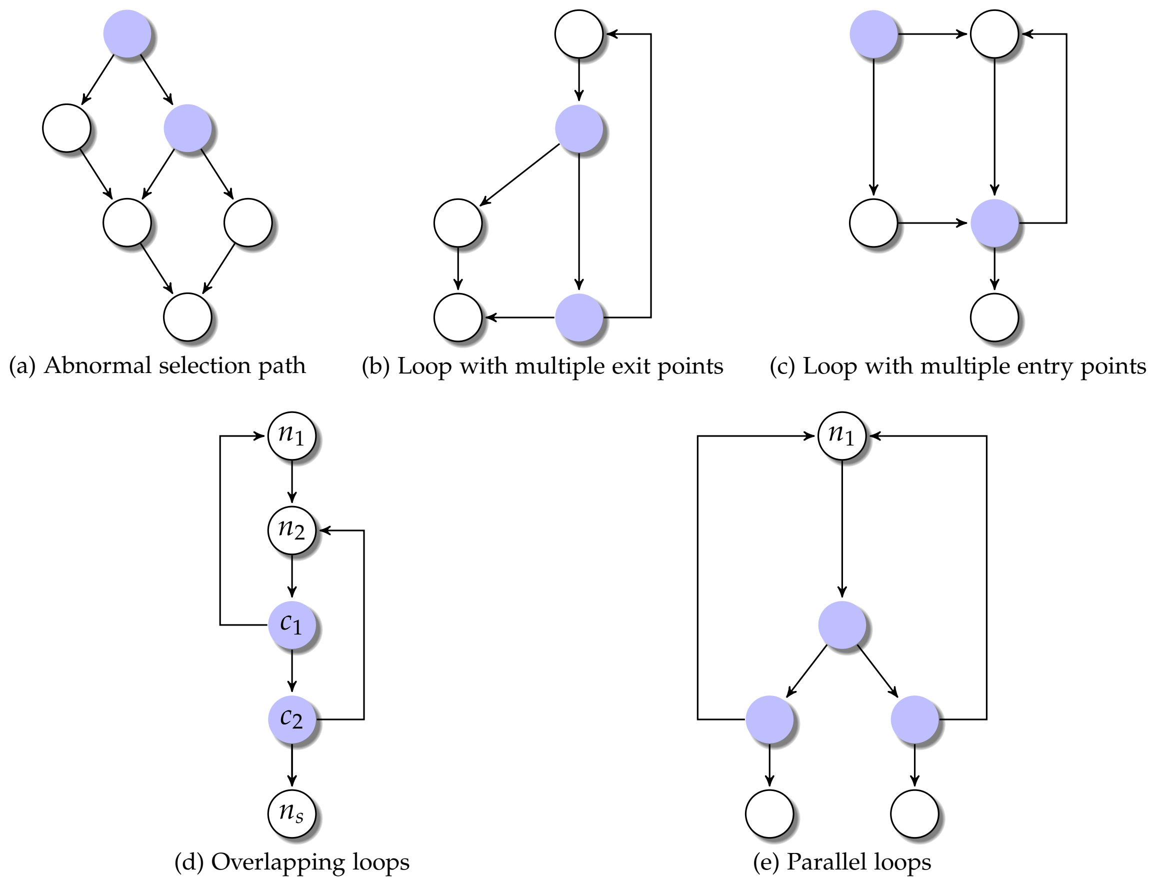
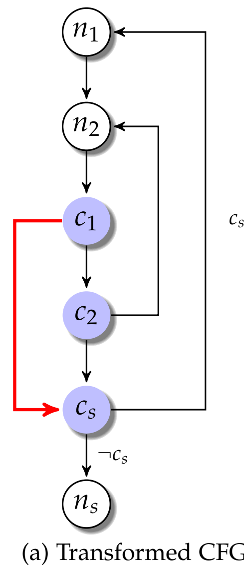


Figure 3.16: The five basic structures which cause unstructured flow diagrams.

3.6 `goto`-Free Output

In this section, we show that our algorithm produces `goto`-free output. Willians *et al.* has shown that there exist five basic structures that lead to unstructured flow diagrams [93]. Here, unstructured flow diagrams are defined as those that cannot be decomposed in terms of three patterns: sequence, selection (i.e., `if-then-else`), and repetition (i.e., `do-while`). These structures are shown in Figure 3.16. He also shows that *given transformations that convert each of these five structures to structured form, then any flow diagram can be transformed to a structured flow diagram*. Here, we show that our algorithm produces structured control flow for all of these five diagrams and thus produces `goto`-free code.



(a) Transformed CFG

```
do
   n1
   do
      n2
      if(c1)
            break
   while(c2)
while(c1)
ns
```

(b) Resulting code

Figure 3.17: Structured form of overlapping loops shown in Figure 3.16d. In this case, the condition that causes control flow to exit the loop through edge (c_1, n_1) is $c_s = c_1$.

- *Abnormal selection path* (Figure 3.16a). This corresponds to the region R_2 in our running example (Figure 3.5). Our algorithm produces structured code for this region as shown in Figure 3.6.

- *Loop with multiple exit points* (Figure 3.16b). The transformations presented in Section 3.5.2 transform these loops into a structured form.

- *Loop with multiple entry points* (Figure 3.16c). The transformations presented in Section 3.5.1 transform these loops into a structured form.

- *Overlapping loops* (Figure 3.16d). This can be reduced to the case of multi-exit loop. The loop headed at n_2 is a multi-exit loop and the loop headed at n_1 is a multi-entry loops. Our algorithm will start by structuring the inner loop (headed at n_2). The multi-exit loop transformation results in the structure shown in Figure 3.17a. Here, we have two nested loops with the red edge represented as a `break` statements. For this transformed CFG, our algorithm produces the decompiled code is shown in Figure 3.17b.

- *Parallel loops* (Figure 3.16e). With a single header node (n_1), parallel loops are special case of multi-exit loop and can thus be transformed into a structured form as shown in Section 3.5.2.

Since our algorithm produces structured code for all the five structures that cause unstructuredness, it thus can produce `goto`-free code.

Summary. At this point we transformed the CFG to an AST containing only high-level control constructs and no `goto` statements. The next section presents the evaluation of our approach.

3.7 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.* [76]. This evaluation used the GNU `coreutils` to evaluate the quality of the decompilation results. We compared our results with Phoenix [76] and Hex-Rays [47]. 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* [24], DISC [56], REC [71], and Boomerang [39] 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.

3.7.1 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. Here, we focus on testing the correctness of our control-flow structuring algorithm.

- **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 [37] and helps scale program analysis [62]. 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.

These metrics can be measured automatically and can capture important code properties that impact readability. Our experience suggests that structured and compact code tends to be more readable and easier to understand than unstructured and lengthy code. However, it is conceivable that in some cases less compact code or code with `goto`s may be easier to understand. For this reason, we evaluate the our approach with a user study involving real-world code samples (Chapter 5).

3.7.2 Experiment Setup & Results

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

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 [76] 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.

The goal in this experiment is to evaluate the correctness of our control-flow structuring algorithm independently from the other decompilation steps. That is, if we start from a correct CFG of the program, the goal is to check if our approach may produce incorrect code. For this, we computed the CFG for each function in the `coreutils` source code and provided it as input to the algorithm. Then, we replaced the original functions with the algorithm output, compiled the restructured `coreutils` source code, and finally executed the tests. Using the CFGs constructed from the source code enables us to isolate the source of errors and attribute any failure in the tests to our algorithm. Since the original source code passes the tests, building the CFG from it and then passing them to our algorithm means that any failure in the test must be caused by our approach. On the other hand, starting from binary code means that there can be errors caused by other decompilation phases, e.g., type analysis which lead to a test failure. Moreover, we would have less coverage since not all decompiled functions will be recompilable due to errors in other decompilation phases. In the Phoenix experiments, the authors attributed most correctness errors to the underlying type recovery component they used, TIE [58].

We used *joern* [104] 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

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 3.2: Correctness results.

assumption of correct parsing, we can attribute the failure of any test on the restructured functions to the structuring algorithm. We used the source files produced by the C-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.

Correctness Results

Table 3.2 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. These errors were mainly caused by issues in CFG construction, which lead to erroneous CFGs. We reported these issues to the authors of *joern* and they will be fixed in later releases. 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. In the future, we plan to evaluate the correctness of our algorithm on the CFGs recovered from binary code once advanced type inference approaches such as [64] have been implemented in our decompiler.

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

Structuredness and Compactness Experiment

We tested and compared 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

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 3.3: Structuredness and compactness results. For the `coreutils` 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.

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 [76] 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 [104] 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*.

Structuredness & Compactness Results

Table 3.3 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 structuring algorithm. 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 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.

Malware Analysis

For our malware analysis, we picked three real-world samples from three malware families: ZeusP2P, Cridex, and SpyEye. The results for decompiling those malware samples shown in Table 3.3 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.

3.8 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 [2, 27] deconstructs the CFG into nested regions called *intervals*. The nesting structure of these regions helps to speed up data-flow analysis. Structural analysis [77] 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.* [40] 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 [41] and [94]. 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 3.2.2.

Decompilers. Cifuentes laid the foundations for modern decompilers. In her PhD thesis [24], 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* [25] 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 [39]. 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* [81]. Although faster than interval analysis, it recovers less structure.

Chang *et. el.* [22] 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 [47]. As noted by Schwartz *et el.* in [76], Hex-Rays seems to use an improved version of vanilla structural analyses.

Yakdan *et al.* [101] employed 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 [76]. It is built on top of the CMU Binary Analysis Platform (BAP) [16]. BAP lifts sequential x86 assembly instructions into an intermediate language called BIL. It also uses TIE [58] 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.

Our focus in this thesis is on decompiling native code. The research community has also explored approaches to decompile other types of code such Java bytecode [60, 61, 63]. Decompiling managed code involves fundamentally different challenges. For example, bytecode is usually more abstract than native code and contains more information such as data types. However, recovering structured control flow is complicated by issues related to Java exceptions and synchronized blocks [61].

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

[55] to disassemble x86 obfuscated code. Jakstab [53] 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) [107] 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 [83] is a binary analysis platform. The CMU Binary Analysis Platform (BAP) [16] 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 [82], REWARDS [59], TIE [58], and [48]. Other work [45, 44, 34, 52] 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 [58].

3.9 Summary

In this chapter 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 work suggests that structured and more compact code is better for human understanding. In the next sections, we present several techniques designed to improve the readability of the decompiled code and evaluate our approach with a user study.

4 Usability Optimizations

Authors' Contributions

The work presented in this chapter is based on our paper published at the 37th IEEE Symposium on Security and Privacy (S&P 2016) [100]. The chapter text is taken and adapted from this paper. The authors' contributions that are relevant to the contents of this chapter are as follows:

- **Khaled Yakdan** designed and implemented the usability optimizations on top of the DREAM decompiler.
- **Elmar Padilla** provided valuable feedback on the usability optimizations.
- **Matthew Smith** provided valuable feedback on the usability optimizations.

The abstraction recovery techniques discussed in previous chapters recover high-level abstractions as produced the compiler. Developers strive to write readable code so that it can be easily maintained. When compiled, compiler optimizations change the code structure into a semantically-equivalent form that satisfy the specific optimization goal. Compiler optimizations are usually tailored towards producing more efficient code or compacter executables. This can negatively impacts the code readability. For example, the compiler might change a loop structure into a more efficient but less readable form. As a result, the readability of the loop recovered by the decompiler is limited. This is a very important reason why state-of-the-

art decompilers still produce very complex and unreadable code. This often forces security experts and malware analysts to go back to analysing the assembler code.

In this chapter, we present several semantics-preserving code transformations to make the decompiled code more readable, thus helping security experts analyzing binary code. The main motivation driving the research presented in this chapter is assisting malware analysts understand and combat malware, which is one of the most challenging yet important cases of binary code analysis. A key idea behind our optimizations is that the high-level abstractions recovered by previous decompilation steps (see Figure 2.1) can be leveraged to devise powerful code simplifications. We have implemented our optimizations as extensions to Dream and call the new version Dream^{++}.

4.1 Introduction

The analysis of malware is a fundamental problem in computer security. It provides the necessary detailed understanding of the functionality and capabilities of malware, and thus forms the basis for devising effective countermeasures and mitigation strategies. Created by professional and highly skilled adversaries, modern malware is increasingly sophisticated and complex. Advanced malware families such as Stuxnet [43], Uroburos [46], and Regin [86] are examples of the level of sophistication and complexity of current malware. These malware samples shows the extraordinary lengths malware authors go to to conceal their activities and make the already tedious and time-consuming task of manual reverse engineering of malware even more challenging and difficult.

Due to the inability of a program to identify non-trivial properties of another program, the generic problem of automatic malware analysis is undecidable [72]. As a result of this limitation, security research has focused on automatically analyzing specific types of functionality, such as the identification of cryptographic functions [20], automatic protocol reverse engineering [19, 95], and the detection of DGA-based malware [5]. As another result of this limitation, security analysts often have to resort to manual reverse engineering for detailed and thorough analysis of malware, a difficult and time-consuming process. As a remedy, security researchers have started to explore approaches that assist analysts during analysis instead of replacing them. The proposed methods accelerate the analysis process by correctly identifying

functions in binaries [78, 7], reliably extracting binary code [107, 66, 55, 53, 12], deobfuscating obfuscated executable code [29, 99], and recovering high-level abstractions from binary code through decompilation [76, 103].

In this chapter, we argue that a human-centric approach can significantly improve the effectiveness of decompilers. To this end, we present several semantics-preserving code transformations to simplify the decompiled code. Improved readability makes the decompiled code easier to understand and thus can accelerate manual reverse engineering of malware. The key insight of our approach is that the abstractions recovered during previous decompilation stages can be leveraged to devise powerful optimizations. The main intuition driving these optimizations is that the decompiled code is easier to understand if it can be formed in a way that is similar to what a human programmer would write. Based on this intuition, we devise optimizations to simplify expressions and control-flow structure, remove redundancy, and give meaningful names to variables based on how they are used in code. Also, we develop a generic query and transformation engine that allows analysts to easily write code queries and transformations. We have implemented our usability extensions on top of the state-of-the-art academic decompiler DREAM [103]. We call this extended version DREAM^{++}.

4.2 Problem Statement & Overview

The focal point of this chapter is on improving the readability of decompiler created code to accelerate the analysis of malware. Code readability is essential for humans to correctly understand the functionality of code [17]. We conducted several informal interviews with malware analysts to identify shortcomings of state-of-the-art decompilers that negatively impact readability. We also conducted cognitive walkthroughs stepping through the process of restructuring malware code produced by Hex-Rays and DREAM to see what the problems of these two decompilers are. A common reason of these issues is the fact that current decompilers recover the program structure as produced by the compiler. In presence of compiler optimizations, compilers change the program structure for more efficiency. These optimizations leverage low-level aspects of the underlying architecture to increase the efficiency of the code, resulting in bad readability when this code is decompiled. We group the discovered problems into three categories

1. *Complex expressions*. State-of-the-art decompilers often produces overly complex expressions. Such expressions are rarely found in source code written by humans and are thus hard to understand. This includes

   (a) *Complex logic expressions*. Logic expressions are used inside control constructs (e.g., `if-then` or `while` loops) to decide the next code to be executed. Complex logic expressions makes it difficult to understand the checks performed in the code and the decisions taken based on them.

   (b) *Number of variables*. Decompiled code often contains too many variables. This complicates analysis since one must keep track of a large number of variables. Although decompilers apply a dead code elimination step, they still miss opportunities to remove redundant variables. In many scenarios, several variables can be merged into a single variable while preserving the semantics of the code.

   (c) *Pointer expressions*. Array access operations are usually recovered as dereference expressions involving pointer arithmetic and cast operations. Moreover, accesses to arrays allocated on the stack are recovered as expressions using the address-of operator (e.g., `*(&v + i)`).

   We present our approach to tackle these problems in Section 4.3.

2. *Convoluted control flow*. The readability of a program depends largely upon the simplicity of its sequencing control [38]. Two issues often complicates the control flow structure recovered by decompilers

   (a) *Duplicate/inlined code*. Binary code often contains duplicate code blocks. This usually results from macro expansion and function inlining during compilation. As a result, analysts may end up analyzing the same code block several times.

   (b) *Complex loop structure*. Control-flow structuring algorithms used by decompilers recognize loops by analyzing the control flow graph. For this reason, they recover the structure produced by the compiler which is optimized for efficiency but not readability. Stopping at this stage prevents decompilers from recovering more readable forms of loops as those seen in the source code written by humans.

We address these problems in Section 4.5. At the core of our optimization is our code query and transformation framework which we describe in Section 4.4.

3. *Lack of high-level semantics* High-level semantics such as variable names are lost during compilation and cannot be recovered by decompilers. For this reason, decompilers usually assign default names to variables. Also, some constants that have a special meaning in a given context, e.g., used by an API function or as magic numbers for file types. In Section 4.6, we describe several techniques to give variables and constants meaningful names based on how they are used in the code.

As an example illustrating these problems, we consider the decompiled code of the domain generation algorithm (DGA) of the Simda malware family produced by three decompilers: Hex-Rays (Figure 4.1), DREAM (Figure 4.2), and our improved DREAM^{++} (Figure 4.3). Here, we only show the main loop where the domains are computed[1]. As shown in the snippets, code produced by Hex-Rays and DREAM is rather complex and hard to understand. In the code produced by Hex-Rays, the loop variable `i` is never used inside the loop and the loop ends with a `break` statement. Moreover, the recovered checks for the parity of the loop counter involves complex low-level expressions (lines 26-30). Accessing the `char` arrays (`v37` and `v30`) uses pointer arithmetic, address-of operators, and dereference operators.

DREAM produced a slightly more readable code but still has a number of issues. Here, the recovered loop structure is not optimal and can be further simplified. Since the initial value of `v18` is zero, the condition of the `if` statement and the enclosed `do-while` loop are identical at the first iteration. This opens up the possibility to transform the whole construct into a more readable `while` loop.

Finally, the optimizations developed during the course of this chapter further reduce the complexity of the code. As can be seen from Figure 4.3, the code contains a simple `for` with a clear initialization step, condition, and increment step. With each loop iteration, a letter is selected from two `char` arrays (`v1` and `v2`) depending on the parity of the loop counter (`i % 2 == 0`) and the result is stored in the output array (`v3`).

Scope. DREAM^{++} is based on the DREAM decompiler which uses IDA Pro [51] to extract a disassembly and the control-flow graph from the binary. Arguably, the resulting disassembly

[1]The complete code can be found in Appendix A

```
void *__cdecl sub_10006390(){
  __int32 v13; // eax@14
  int v14; // esi@15
  unsigned int v15; // ecx@15
  int v16; // edx@16
  char *v17; // edi@18
  bool v18; // zf@18
  unsigned int v19; // edx@18
  char v20; // dl@21
  char v23; // [sp+0h] [bp-338h]@1
  int v30; // [sp+30Ch] [bp-2Ch]@1
  __int32 v36; // [sp+324h] [bp-14h]@14
  int v37; // [sp+328h] [bp-10h]@1
  int i; // [sp+330h] [bp-8h]@1
  // [...]
  v30 = *"qwrtpsdfghjklzxcvbnm";
  v37 = *"eyuioa";
  // [...]
  v14 = 0;
  v15 = 3;
  if ( v13 > 0 )
  {
    v16 = 1 - &v23;
    for ( i = 1 - &v23; ; v16 = i )
    {
      v17 = &v23 + v14;
      v19 = (&v23 + v14 + v16) & 0x80000001;
      v18 = v19 == 0;
      if ( (v19 & 0x80000000) != 0 )
        v18 = ((v19 - 1) | 0xFFFFFFFE) == -1;
      v20 = v18 ? *(&v37 + dwSeed / v15 % 6) : *(&v30 + dwSeed / v15 % 0x14);
      ++v14;
      v15 += 2;
      *v17 = v20;
      if ( v14 >= v36 )
        break;
    }
  }
  // [...]
}
```

Figure 4.1: Excerpt from the decompiled code generated by Hex-Rays of the domain generation algorithm of the Simda malware family. This example shows the main loop where the domain names are generated. At a high level, letters are picked at random from two arrays. Choosing the array from which to copy a letter is based on whether the loop counter is even or odd.

```
LPVOID sub_10006390(){
  int v1 = *"qwrtpsdfghjklzxcvbnm";
  int v2 = *"eyuioa";
  // [...]
  int v18 = 0;
  int v19 = 3;
  if(num > 0){
    do{
      char * v20 = v18 + (&v3);
      int v21 = v18 + 1;
      int v22 = v21;
      int v23 = v21 & 0x80000001L;
      bool v24 = !v23;
      if(v23 < 0)
        v24 = !(((v23 - 1) | 0xfffffffeL) + 1);
      char v25;
      if(!v24)
        v25 = *(((dwSeed / v19) % 20) + (&v1));
      else
        v25 = *(((dwSeed / v19) % 6) + (&v2));
      v18++;
      v19 += 2;
      *v20 = v25;
    }while(v18 < num);
  }
  // [...]
}
```

Figure 4.2: Decompiled code generated by DREAM for the same sample as Figure 4.1.

is not perfect and can contain errors if the binary is deliberately obfuscated. For the scope of this thesis, we assume that the assembly provided to the decompiler is correct. Should the binary code be obfuscated, tools such as [55, 107, 12] can be used to extract the binary code. Furthermore, recent approaches such as [29, 99] can be used to deobfuscate the binary code before providing it as input to the decompiler.

A high-level overview of our approach is as follows. First, the binary file is decompiled using DREAM. This stage decompiles each function and generates the corresponding control flow graph (CFG) and the abstract syntax tree (AST). Each node in the AST represents a statement or an expression in DREAM's intermediate representation (IR). Our work starts here. We develop three categories of semantics-preserving code transformations to simplify the code and

```
1  LPVOID sub_10006390(){
2    char * v1 = "qwrtpsdfghjklzxcvbnm";
3    char * v2 = "eyuioa";
4    // [...]
5    int v13 = 3;
6    for(int i = 0; i < num; i++){
7      char v14 = i % 2 == 0 ? v1[(dwSeed / v13) % 20] : v2[(dwSeed / v13) % 6];
8      v13 += 2;
9      v3[i] = v14;
10   }
11   // [...]
12 }
```

Figure 4.3: Decompiled code generated by DREAM++ for the same sample in Figure 4.1.

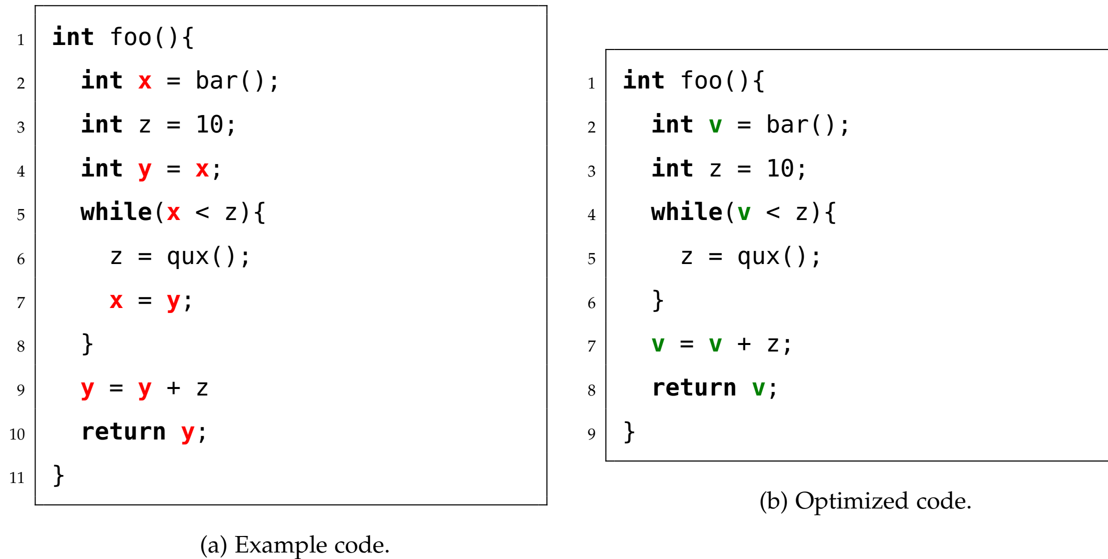
increase readability. These categories are expression simplification, control-flow simplification and semantics-aware naming. In the following sections, we discuss our optimizations in detail.

4.3 Expression Simplification

In this section, we present our optimizations to simplify expressions and remove redundancy from decompiled code.

4.3.1 Congruence Analysis

Congruence analysis is our approach to remove redundant variables from the decompiled code. The key idea is to identify variables that represent the same value and can be replaced by a single representative variable while preserving semantics. We denote such variables as *congruent variables*. DREAM already performs several optimizations to remove redundancy such as expression propagation and dead code elimination. However, there exist scenarios where traditional dead code elimination algorithms cannot remove redundant code. A prominent example is when compilers emit instructions to temporarily save some values that are later restored for further use. Depending on the control structure, this may result in circular dependency between the corresponding variables in the decompiler IR, preventing dead code elimination from removing them. As an example illustrating these scenarios, we consider the example shown in Figure 4.4a. In this example, lines 4 and 7 copy a value between variables x



(a) Example code.

(b) Optimized code.

Figure 4.4: Congruence Analysis

and `y`. Also, replacing `x` and `y` by a single representative, e.g., variable `v`, does not change the semantics of the program. Moreover, this replacement results in two trivial copy statements of the form `v = v` (lines 4 and 7) that can be safely removed, resulting in the more compact and readable code shown in Figure 4.4b.

This simple example gives insight into the different properties of code that play a role in the characterization of variable congruence. In summary, the following aspects need to be covered.

1) *Same Types*: Congruent variables have the same data types. This requirement is necessary to avoid the changing semantics because of implicit type conversions. For example, the transformation would not be semantics-preserving if y was of type `short`.

2) *Non-Interfering Definitions*: Replacing congruent variables by a single representative does not change the definitions that reach program points where these variables are used. Note that this does not require that the live ranges of congruent variables do not interfere. For example, the definition of `x` at line 7 is located in the live range of `y`, i.e., between a definition of `y` (line 4) and a corresponding use of `y` (line 9). However, the definition is a simple copy statement `x = y` and therefore using any of `x` or `y` at line 9 preserves semantics.

3) *Congruence-Revealing Statements*: The previous checks are enough to guarantee the semantics-preserving property of unifying variables. However, applying this to all variables without limitation may negatively impact readability. Two non-interfering variables of the same type may have different semantics (e.g., one integer variable used as a loop counter and a second integer used as the size of a buffer). Not merging such variables enables us to give each of them a representative name based on how the variable is used in code. Based on that we limit congruence analysis to variables for which the code contains indications that they are used for the same purpose. That is, we only check variables involved in copy statements of the form `x = y`, which we denote as *congruence-revealing* statements.

At the core of these checks is information about liveness of variables. To this end, we perform a fixed-point intraprocedural *live variable* analysis, a standard problem from compiler design [62, p. 443]. At a high level, live variable analysis determines which variables are *live* at each point in the program. A variable v is live at a particular point in the program $p \in P$ if p is located on an execution path from a definition of v and a use of v that does not contain a redefinition of v. This set of program points constitute the *live range* of the variable.

$$\textsc{LiveRange}(v) = \{p \in P : v \text{ live at } p\}$$

Algorithm 3 implements this idea by first calculating the set of candidate variable pairs, i.e., variables of the same types that are involved in congruent-revealing statements, and then checking these pairs for congruence. For each candidate pair (x, y), the algorithm checks if they do not have interfering definitions. In particular, the procedure InterferenceFree checks if each definition of variable y is either not located in the live range of x, or it is a copy statement of the form $y = x$. The same check is also done at the definitions of x. When two congruent variables are identified, the procedure Unify 1) chooses one representative variable v; 2) replaces all occurrences of the concurrent variables by the representative; and 3) removes the trivial copy statements resulted from this unification (of the form `v = v`). Next, the set of variables V is updated. Finally, liveness information of the newly added variable is updated as follows:

$$\textsc{LiveRange}(v) = \textsc{LiveRange}(x) \cup \textsc{LiveRange}(y)$$

Algorithm 3 Congruence analysis

```
1: procedure MergeCongruentVars(V)
2:     for (a, b) ∈ Candidatates(V) do
3:         if Congruent(a, b) then
4:             v ← Unify(a, b)
5:             V ← V \ {a, b}
6:             V ← V ∪ {v}
7:             UpdateLiveness(v)
8: procedure Congruent(a, b)
9:     return InterfenceFree(a, b) ∧ InterfenceFree(b, a)
10: procedure InterfenceFree(x, y)
11:     for all d ∈ Def(y) do
12:         if d ∈ LiveRange(x) ∧ d ≢ y = x then
13:             return false
14:     return true
```

Not that we do not require that congruent variables must have the same values at all program points. They may have different values at points where their live ranges do not interfere. For example, although different values of variables `x` and `y` reach the return statement in the code shown in Figure 4.4a, `x` is not live at lines 9 and 10. This enables us to use the same variable for both `x` and `y`.

It is worth mentioning that our approach is similar to the local variables packing step [89] that is used in the Soot framework [57] to merge local variables in order to produce compact code with the least possible number of locals.

4.3.2 Condition Simplification

The goal of this step is to find the simplest high-level form of logic expressions in the decompiled code. These expressions are very important for understanding the control flow of a program since they are used in control statements, such as `if-then-else` statements or `while` loops, to decide what code to execute next. Simplifying logic expressions is helpful in two aspects: first, it helps to recover the semantically equivalent high-level conditions to the low-level checks emitted by the compiler. Second, it helps to clear any misunderstanding caused by errors in the original code.

Low-level checks. During compilation a compiler uses a transformation called *tiling* to reduce the high-level program statements into assembly statements. As a result, each high-level

statement can be transformed into a sequence of semantically equivalent assembly instructions. During this process, high-level predicates are transformed to semantically equivalent low-level checks that can be executed efficiently. As an example, we consider the code shown in Figure 4.1. The right-hand side of the assignment at line 30 is a complex expression that checks whether the variable `v19` is an even or odd number. This does not look like a common operation used in source code, but it is equivalent to the high-level operation of computing `v19 % 2 == 0`.

Errors in the code. Malware code may contain logic errors that can create confusion for analysts. Malware analysts assume that the code they analyze performs some meaningful task they need to find out. They also know that malware often uses several tricks to hide its true functionality. With this mindset, when analysts observe a seemingly useless code, they need to double-check in order to exclude the possibility of a trick aimed at making the code looks useless. As a result, some time is wasted. The simple example from the Stuxnet malware family shown in Figure 4.5a illustrates this case. This code checks the version of the Windows operating system, a common procedure in *environment-targeted malware* [98]. However, the OR expression (marked in red) is always satisfied; any integer is either bigger than 5 or smaller than 6. Most probably, the malware authors intended to use an AND expression instead but did not for some reason. Simplifying this expression results in the code shown in Figure 4.5b.

To provide a generic simplification approach, we base our techniques on the `Z3` theorem prover [32]. Our approach proceeds as follows. First, we transform logic expressions in the Dream IR into semantically equivalent symbolic expressions for the `Z3` theorem prover. To achieve a faithful representation, we model variables as fixed-size bit-vectors depending on their types. The theory of bit-vectors allows modeling the precise semantics of unsigned and of signed two-complements arithmetic. During this transformation, we keep a mapping between each symbolic variable and the corresponding variable it represents in the original logic expression. Second, we use the theorem prover to simplify and normalize the symbolic expressions. Finally, we use the mapping to construct the simplified version of the logic expression in Dream IR.

```
1 //[...]
2 result = GetVersionExW(&VersionInformation);
3 if(result && VersionInformation.dwPlatformId == 2
4    && ( VersionInformation.dwMajorVersion >= 5
5       || VersionInformation.dwMajorVersion <= 6))
6 //[...]
```

(a) Hex-Rays

```
1 //[...]
2 BOOL result = GetVersionExW(&VersionInformation);
3 if(result != 0 && VersionInformation.dwPlatformId == 2)
4 //[...]
```

(b) DREAM^{++}

Figure 4.5: Excerpt from the decompiled code from a Stuxnet sample. The code checks the version of the Windows operating system.

4.3.3 Pointer Transformation

Accessing values through pointer dereferencing using pointer arithmetic can be confusing. Also, accessing buffers allocated on the stack may result in convoluted decompiled code that is difficult to understand.

Pointer-to-array transformation. Here, we use the observation that in C a pointer can be indexed like an array name. This representation clearly separates the pointer variable from the expression used to compute the offset from the start address. To guarantee the semantics-preserving property of this transformation, we search for variables of pointer types that are accessed consistently in the code. That is, all data that is read or written using the pointer variable have the same type τ. In this case, dereferencing these variables can be represented as array with elements of type τ. Here the resulting offset expression must be adjusted according to the size of type τ. For example, if a pointer `p` is consistently used to access 4-byte integers, then expressions such as `*(p + 4 * i)` can be transformed into the more readable `p[i]` form.

Reference-to-pointer transformation. In this step, we transform variables that are only used in combination with *address-of operator* (&) into pointer variables. One of the first steps in DREAM is

variable recovery that recovers individual variables from the binary code. For example, functions usually allocate a space on the stack to store local variables. Expressions accessing values in this stack frame are then recovered as local variables. For efficiency, buffers are often allocated on the stack when the maximum size is known at compile time. In this cases the variable recovery step represents the buffer as local variable `v` and expressions that access items inside this buffer are represented using the address-of operator as `&v`, resulting in a decompiled code that is hard to understand. For example, reading a character from a buffer allocated on the stack is represented as `*(&v37 + dwSeed / v15 % 6)` (line 31 in Figure 4.1). If a variable `v` is only accessed in the code as `&v`, we replace `v` by a pointer variable `v_ptr` that replaces address expressions `&v`. This may also create an opportunity to further simplify pointer dereferencing expressions in which the resulting pointer variable is involved in as array indexing. The previous example can be represented as `v37_ptr[dwSeed / v15 % 6]`.

4.4 Code Query and Transformation

At the core of our subsequent optimizations is our generic approach to search for code patterns and apply corresponding code transformations. The main idea behind our approach is to leverage the inference capabilities of logic programming to search for patterns in the decompiled output. To this end, we represent the decompiled code as logic facts that describe properties of the corresponding abstract syntax tree. This logic-based representation enables us to elegantly model search patterns as logic rules and efficiently perform complex queries over the code base. Usability is a key design goal, and therefore we enable users of our system to define search rules using normal C code and provide a rule compiler to compile them into the logic rules needed by our engine. We use the platform-independent, free SWI-Prolog implementation [92]. In the following, we describe our approach in detail.

4.4.1 Logic-Based Representation of Dream IR

This step takes as input the abstract syntax tree (AST) generated by Dream and outputs the corresponding logic facts, denoted as *code facts*. We represent each AST node as a code fact that describes its properties and nesting order in the AST. Table 4.1 shows the code facts for selected statements and expressions in Dream's intermediate representation (IR). The predicate

symbol (fact name) represents the AST node type. The first parameter is a unique identifier of the respective node. The second parameter is the unique identifier of the parent node (e.g., the containing `if` statement). Node ids and parent ids represent the hierarchical syntactic structure of decompiled code. Remaining parameters are specific to each fact and are described in detail in Table 4.1.

We generate the code facts by traversing the input AST and producing the corresponding code fact for each visited node. The code facts are stored in a fact base $\mathcal{F}$, which will be later queried when searching for code patterns. As a simple example illustrating the concept of code facts, we consider the code sample shown in Figure 4.6a. The corresponding code facts for the function body are shown in Figure 4.6c. The body is a sequence ($id = 3$) of two statements: an `if-then-else` statement ($id = 4$) and a `return` statement ($id = 14$). These two statements have the sequence node as their parent and their order in the sequence is represented by the order of the corresponding ids inside the sequence code fact.

4.4.2 Transformation Rules

The logic-based representation of code enables us to elegantly model search patterns 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. The premises describe the properties of the code pattern that we search for. In case of code queries, the conclusion is to simply indicate the existence of the searched pattern. For code transformation, the conclusion represents the transformed form of the identified code pattern.

We realize inference rules as Prolog rules, which enables us to ask Prolog queries about the program represented as code facts. Figure 4.7 shows two simple examples that illustrate the idea of modelling code search patterns as Prolog rules. The rule `if_condition` searches for condition expressions used in `if` statements. Rule parameters are Prolog variables that represent the pieces of information to be extracted from the matched pattern. The rule body represents the premises that must be fulfilled in order for the rule to return a match. At a high level, when a query is executed, Prolog tries to find a satisfying assignment to variables of the rule

	Code Fact	Description
Statements	`sequence`$(id, p_{id}, [\#s_1, \ldots, \#s_n])$	*sequence* of statements $s_1, \ldots, s_n$
	`loop`$(id, p_{id}, \tau, \#e_c, \#s_b)$	*loop* of type $\tau \in \{\tau_{\textbf{while}}, \tau_{\textbf{dowhile}}, \tau_{\textbf{endless}}\}$ and continuation condition e_c and body s_b
	`if`$(id, p_{id}, \#e_c, \#s_{\texttt{then}}, \#s_{\texttt{else}})$	*if* statement with condition e_c, the `then` part $s_{\texttt{then}}$, and the `else` part $s_{\texttt{else}}$
	`switch`$(id, p_{id}, \#e_v, [\#s^1_{\texttt{case}}, \ldots, \#s^n_{\texttt{case}}])$	*switch* statement with variable e_v and a set of cases $s^1_{\texttt{case}}, \ldots, s^n_{\texttt{case}}$
	`case`$(id, p_{id}, \#e_{\texttt{label}}, \#s)$	*case* statement with a label $e_{\texttt{label}}$ and a statement s
	`assignment`$(id, p_{id}, \#e_{\texttt{lhs}}, \#e_{\texttt{rhs}})$	*assignment* of the form $e_{\texttt{lhs}} = e_{\texttt{rhs}}$
	`return`$(id, p_{id}, \#e)$	*return* statement that returns expression e
	`break`(id, p_{id})	*break* statement
Expressions	`call`$(id, p_{id}, \#e_{\texttt{callee}}, [\#e^1_{\texttt{arg}}, \ldots, \#e^n_{\texttt{arg}}])$	*call* expression of the function $e_{\texttt{callee}}$ with arguments $e^1_{\texttt{arg}}, \ldots, e^n_{\texttt{arg}}$
	`operation`$(id, p_{id}, \texttt{op}, [\#e^1_{\texttt{e}}, \ldots, \#e^n_{\texttt{e}}])$	*operation* (e.g., addition or multiplication) with operand `op` involving expressions $e^1_{\texttt{e}}, \ldots, e^n_{\texttt{e}}$
	`ternaryOp`$(id, p_{id}, \#e_c, \#s_{\texttt{then}}, \#s_{\texttt{else}})$	*ternary operation* of the form $e_c \texttt{?} s_{\texttt{then}} \texttt{:} s_{\texttt{else}}$
	`numericConstant`(id, p_{id}, v)	*numeric constant* of value v
	`stringConstant`(id, p_{id}, v)	*string constant* of value v
	`memoryAccess`$(id, p_{id}, \#e_{\texttt{address}})$	*memory access* to address $e_{\texttt{address}}$
	`localVariable`$(id, p_{id}, \texttt{name}, \tau)$	*local variable* with name `name` and type τ
	`globalVariable`$(id, p_{id}, \texttt{name}, \tau)$	*global variable* with name `name` and type τ
	`identifier`$(id, p_{id}, \#e_{\texttt{var}})$	*identifier* represents the occurrence of a variable $e_{\texttt{var}}$ in an expression p_{id}

Table 4.1: Logic-based predicates for the Dream IR. Each predicate has an *id* to uniquely represent the corresponding statement or expression. The second argument of each code fact is the parent id *pid* that represent the id of containing AST node. For a statement or expression e, we denote by $\#e$ the id of e.

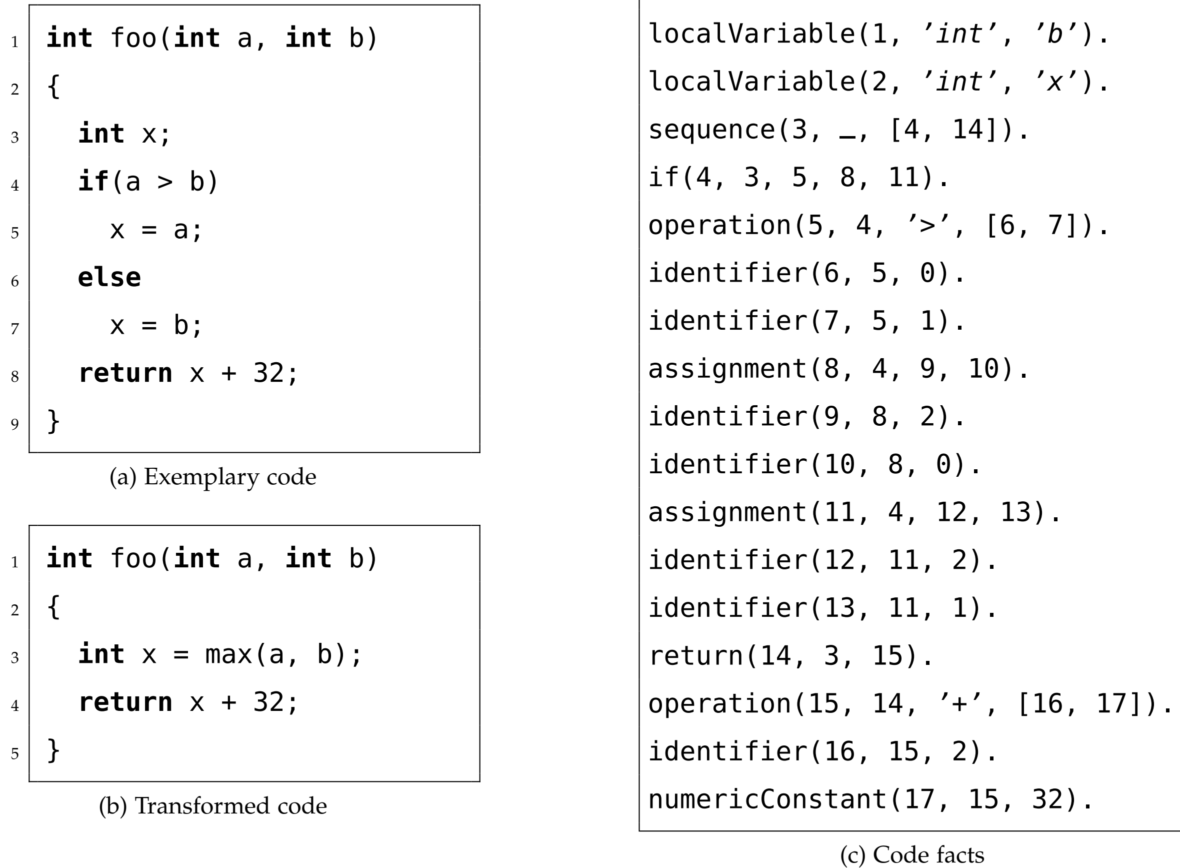


Figure 4.6: Code representations.

that makes it consistent with the facts. For example, the query `if_condition(Condition)` executed on the fact base in Figure 4.6c returns the match {`Condition=5`}, the id of the code fact corresponding to the condition of the `if` statement in Figure 4.6a. This unification is done by matching the rule only premise with the corresponding code fact of the `if` statement.

A very powerful aspect of logic rules is that the corresponding queries can be adapted for multiple purposes. For example, the second rule `assignment_to_local` searches for assignments to a local variable given its name. Using a concrete variable name, the query returns all assignments to the corresponding variable (e.g., `assignment_to_local(Assignment, 'x')`). On the other hand, using a Prolog variable for the name, the query returns all assignments to all variables (e.g., `assignment_to_local(Assignment, Name)`).

```
1  if_condition(Condition) :-
2    if(_, _, Condition, _, _).
3
4  assignment_to_local(Assignment, VarName) :-
5    assignment(Assignment, _, Lhs, _),
6    identifier(Lhs, Assignment, Variable),
7    localVariable(Variable, _, VarName).
```

Figure 4.7: Sample search patterns

```
Signature:
  max(result, v1, v2){
    if(v1 > v2)
      result = v1;
    else
      result = v2;
  }
Transformation:
  result = max(v1, v2);
```

Figure 4.8: Sample transformation rule

Transformation rules can be written in normal C code. Figure 4.8 shows a sample transformation rule that searches for `if` statements that compute the largest of two values and replace them by a call to the `max` library function. A transformation rule consists of two parts: *rule signature* and *code transformation*. The rule signature describes the code pattern to be searched for and is written as normal C function declaration: the list of parameters, denoted as *rule parameters*, represents the variables that need to be matched to the actual variables by Prolog inference engine so that the transformed code can be constructed. The function body represents the code pattern. The transformation part describes the transformed code that should replace the matched pattern. Also here, the transformation is written as nomal C code in terms of the rules parameters.

We compile transformation rules into logic rules that can be used by Prolog's inference engine. To this end, we parse the rule body and then traverse the resulting AST. For each visited AST node, we generate the corresponding code fact. Here, we use Prolog variables for the generated fact identifiers. These variables will be then bound to the actual identifiers from the fact base when the inference engine finds a match. Finally, the compiled rule is stored in the rule base $\mathcal{R}$ and the corresponding query in the query base $\mathcal{Q}$.

4.4.3 Applying Transformation

We first initialize Prolog with the code base $\mathcal{F}$ and the rule base $\mathcal{R}$. We then iteratively apply the queries in the query set $\mathcal{Q}$. If a match is found, the inference engine *unifies* the rule arguments to the identifiers of the corresponding code facts. In this case, we construct the equivalent transformed code. To this end, we first parse the transformation string to construct the corresponding AST. During this process, we use the corresponding AST node for each signature argument to get the transformed code in terms of the original variables from the initial code base. For example, applying the sample rule in Figure 4.8 to the fact base shown in Figure 4.6c returns one match: $\{\texttt{result} = \texttt{x}, \texttt{v1} = \texttt{a}, \texttt{v2} = \texttt{b}\}$. This enables us to replace the complete `if` statement by the function call `x = max(a, b)` to get the code shown in Figure 4.6b. Finally, we update the fact base $\mathcal{F}$ so that it remains consistent with the AST.

The code query and transformation engine is the basis for our subsequent code optimizations that identify certain code patterns and corresponding transformations aimed to simplify code and improve readability.

4.5 Control-Flow Simplification

In this section we present our techniques to simplify the control flow of decompiled code.

4.5.1 Loop Transformation

Compiler optimizations often change the structure of loops in the source code. While this optimization is aimed to increase efficiency, the resulting loop structure becomes less readable, reducing the quality of decompiled code. A well-known loop optimization is *inversion*, which changes the standard `while` loop into a `do-while` loop wrapped in an `if` conditional, reducing the number of jumps by two for cases where the loop is executed. That is, loops of the form `while(e){...}` are transformed into `if(e){do{...}while(e);}`. Doing so duplicates the condition check (increasing the size of the code) but is more efficient because jumps usually cause a pipeline stall. Additionally, if the initial condition is known to be true at compile-time and is side-effect-free, the `if` guard can be skipped.

Here we make the observation that `while` loops are more readable than `do-while` loops since the continuation condition is clear from the start. Moreover, some `while` loops can be further simplified into `for` loops where the initialization statement, continuation condition, and the increment statements are clear from the start. Based on this observation, we analyze `do-while` loops and check if they can be transformed into `while` loops. Here, we distinguish between two cases:

Guarded `do-while` loops. loops of the form `if(c1){do{...}while(c2);}` are transformed into `while(c2){...}` if it can be proven that $c_1 == c_2$ at the start of the first iteration of the loop. Note that c_1 and c_2 does not have to identical logical expressions. As an example, we consider the code sample shown in Figure 4.9a. The conditions `*(_BYTE *)v7 != 0` and `*(_BYTE *)(v8 + v7)!= 0` are both yield the same Boolean value at the entry of loop. Note that the reaching definition of variable `v8` at this point is `v8 = 0`.

Unguarded `do-while` loops. For these loops we only check if the loop condition is true for the first iteration. In this case, the loop can be transformed into `while` loop.

To check the value of logic expressions at loop entry, we compute the set of definitions for loop variables that reach the loop entry. To this end, we perform a fixed-point intraprocedural *reaching definitions* analysis, a standard problem from compiler design [62, p. 218]. Often the reaching definitions for loop variables are assignments of constant values that represent the initial value of a loop counter. This makes it easy to substitute this initial value in the logic expressions and check for equivalence at loop entry.

4.5.2 Function Outlining

Function inlining is a well-known compiler optimization where all calls into certain functions are replaced with an in-place copy of the function code. This improves runtime performance since the overhead of calling and returning from a function is completely eliminated. In the context of code obfuscation, inlining is a powerful technique [28]. It makes reverse engineering harder in two ways: first, several duplicates of the same code are spread across the program. As a result, analysts end up analyzing several copies of the same code. Second, internal abstractions such as the calling relationships between functions in the program are eliminated.

```
1  int sub_408A70(int a1, int a2){
2    [...]
3    v8 = 0;
4    if ( *(_BYTE *)v7 )
5    {
6      do
7        ++v8;
8      while ( *(_BYTE *)(v8 + v7) );
9    }
10   v9 = 0;
11   if ( *(_BYTE *)a1 )
12   {
13     do
14       ++v9;
15     while ( *(_BYTE *)(v9 + a1) );
16   }
17   if ( v8 == v9 ){
18     [...]
19   }
20   [...]
21 }
```

(a) Hex-Rays

```
1  int sub_408A70(char * str2, void
       * a2){
2    [...]
3    len1 = strlen(str1);
4    if(len1 == strlen(str2)){
5      [...]
6    }
7    [...]
8  }
```

(b) DREAM^{++}

Figure 4.9: Excerpt from the code of the Cridex malware family showing the code inlining technique.

Reversing function inlining is valuable for the manual analysis of malware. As a simple example illustrating the benefits of function outlining, we consider the excerpt code from the Cridex malware family shown in Figure 4.9. Each of the two loops in Hex-Rays decompiled code shown in Figure 4.9a computes the length of a string by incrementing the counter by one for each character until the terminating null-character is found. DREAM^{++} identified these two blocks as an implementation of the `strlen` library function and replaced them with corresponding function calls as shown in Figure 4.9b. This simple example gives insights into the benefits of function outlining for code analysis.

1) *Compact code.* Replacing a code block by the equivalent function call eliminates duplicate code blocks and results in a more compact decompiled output. The whole code block is replaced by a function call whose name directly reveals the functionality of the code block. Moreover, temporary variables used inside the block are removed from code, reducing the number of variables that an analyst should keep track of.

2) *Meaningful variable names.* Outlined functions have known interfaces that include the names of their parameters. These names represent their semantics and reveals important information about the variable job. We leverage this information to give meaningful names the variables in the decompiled output.

3) *Improved Type Recovery.* Approaches to recover types from binary code such as [59, 58] rely on *type sinks* as a reliable starting points. Type sinks are points in the program where the type of a given variable is known. This includes calls to functions whose signatures are known. Outlining a function generates a new type sink that can be used to improve the performance of type inference algorithms.

4) *Recovering inter-dependencies.* Function outlining implicitly recovers calling relationships between the inlined function and the functions calling it. That is, it identifies points in the program that call the function. Calling relationships are very important for manual reverse engineering. After having analyzed a given function, malware analysts can draw conclusions about the calling functions.

We leverage our code query and transformation engine to easily include multiple transformation rules for several functions that copy, compare, compute the length, and initialize buffers. For example, we handle `strcpy`, `strlen`, `strcmp`, `memset`. For string functions, both 8-bit and 16-bit character versions are handled. We also include signatures for the version of string functions that take buffer length as argument.

Users of our system can easily add new transformation rules to handle new functions. When an analyst observes a repeating code pattern, she can simply write a transformation rule that replace the whole code block by a function call with a name that represents its functionality. All other copies of the same block will be outlined. Code blocks are not only duplicated as a result of function inlining. In C, *function-like macros* are pre-processor macros that accept arguments and are used like normal function calls. These macros are handled by the pre-processor, and are thus guaranteed to be inlined.

4.6 Semantics-Aware Naming

In this section we describe several readability improvements at the level of variables in the decompiled code.

4.6.1 Meaningful Names

Variable names play an important role when analyzing source code. These names reveal valuable information about the purpose of this variable and how it is used in the program. We give variables meaningful names based on the context in which they occur. Here we distinguish the following cases:

Standard library calls. With well-defined API, standard library calls are important source of variable names. For example, the Windows API `URLDownloadToFile`, which downloads data from the Internet and saves them to a file, takes five arguments. Among them one argument, named `szURL`, represents the URL to download. A second argument, named `szFileName`, represents the name or full path of the file to create for the download. By analyzing library function calls and returns, we rename variables used as parameters or return values, directly revealing their purpose to the analyst.

Context-based naming. The way a variable is used in code gives insights into its purpose. We analyze the context in which variables are used to provide meaningful names to them. More specifically, we distinguish the following cases:

1) *Loop counters*. We query the decompiled code for *counting loops*, i.e., loops that updates a variable inside their body and then test the same variable in their continuation condition. Counting variables in short `for` loops are renamed to `i`, `j`, or `k`. Counting variables for other loops are renamed to `counter`.

2) *Array indices*. We rename variables used as indexes for arrays to `index`

3) *Boolean variables*. Variables that contain the result of evaluating a logic expressions are renamed to `cond`. This encodes the fact that they represent testing a condition in the variable name.

When multiple variables are identified that can take the same name, we add subscripts to the default names to have unique names. For example if three loop counters are identified, they are renamed to `counter1`, `counter2`, and `counter3`.

4.6.2 Named Constants

Constants are important corner pieces in the process of reverse engineering. For example, some cryptographic algorithms uses magic numbers, and several file formats include magic numbers to identify the file type. Also standard library functions assign specific constant to special meanings. Usually these numbers have a textual representation in the source code. We use two sources to identify these special constants.

Library API constants. Many functions in the C standard library and Windows API define special named constants. These constants have a specific meaning and are thus given representative names. During compilation compilers replace this symbolic representation of the constant by the corresponding numeric value. For example, the function `CreateFile` uses the constant `GENERIC_READ` to request a read access to the opened file. This becomes `0x80000000` in the binary. To recover the symbolic, easily remembered names of these constants, we check for the occurrence of named constants for a wide range of library function. For example this transformation would transform the API function call `CreateFileA(f, 0x80000000, 1, ...)` into the more readable form `CreateFileA(f, GENERIC_READ, FILE_SHARE_READ, ...)`.

File magic numbers. For many file types, a file starts with a short sequence of bytes (mostly 2 to 4 bytes long) to uniquely identify its type. Detecting such constants in files is a simple and effective way of distinguishing between many file formats. For example, DOS MZ executable file format and its descendants (including NE and PE) begin with the two bytes `4D 5A` (characters `'MZ'`). Malware usually downloads files from its server at run time and may check which file type it received. We check if these constants are used in the conditions of flow control statements.

4.7 Related Work

A wealth of research has been conducted on decompilation and the development of principled methods for recovering high-level abstractions from binary code. At a high level, there are four lines of research relevant to the work presented in this chapter. First, approaches to extract binary code from executables. Second, research on recovering abstractions required for source code reconstruction. Third, work on end-to-end decompilers. Finally, techniques to query code bases and apply transformations.

Binary code extraction. A fundamental step for decompilation is the correct extraction of binary code. Kruegel et al. [55] presented a method to disassemble x86 obfuscated code. Kinder et al. [53] proposed a method that interleaves multiple disassembly rounds with data-flow analysis to achieve accurate and complete CFG extraction. The binary analysis platform BitBlaze [83] and its successor BAP [16] use value set analysis (VSA) [6] to resolve indirect jumps. Run-time packers are often used by malware-writers to obfuscate their code and hinder static analysis [88]. To handle these cases, the research community proposed approaches that rely on dynamic analysis to cope with heavily obfuscated. This include approaches to extract a complete CFG [66], extract binary code from obfuscated binaries [107], deobfuscate obfuscated executable code [29, 99]. A closely related topic is the identification of functions in binary code. Recently, security research started to explore approaches based on machine learning to solve this problem. ByteWeight [7] learn signatures for function starts using a weighted prefix tree, and recognize function starts by matching binary fragments with the signatures. Shin et al. [78] use neural networks.

Abstractions recovery from binary code. Source code reconstruction requires the recovery of two types of abstractions: data type abstractions and control flow abstractions. Previous work addressed principled methods to recover these abstractions from binary code. Recent work proposed static and dynamic approaches to recover both scalar types (e.g., integers or shorts) and aggregate types (e.g., arrays and structs). Prominent examples include REWARDS [59], Howard [82], TIE [58], and MemPick [48]. Other work [45, 44, 34, 52] focused on C++ specific issues, such as recovering C++ objects, reconstructing class hierarchy, and resolving indirect calls resulting from virtual inheritance.

Early work on control structure recovery relied on interval analysis [2, 27], which deconstructs the CFG into nested regions called intervals. Sharir [77] subsequently refined interval analysis into structural analysis. Structural analysis recovers the high-level control structure by matching regions in the CFG against a predefined set of patterns or region schemas. Engel et al. [40] 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.

Significant advances has been made recently in the field of control flow structure recovery. Schwartz et el. [76] proposed two enhancements to vanilla structural analysis: 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. Yakdan et. al. [103] proposed *pattern-independent* control flow structuring, an approach that relies on the semantics of high-level control constructs rather than the shape of the corresponding flow graphs. Their method is a departure from the traditional pattern-matching approach of structural analysis and is able to produce a `goto`-free output.

Code query and transformation. Several code query technologies based on first-order predicate logic have been proposed. They are mainly used in software engineering for detecting design patterns or patterns of problematic design. These techniques support specific source languages and they either introduce new languages for modeling code queries such as CrocoPat [10, 11] and Soul [97], or users of these tools have to write logic rules directly such as JTransformer [54]. Our code query and transformation engine is based on the Dream IR and enables malware analysts to directly write transformation rules as normal C code.

Decompilers. Cifuentes laid the foundations for modern decompilers. In her PhD thesis [24], she presented several techniques for decompiling binary code that spans a wide range of techniques from data-flow analysis and control-flow analysis. These techniques were implemented in dcc, a decompiler for Intel 80286/DOS to C. Cifuentes et al. also developed *asm2c*, a SPARC assembly to C decompiler, and used it to decompile the integer SEPC95 programs [26].

Van Emmerik proposed to use the Static Single Assignment (SSA) form for decompilation in his PhD thesis [39]. His work shows that SSA enables efficient implementation of many decompiler components such as expression propagation, dead code elimination, and type analysis. His techniques were implemented in the open-source Boomerang decompiler. Although faster than interval analysis, it recovers less structure. Another open-source decompiler [21] is based on the work of van Emmerik.

Chang et el. [22] created a modular framework for building pipelines of cooperating decompilers. Decompilation is performed by a series of decompilers connected by intermediate languages. Their work demonstrates the possibility of using source-level tools on the decompiled source to find bugs that were known to exist in the original C code.

Hex-Rays is the *de facto* industry standard decompiler [47]. Hex-Rays is developed by Ilfak Guilfanov and built as plugin for the Interactive Disassembler Pro (IDA). Since it is closed source, little is known about the exact approach used. It uses an enhanced version of vanilla structural analysis and has an engine to recognize several inlined functions. There are also other decompilers available online such as DISC [56] and REC [71]. However, our experience suggests that all previously mentioned decompilers are not as advanced as Hex-Rays.

Phoenix is an advanced academic decompiler created by Schwartz et al. [76]. It is built on top of the Binary Analysis Platform (BAP) [16], which lifts sequential x86 assembly instructions into the BIL intermediate language. It also uses TIE [58] to recover types from binary code. Phoenix uses an enhanced structural analysis algorithm that can correctly recover more structure than vanilla structural analysis. Schwartz et al. were the first to measure correctness of decompiler as a whole. Their methods rely on checking if the decompiled code can pass the automatic checks written for source code.

All presented works presented above share two common characteristics. First, they do not leverage the recovered abstraction to simplify the decompiled code, and thus they miss opportunities to improve readability. At best, minimal readability enhancements are implemented.

4.8 Summary

In this chapter, we created a host of novel readability-focused code transformations to improve the quality of decompiled code. Our transformations simplify both program expressions and

control flow. They also assign meaningful names to variables and constants based on the context in which they are used in the program.

In the next chapter, we describe our user study to evaluate the quality of decompilers for malware analysis and presents the results.

5 Malware Analysis User Study

Authors' Contributions

The work presented in this chapter is based on our paper published at the 37th IEEE Symposium on Security and Privacy (S&P 2016) [100]. The chapter text is taken and adapted from this paper. The authors' contributions that are relevant to the contents of this chapter are as follows:

- **Khaled Yakdan** implemented the online study platform and conducted the pre-study.
- **Sergej Dechand** designed the user study and implemented the online study platform. Sergej also performed the statistical evaluation of the study results.
- **Elmar Padilla** was very helpful in discussing the design and evaluation of the user study.
- **Matthew Smith** provided valuable feedback on the design of the user study and the evaluation of the study results.

Manual reverse engineering of binary code has been a main driving force behind decompiler research. However, previous work has never featured user studies to evaluate whether and to what extent the proposed approaches can actually help human analysts. Cifuentes et al.'s pioneering work [24] and numerous subsequent works [22, 26, 44, 39, 76, 103] all evaluated the decompiler quality based on some machine-measurable readability metric such as

the number of `goto` statements in the decompiled code or how much smaller the decompiled code was in comparison to the input assembly. These metrics are based on the assumption that compacter code is easier to analyze or code with less `goto` is easier to understand. Without well-designed user studies, these assumptions remain unsupported. Moreover, a significant amount of previous work featured a manual qualitative evaluation on a few, small, sometimes self-written, examples [24, 45, 44, 39]. To show the effectiveness of decompilers in practice, they should be evaluated using real-world code examples.

In this chapter we present the first user study on malware analysis. We have chosen to use malware in our study because malware represents one of the most challenging cases for binary analysis tools. We conducted a study with 21 students who had completed a course on malware analysis and binary decompilation and 9 professional malware analysts. The study included 6 reverse engineering tasks of real malware samples that we obtained from independent malware experts. The results of our study show that our improved decompiler DREAM^{++} produced significantly more understandable code that outperforms both the leading industry and academic decompilers: Hex-Rays [47] and DREAM. Using DREAM^{++} participants solved 3× more tasks than when using Hex-Rays and 2× more tasks than when using DREAM. Both experts and students rated DREAM^{++} significantly higher than the competition.

5.1 User Study Design

The goal of our study is to test the readability of the code produced by our improved decompiler DREAM^{++} compared to its previous version DREAM and the industry standard Hex-Rays[1]. We planned a user study in which participants have to solve a number of reverse engineering tasks with different decompilers. The participants task is to analyse the code snippets and answer a number of questions about the functionality of the code. Our web-based study platform showed the code in split screen together with the questions. Participants could edit the code to help with the analysis.

To measure user perception, after each task we asked the participants for feedback and a couple of questions regarding readability. Each participant got a number of code snippets

[1] We opted to not compare DREAM^{++} to any other decompilers because of the upper bound of the number of participants (see §5.1.3).

produced by different decompilers without being told which decompiler was being used, so we would get an unbiased evaluation of the code. Only at the end of the study we showed the users the code produced by all decompilers side by side and asked them to give an overall rating for the decompilers.

5.1.1 Task Selection

To minimize the risk of bias, i.e., subconsciously selecting tasks which would favor our decompiler, we approached three independent professional malware analysts for the process of task selection. The analysts were known to us, but were not involved in the study or the work on the decompiler. We told them that we wanted to conduct a study on malware analysis and requested that they supply us with malware code snippets they themselves had to analyse in the course of their work. We requested that the snippets fulfil some sort of function that should be understandable without needing the rest of the malware code. In total we got 8 snippets of code. Two of the snippets contained an XOR based encryption/decryption algorithm, so we removed one of those and were left with 7 malware snippets. We ran a pre-study which will be described in more detail below to test the tasks. For this an even number of tasks was preferable so we added one additional code snippet. Based on the results of the pre-study we removed this and one other snippet so we are left with 6 snippets. We grouped these snippets into two groups: *Medium* (three tasks), and *Hard* (three tasks). In the following we describe the tasks in detail.

1. *Encryption* Encoding functions are used widely in malware as well as benign applications. Malware can encrypt exchanged messages with C&C servers and encode internal strings to avoid static analysis. This task is a function from the Stuxnet malware that decrypts the `.stub` section that contains Stuxnet's main DLL.

2. *Custom Encoding* This task is an XOR encryption/decryption function from the Stuxnet malware family. This function performs a word-wise XOR with `0xAE12` and is used by many Stuxnet components to disguise some strings.

3. *Resolving API Dynamically* In order to avoid static analysis, malware usually avoids listing the API functions it needs in the import table. Instead, it can resolve them dynamically

at runtime. The task is a function from the Cridex malware that takes as input the name of an API function and returns the corresponding starting address.

4. *String Parsing* Malware often receives commands and configuration files from the server. Thus, it needs to parse these commands to extract parameters and other information from C&C messages. This task is the injects parsing function from the URLZone malware. The function examines a string for the occurrence of the first sequence `%[A-Z0-9]%` and returns a pointer to start of such a string and its length.

5. *Download and Execute* A very common function is to download an executable from a C&C server and later executing it. This can for example be the case for [18]. The task involves analyzing the update mechanism of the Andromeda malware. The snippet downloads a file from a remote server and checks if it is a valid PE executable or a Zip archive containing an executable. In this case the file is saved on disk and executed.

6. *Domain Generation Algorithm* Malware is often equipped with domain generation algorithms (DGA) to dynamically generate domain names used for C&C (e.g., depending on seed values such as the current date/time and Twitter trends) [5]. It is a powerful technique to make botnets more resilient to attacks. The task contains the DGA of the Simda malware.

The full decompiled malware source code used in our study is presented in Appendix A. Here, we present the decompiled code from the three decompilers as well as the set of tasks.

5.1.2 Pre-Study

Before conducting our main user study we conducted a small scale pre-study for following purpose: it is best practice to test user studies in a pre-study to unearth problems with the task design and the study platform. We also wanted to check whether our cognitive walk-through and informal interviews had missed any important issues that should be dealt with before conducting the full study. To plan the study and the appropriate compensation, we needed estimates on how long each task would take.

Since malware analysis is a highly complex task requiring specialized skills, we recruited students who had successfully completed our malware boot camp. The malware boot camp is

a lab course held each semester at our university, in which students are introduced to the field of malware and binary code analysis. For the pre-study we recruited two of these students.

We conducted the study in our usability lab and used a think-aloud protocol, asking participants to vocalize their thought processes and feelings as they performed their tasks. We chose this protocol to obtain insights into the users' thought processes, and the barriers and problems they faced. In the pre-study we only tested DREAM^{++} and Hex-Rays. The first participant got four tasks decompiled with DREAM^{++} and four decompiled with Hex-Rays and had to answer questions about the functionality of the code. The second participant got the inverse selection. Assignment was randomized. After each task, the participants were asked to provide feedback about the quality of the code they analyzed. Then, they were shown the output generated by the other decompiler and asked which code they find more readable and how long they think they would take to analyze that output.

Pre-Study Results

There were only minor bugs with the study platform and the participants understood the task descriptions without problems. Based on their think aloud feedback, we did not find any open problems that had not been discovered during the cognitive walkthrough. We also ranked our tasks based on reported difficulty.

In the main study, we wanted to additionally test DREAM implying a smaller number of task samples since we never show code from different decompilers for the same task. Based on the pre-study we estimated that the main study should be completed in 3 hours.

Table 5.1 shows an overview of the results and the comments made by the participants. Task 8 is the task we added to balance the study.

PARTICIPANT	DECOMPILER	PERFORMANCE	DURATION	PARTICIPANT FEEDBACK
Task 1: Encryption				
P1	DREAM^{++}	●	12 m	1.1 DREAM^{++}'s output is similar to what I would write 1.2 I would need 2x more time for Hex-Rays.
P2	Hex-Rays	●	16 m	1.3 DREAM^{++}'s is shorter and easier to understand.
Task 2: Custom Encoding				

P2	DREAM^{++}	●	9 m	2.1 It was easy to follow and understand the code. 2.2 Hex-Rays code is complex and I would take longer to understand.
P1	Hex-Rays	●	42 m	2.3 I was confused about the loop condition. 2.4 The code is difficult to understand. 2.5 default variable names makes it more difficult 2.6 DREAM^{++}: shorter, less variables, loop is easier to understand 2.7 I would give DREAM^{++} 8/10 and Hex-Rays 4/10
Task 3: Resolving API Dynamically				
P2	DREAM^{++}	●	16 m	3.1 DREAM^{++}'s output could be further simplified. 3.2 For the Hex-Rays output I would need at least 45 minutes. 3.3 I find the meaningful names assigned by DREAM^{++} helpful.
P1	Hex-Rays	●	23 min	3.4 Hex-Rays has several redundant variables. DREAM^{++} output has less variable. 3.5 I find the code in the last loop a spaghetti code.
Task 4: String Parsing				
P1	DREAM^{++}	●	22 m	4.1 I find the code easy to understand because it looks like the code I would write when programming. 4.2 DREAM^{++}'s output is much better. 4.3 It helped me that the DREAM^{++} has less variables.
P2	Hex-Rays	○	43 m	4.4 DREAM^{++}'s code has less variables and thus easier to understand. 4.5 The control flow is easier to understand since no `goto` statements. 4.6 It is much easier to follow the control flow in DREAM^{++} output.
Task 5: Download and Execute				
P2	DREAM^{++}	●	16 m	5.1 Named constants help. 5.2 No `goto` spaghetti code.

P1	Hex-Rays	●	36 m	5.3 `goto` statements are confusing: jumping out of the loop and then back in it. 5.4 DREAM^{++}'s output is easier to understand. One can simply read the code sequentially without worrying about these jumps. 5.5 I cannot say how much this will influence the time I need to solve the task when analyzing DREAM^{++}'s output.
Task 6: Domain Generation Algorithm				
P1	DREAM^{++}	●	33 m	6.1 The control flow inside the function is easy to understand 6.2 For the Hex-Rays code, I would need at least 60 minutes (probably 90 minutes). Maybe I would give up after that.
P2	Hex-Rays	◐	36 m	6.3 Code looks very weird. 6.4 I gave up because I do not think I could understand the code in the loop.
Task 7: Checking OS Version				
P1	Hex-Rays	●	3 m	No Comments
P2	DREAM^{++}	●	7 m	No Comments
Task 8: Persistence				
P1	DREAM^{++}	●	2 m	No Comments
P2	Hex-Rays	●	2 m	No Comments

Table 5.1: Test User Study. The third column denotes the result of performing the task: ● task is completely solved, ◐ = task is partially solved, and ○ = task is not solved. Tasks are ordered according to difficulty level as shown by the pre-study.

5.1.3 Methodology

We conducted a within-subjects design experiment where participants had to analyze the code snippets decompiled by the three different decompilers (DREAM^{++}, DREAM and Hex-Rays). To begin with, we provided a detailed explanation of the concept and the procedure to participants. Participants were allowed to use the Internet during the study since this mirrors how analysts actually work. The goal was to remove all aspects not related to the quality of decompiled code. The tutorial was followed by a training phase to ensure that the participants were familiar enough with the system to avoid system related mistakes. We provided a sample code snippet and instructed the participants to rename variables in it and look for information about a library function online.

The main user study is almost unchanged from the pre-study. The methodology differs in the following points. The decompiler names were blinded so as not to bias the participants. And the study was not conducted in the lab but via our online study platform. This decision was made for several reasons: Firstly, not all of the students who have completed the malware bootcamp were still living locally and we wanted to maximise our recruiting pool, since participants at this level are very scarce. We also wanted to conduct the study with professional malware analysts and it is unrealistic to expect them to come to the lab. We decided to conduct the entire study online, to keep the results comparable.

Variables and Conditions

In our experiment, we have two independent variables:

1. *Decompiler* Decompiler used to solve a given task and has three conditions: DREAM^{++}, DREAM, and *Hex-Rays*. Hex-Rays is the leading industry decompiler that is widely used by malware analysts. Therefore, we compare DREAM^{++} to Hex-Rays to examine whether our approach improves the current state of malware analysis. We tested the latest Hex-Rays version, which is 2.2.0.150413 as of this writing. Also, we compare DREAM^{++} to the original DREAM decompiler to evaluate the usefulness of the extensions presented in Chapter 4.

2. *Difficulty* A within-subjects factor that represents the difficulty of the task. Based on the results from our pre-study (§5.1.2), we grouped the tasks according to their difficulty into two groups (*medium* and *hard*), each containing three tasks.

Condition Assignment

We chose a within subjects design since personal skill is a strong factor in performance. To avoid learning and fatigue effects in our within-study design, the order in which participants used decompilers within each difficulty level, but also the difficulty levels were permutated using the counterbalanced measures design. Figure 5.1 shows the details of our counterbalance design: Within each difficulty level, there are 6 possibilities to order the three decompilers. The two difficulty levels are also permutated (red vs. black in the figure). We opted to balance on

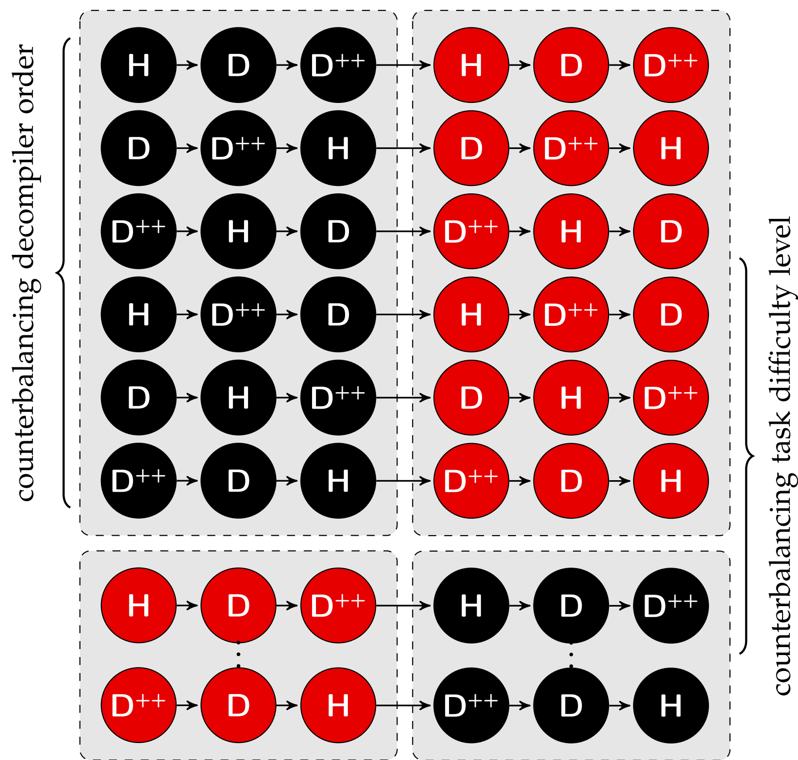


Figure 5.1: Counterbalancing the order of decompiler and difficulty levels. Nodes in each horizontal sequence represent the tasks performed by one participant. Letters denote the used decompiler for the task and colors represent task difficulty level: medium (black) or hard (red).

difficulty level instead of task level since this gives us a counterbalance permutations of 12 ($3! * 2!$) instead of 4320 ($3! * 6!$). Since we could not hope to recruit 4320 participants we opted for the compromise of recruiting multiples of 12 participants using all rows of our counter. Counterbalancing the order of difficulty level doubles the total numbers of possible orderings. This design ensures that each decompiler and each difficulty level gets the same exposure across the study and minimizes the overall learning and fatigue effects. This also guarantees that each participant gets the same number of medium and hard questions for each decompiler. This is important to control for individual differences between participants and avoid skewing the results by eliminating the possibility of a skilled and motivated participant performing all of her tasks using one decompiler, while a less skilled participant performs her tasks with another decompiler.

User Perception

After finishing each task, participants are shown a brief questionnaire, where they can score the quality of the code produced by the decompiler, and a text field for additional feedback. Here, the participants are able to see the code again. We asked a total of 8 questions, 6 on readability properties and 2 on trust issues. Similar to the System Usability Score (SUS) (SUS) [14], the questions are counterbalanced (positive/negative) to minimize the response bias, e.g., *"This code was easily readable"* and *"It was strenuous to understand what this code did"*. The full question set can be found in Table 5.2.

Statement	Strongly disagree	Disagree	Neutral	Agree	Strongly agree
This code was easily readable	○	○	○	○	○
It was strenuous to understand what this code did	○	○	○	○	○
This code looks similar to the way I would write code	○	○	○	○	○
It was hard to understand what the variables mean	○	○	○	○	○
It was pleasant to work with this code	○	○	○	○	○
I am sure that I correctly understood what this code does	○	○	○	○	○
I trust that the decompiled code is correct	○	○	○	○	○
I would rather analyze the assembly code	○	○	○	○	○

Table 5.2: Questions after each task.

Overall rating. In addition to the questionnaire after each task, at the end of the study, we asked the participants about an overall rating on a scale from 1 (worst) to 10 (best). During this step, participants were shown the code snippets for every task and every decompiler side by side to facilitate the direct comparison. To avoid biasing the participants the decompilers were named M, P and R.

Statistical Testing

For all statistical hypothesis testing, we opted for the common significance level of $\alpha = 0.05$. To account for multiple testing, all p-values are reported in the Holm-Bonferroni corrected version [50].

Continuous tests such as time intervals or user-ratings are tested with a Holm-Bonferroni corrected Mann–Whitney U test (two-tailed). Rather then testing all pairs for the pairwise comparison, we only perform tests with DREAM^{++} (DREAM^{++} vs. DREAM and DREAM^{++} vs. Hex-Rays). The effect size is reported by mean comparisons and the usage of the *common language effect size* method. Categorical contingency comparisons are tested with the two-tailed Holm-Bonferroni corrected Barnard's Exact test.

5.2 User Study

In this section, we present our study results: after discussing the demographics of our participants, we proceed with the code analysis experiment, and then the user perception discussion.

5.2.1 Participants

We sent 36 invitations to students who had completed the malware boot camp at our university with the aim of getting 24 participants to join for a compensation of 40 Euro. The malware boot camp is a lab course held each semesters at our university, in which students are introduced to the field of malware and binary code analysis. The invitation text can be found in the appendix 5.2.1.

Recruitment Advertisement

Students. To recruit our student participants, we sent the following email:

Subject: **Invitation to a Decompiler Study**

we would like to invite you to participate in a research study conducted by researchers at the University of Bonn on the quality of binary code decompilers for malware analysis. The study evaluates three state-of-the-art industrial and academic decompilers. You will be asked to complete six reverse engineering tasks. For each task you will get the decompiled code of

one function from a malware sample and a set of questions regarding its functionality. That is, you will be analysing high-level C code. The study is not aimed at testing your ability but the quality of the decompilers.

The study will take you approximately 3 hours and will take place at the University of Bonn. There will be several dates available to take part in the study. This study is anonymous, so no personally identifying information will be collected during the study and we will only report the aggregated results in a scientific publication. In appreciation of your choice to participate in the project, you will be paid 40 Euro.

Experts. To recruit our expert participants, we sent the following email:

Subject: **Invitation to a Decompiler Study**

we would like to invite you to participate in a research study conducted by researchers at the University of Bonn on the quality of binary code decompilers for malware analysis. The study evaluates three state-of-the-art industrial and academic decompilers. You will be asked to complete six reverse engineering tasks. For each task you will get the decompiled code of one function from a malware sample and a set of questions regarding its functionality. The study is not aimed at testing your ability but the quality of the decompilers.

The study will take you approximately 2 hours. You will be given the URL to our online study platform and can perform the study remotely. You can take breaks between the tasks, however the tasks themselves need to be completed uninterrupted. This study is anonymous, so no personally identifying information will be collected and we will only report the aggregated results in a scientific publication. In appreciation of your choice to participate in the project, we will pass along your comments to the developers of the decompilers for consideration. Also, you will get free access to an improved decompiler as soon as the decompiler is released. You will also be helping the malware analysis community.

Questions About Participants' Demographics

1. Gender?

- ○ Male
- ○ Female
- ○ Prefer not to answer

2. What is your age? (text field)

3. Employment Status: Are you currently?

- ○ Student
- ○ Other: (text field)

4. How many years experience do you have in malware analysis? (text field)

5. How many years experience do you have in reverse engineering? (text field)

6. Which binary code decompilers did you use before?

- ☐ Boomerang
- ☐ Hex-Rays
- ☐ REC
- ☐ DISC
- ☐ Other: (text field)

22 students took part in the study. One student began the study but only completed one task, so we removed this student from the sample, leaving us with 21 participants. Among the student group, the median age was 26 years. The oldest participant was 31 years and the youngest 19 years old. Two participants did not report their age. One of the participants was female. The median malware analysis experience in years is 1 year. One participant reported to have 14 years malware analysis experience, 7 participants reported less than a year.

We also invited 31 malware experts from commercial security companies. Based on informal talks, we were told that offering these experts the same compensation would not motivate them since time is more valuable than money. However, it was suggested that many would be intrinsically motivated to help because any improvements made in this domain would ultimately benefit them. Based on this feedback, we opted to offer early access to *an academic decompiler* for the participants and giving them access to DREAM^{++} after the study. 16 malware analysts started the study. However, 5 looked at the first task only without actually starting the study. In total 9 malware analysts took part in the study. all male with a median age of 30 (2 did not disclose their age and gender).

Decompiler	Avg. Score	p-value	Pass	Fail	p-value
Students					
DREAM^{++}	70.24		30	12	
DREAM	50.83	**0.002**	16	26	**0.002**
Hex-Rays	37.86	**<0.001**	11	31	**<0.001**
Experts					
DREAM^{++}	84.72		15	3	
DREAM	79.17	0.234	15	3	0.570
Hex-Rays	61.39	0.086	9	9	0.076

Table 5.3: Aggregated Experiment Results

5.2.2 Malware Analysis Experiment

We assigned a weight to each question according to its importance in understanding the task and scored all answers. We conducted a two-pass scoring approach for calibration purposes. Since the answers contained variable names and referred to loop structures it was not possible to blind this part of the evaluation[2]. Table 5.3 summarizes the results of the code analysis experiment. The table shows the average score achieved by each group for each decompiler and the corresponding number of successful/unsuccessful tasks. Unsurprisingly, professional analysts performed better than students. In both groups participants performed better when using DREAM^{++} compared to both DREAM and Hex-Rays. In the student group, our sample size provides sufficient statistical power to confirm the a statistically significant difference in scores. Another interesting observation is that in relation experts did much better with Hex-Rays that the students did, suggesting that they have gotten used to working with code produced by Hex-Rays. A more detailed information on the results of the students group is provided in Table 5.4. Here, we show for each task the mean and the corresponding standard deviation for the score and the time needed for each task for each decompiler. Also, the number of successful and unsuccessful task are shown.

To get a better feeling for the results, we additionally marked each task as a pass if participants scored over 70%. This level was chosen based on our judgment that these tasks had been answered sufficiently meaning that their results would be useful in a malware analysis

[2]All data will be released to the community to increase transparency.

Decompiler	Score		Time		Result	
	Mean	Stdev	Median	Stdev	Pass	Fail
Task 1						
DREAM++	45.71%	32.78	8.06	3.01	3	4
DREAM	56.43%	31.02	28.55	10.69	4	3
Hex-Rays	25.71%	36.3	57.42	23.65	2	5
Task 2						
DREAM++	71.88%	28.72	29.88	17.55	6	2
DREAM	61.67%	30.51	29.72	6.68	3	3
Hex-Rays	29.29%	33.85	18.58	0.0	1	6
Task 3						
DREAM++	80.83%	15.92	13.96	4.43	5	1
DREAM	32.5%	26.81	31.6	0.0	1	7
Hex-Rays	30.71%	26.65	26.94	0.0	1	6
Task 4						
DREAM++	74.29%	23.06	31.04	7.31	5	2
DREAM	55.71%	28.71	22.52	3.17	2	5
Hex-Rays	43.57%	27.87	50.34	10.86	2	5
Task 5						
DREAM++	81.25%	11.92	22.85	9.67	7	1
DREAM	80.0%	18.48	27.12	6.32	5	1
Hex-Rays	68.57%	21.99	37.13	15.65	4	3
Task 6						
DREAM++	66.67%	23.92	28.97	5.81	4	2
DREAM	30.0%	20.62	53.63	0.0	1	7
Hex-Rays	29.29%	27.31	44.12	0.0	1	6
Decompiler	**Mean**	**Stdev**	**Median**	**Stdev**	**Pass**	**Fail**

Table 5.4: Detailed study results for all tasks for the students group.

team. Here, the difference between the professional analysts and the students becomes more apparent. However, in both groups DREAM^{++} performed better than the competition.

We also measured the time needed to complete the different tasks. We cannot do a statistical analysis of the time means because of a limited number of samples. Note, that only successfully completed a task can be considered in that comparison. Unfortunately, many participants failed or gave up tasks using Hex-Rays and DREAM. Nonetheless, there is a trend for participants solving tasks faster with DREAM^{++}, but more samples are needed to quantify this reliably. A detailed overview on task level, including average time spend on a task, can be

found in Table 5.4. The table shows the detailed study results for the student participants. For each task, the average score achieved and corresponding standard deviation are shown. Next, we show average time needed to complete the task and the corresponding standard deviations. Finally, the number of task that was solved successfully/unsuccessfully are mentioned.

5.2.3 User Perception

To measure user perception, after each task, we asked our participants 8 questions. The full list of questions can be seen in Table 5.2. Figure 5.2 summarizes the aggregated user perception results. Here, we distinguish between two groups of questions: usability perception (questions 1-6) and trust perception (questions 7&8). The trust issue is interesting because decompilers do make mistakes or create misleading code and it is common for malware analysts to fall back to analyzing assembly code. Thus, we wanted to see whether there were different levels of trust in the code.

A pairwise comparison between the decompilers shows a high statistical significance difference: $p < 0.001$ for usability related questions for experts and students. The trust related questions showed only a statistically significant difference in comparison to Hex-Rays ($p < 0.001$ in the student group, $p = 0.03$ in the expert group). The color coding shows the agreement level and the percentage numbers on the right and left hand side summarize the percentage of participants who rated positively and negatively respectively.

Detailed user perception. Figure 5.3 shows the participant agreement with each of the statements in Table 5.2.

Overall rating. As explained in Section 5.1.3, after finishing all tasks, we asked the participant to provide an overall rating for each decompiler depending on how they evaluate the readability of the decompiled code. This rating is on a scale from 1 (worst) to 10 (best). Figure 5.4 shows a box plot with a rating distribution overview. It can be clearly seen that DREAM^{++} achieves higher scores than the competition for both, students and experts (adjusted $p < 0.001$ for all pairwise comparisons). Interestingly, while it can be clearly seen that the experts rated Hex-Rays better compared to the students, they were also more enthusiastic about DREAM^{++}. These are very promising results, both groups clearly prefer the code produced by our im-

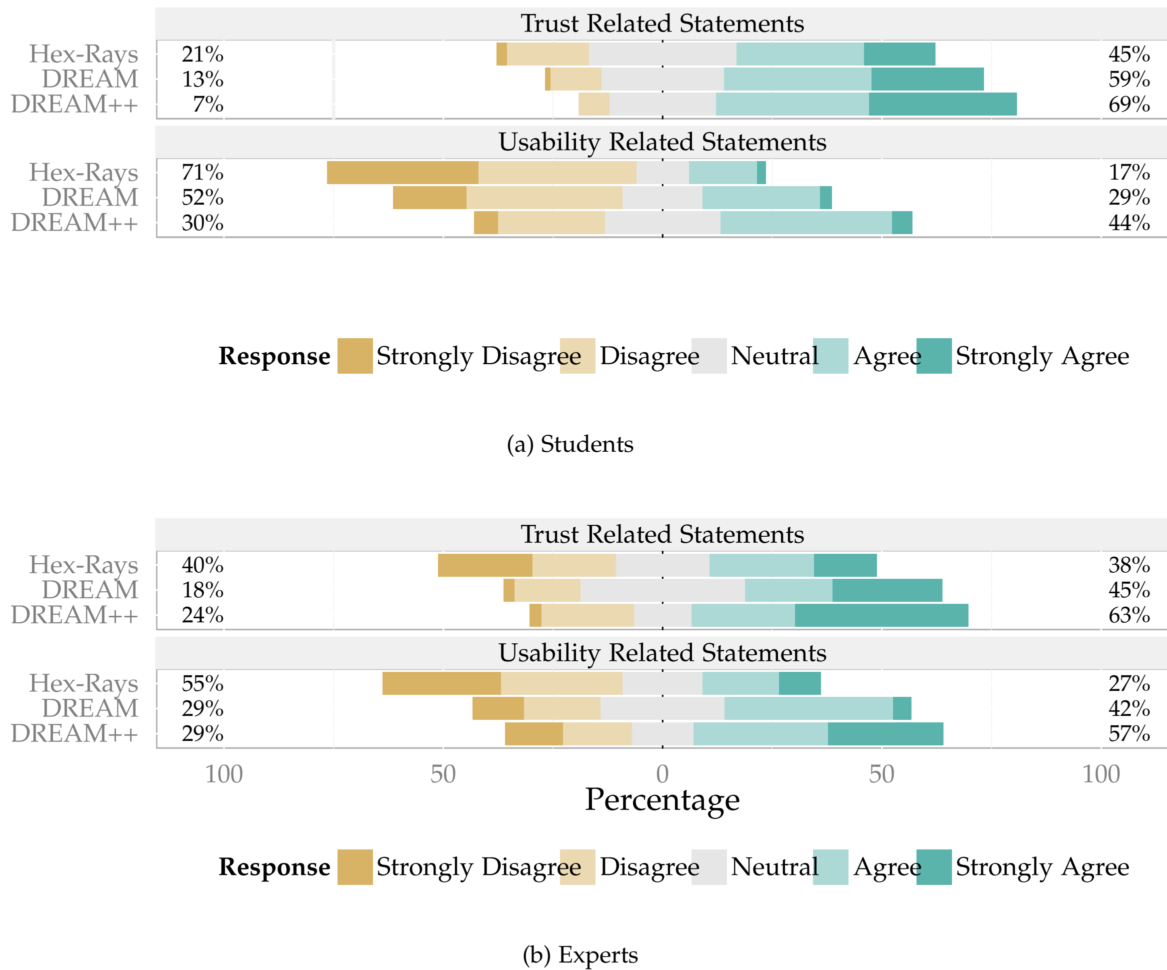


Figure 5.2: Aggregated participant agreement with the statements related to usability perception (6 questions) and trust in correctness (2 questions).

proved decompiler and even though the experts cope well with Hex-Rays they gave DREAM^{++} outstanding marks.

5.3 Related Work

To the best of our knowledge, we are the first to conduct user studies in the domain of malware analysis.

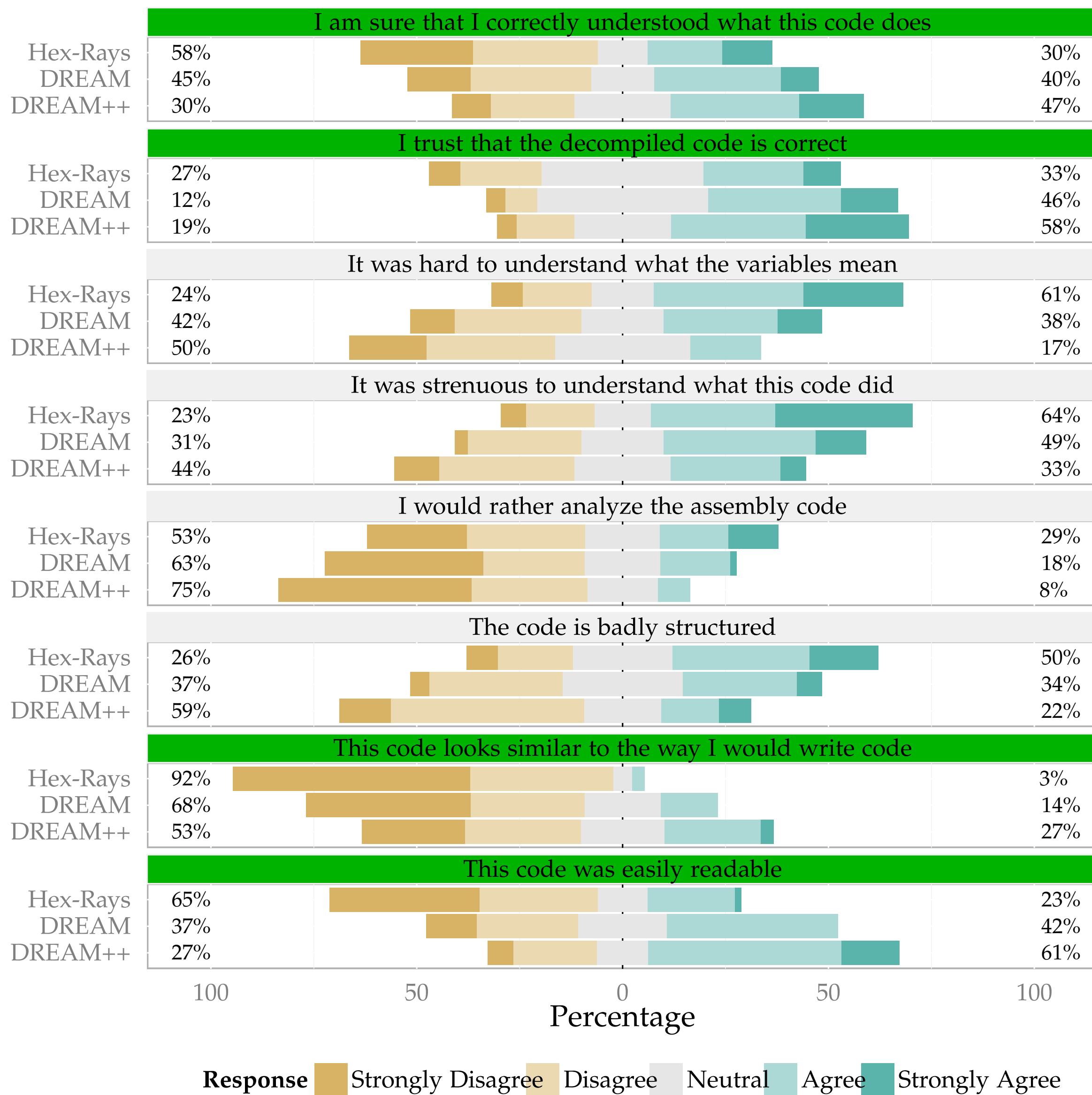


Figure 5.3: Participant agreement with the statements from Table 5.2. The positive statements are marked in green and the negative statements are marked in gray.

5.4 Summary

In this chapter, we validated our readability improvements with the first user study in the domain of reverse engineering, involving both students and professional malware analysts. The results clearly show that our human-focused approach to decompilation offers significant

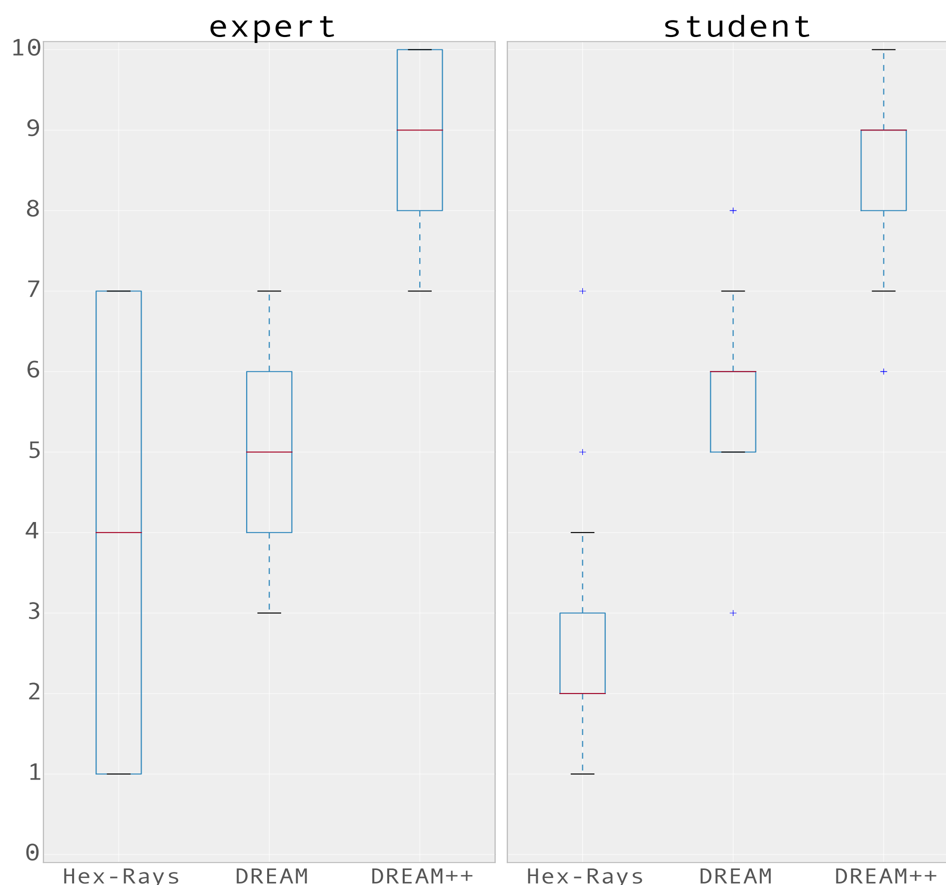


Figure 5.4: Boxplot for Rating

improvements, with DREAM^{++} outperforming both DREAM and Hex-Rays. Despite these large improvements, we believe we have only barely scratched the surface of what can be done in this highly technical domain. We believe that research community should focus more on the experts involved in this field. We hope that our user study can serve as a template for similar studies in malware analysis.

6 Conclusion and Future Work

6.1 Conclusion

Decompilation provides an attractive method to assist the analysis of the binary code. By recovering high-level abstractions from the binary code, both manual and automatic security analyses can be performed on a more abstract high-level representation of the binary code. This leads to a better efficiency and performance since the analysis can leverage the high-level abstractions available in the source code. The main focus of this thesis is devise techniques to improve the readability of the decompiler code to assist human analysis in the analysis of binary code. To achieve that goal, the thesis introduces the following techniques:

Novel control-flow structuring techniques. We presented a pattern-independent control-flow structuring algorithm to recover control-flow structure of a program. The algorithm is a departure from the traditional patter-matching approach of structure analysis. The main feature of this algorithm is that it produces a fully-structured code, making DREAM the first decompiler that produces a `goto`-free output.

Usability extensions for better readability. We developed several semantics-preserving code transformations to improve the readability of the decompiled code. These transformations leverage the high-level abstractions recovered during previous decompilation steps to simplify expressions and control-flow structure. Also, by analyzing the context in which variables are used in the program, they are assigned meaningful names that reveal the purpose of these variables.

First user study to evaluate decompiler quality. We designed and performed the first user study to evaluate the quality of decompiler for binary code analysis. Our study included six tasks from real malware samples. The participants included both students with little experience with malware analysis and malware experts. The results of our study clearly show that our human-focused approach to decompilation offers significant improvements over existing solutions. We hope that this study will serve as a template for user studies for decompilation research.

6.2 Future Work

While the results of evaluating the methods developed in this thesis are very promising, several directions for future work exit.

Independence of IDA. Currently, we rely on IDA Pro for code extractions and variable recovery and partially for type analysis. IDA rely on certain assumptions regarding the calling convention of binary functions to detect arguments and local variables. Also, IDA seems to guess types of variables in many situations. Future work includes replacing IDA with new promising tools such as `angr` [80] which has seen active use in the academia recently [85, 79, 65, 67, 90], `radare` [70], or REV.NG [35, 36].

Use existing intermediate representations. Several intermediate representations for program analysis exist. This includes VEX IR from the Valgrind framework and LLVM IR from the LLVM compiler infrastructure. LLVM IR is particularly interesting due to the existing of a complete compiler framework that uses this IR. This opens the door to reuse a wealth of existing compiler optimizations for decompilation. This can be very useful when dealing with obfuscated code as shown in [69].

Interactive decompilation. Currently, our decompiler only outputs the decompiled code to the user. An interesting and very useful addition would be to include user input in the decompilation process. This interaction enables experts to give hints to the decompiler that can be used to improve and correct its analyses.

Exploring new applications. Manual binary code analysis has been the primary driving force behind this work. Exploring other applications is planned for future work. This includes the

identification of bugs in binary code using source code techniques. This is possible by using the decompiler to produce a high-level source code upon which the source-based techniques can be applied.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition, 2006.

[2] Frances E. Allen. Control Flow Analysis. In *Proceedings of ACM Symposium on Compiler Optimization*, 1970.

[3] Jens Palsberg Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[4] Dennis Andriesse, Christian Rossow, Brett Stone-Gross, Daniel Plohmann, and Herbert 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] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[6] Gogul Balakrishnan. *WYSINWYX What You See Is Not What You eXecute*. PhD thesis, University of Wisconsin at Madison, 2007.

[7] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[8] Thomas Barabosch, Adrian Dombeck, Khaled Yakdan, and Elmar Gerhards-Padilla. Bot-Watcher: Transparent and Generic Botnet Tracking. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2015.

[9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, February 2010.

[10] Dirk Beyer. Relational Programming with CrocoPat. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[11] Dirk Beyer, Andreas Noack, and Claus Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering (TSE)*, 31(2), 2005.

[12] Guillaume Bonfante, Jose Fernandez, Jean-Yves Marion, Benjamin Rouxel, Fabrice Sabatier, and Aurélien Thierry. CoDisasm: Medium Scale Concatic Disassembly of Self-Modifying Binaries with Overlapping Instructions. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[13] Erik Bosman, Asia Slowinska, and Herbert Bos. Minemu: The World's Fastest Taint Tracker. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection (RAID)*, 2011.

[14] John Brooke. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[15] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. RICH: Automatically Protecting Against Integer-Based Vulnerabilities. In *Proceedings of the 14th Network and Distributed System Security Symposium (NDSS)*, 2007.

[16] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. BAP: A Binary Analysis Platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV)*, 2011.

[17] Raymond P. L. Buse and Westley R. Weimer. Learning a Metric for Code Readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, July 2010.

[18] Juan Caballero, Chris Grier, Christian Kreibich, and Vern Paxson. Measuring Pay-per-Install: The Commoditization of Malware Distribution. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[19] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling Active Botnet Infiltration Using Automatic Protocol Reverse-Engineering. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, 2009.

[20] Joan Calvet, José M. Fernandez, and Jean-Yves Marion. Aligot: Cryptographic Function Identification in Obfuscated Binary Programs. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[21] Francois Chagnon. Decompiler. `https://github.com/EiNSTeiN-/decompiler`. Page checked 8/20/2015.

[22] Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula. Analysis of Low-level Code Using Cooperating Decompilers. In *Proceedings of the 13th International Conference on Static Analysis (SAS)*, 2006.

[23] Walter Chang, Brandon Streiff, and Calvin 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.

[24] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[25] Cristina Cifuentes. Structuring Decompiled Graphs. In *Proceedings of the 6th International Conference on Compiler Construction (CC)*, 1996.

[26] Cristina Cifuentes, Doug Simon, and Antoine Fraboulet. Assembly to High-Level Language Translation. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, 1998.

[27] John Cocke. Global Common Subexpression Elimination. In *Proceedings of the ACM Symposium on Compiler Optimization*, 1970.

[28] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical report, Department of Computer Sciences, The University of Auckland, 1997.

[29] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of Virtualization-obfuscated Software: A Semantics-based Approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[30] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[31] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 13:451–490, 1991.

[32] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[33] The decompilation wiki. `http://www.program-transformation.org/Transform/DeCompilation`.

[34] David Dewey and Jonathon 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.

[35] Alessandro Di Federico and Giovanni Agosta. A jump-target identification method for multi-architecture static binary translation. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2016.

[36] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. REV.NG: A Unified Binary Analysis Framework to Recover CFGs and Function Boundaries. In *26th International Conference on Compiler Construction (CC)*, 2017.

[37] Edsger W. Dijkstra. Letters to the Editor: Go to Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, March 1968.

[38] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1976.

[39] Michael James Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007.

[40] Felix Engel, Rainer Leupers, Gerd Ascheid, Max Ferger, and Marcel 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.

[41] Ana Erosa and Laurie J. Hendren. Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages*, 1994.

[42] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS)*, 2016.

[43] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet Dossier. Symantec Corporation, 2011.

[44] Alexander Fokin, Egor Derevenetc, Alexander Chernov, and Katerina Troshina. SmartDec: Approaching C++ Decompilation. In *Proceedings of the 2011 18th Working Conference on Reverse Engineering (WCRE)*, 2011.

[45] Alexander Fokin, Katerina Troshina, and Alexander Chernov. Reconstruction of Class Hierarchies for Decompilation of C++ Programs. In *Proceedings of the 14th European Conference on Software Maintenance and Reengineering (CSMR)*, 2010.

[46] G Data SecurityLabs. Uroburos Highly complex espionage software with Russian roots. G Data Software AG, 2014.

[47] Ilfak Guilfanov. Decompilers and Beyond. In *Black Hat, USA*, 2008.

[48] Istvan Haller, Asia Slowinska, and Herbert Bos. MemPick: High-Level Data Structure Detection in C/C++ Binaries. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, 2013.

[49] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[50] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.

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

[52] Wesley Jin, Cory Cohen, Jeffrey Gennari, Charles Hines, Sagar Chaki, Arie Gurfinkel, Jeffrey Havrilla, and Priya 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.

[53] Johannes Kinder and Helmut Veith. Jakstab: A Static Analysis Platform for Binaries. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*, 2008.

[54] Günter Kniesel, Jan Hannemann, and Tobias Rho. A Comparison of Logic-based Infrastructures for Concern Detection and Extraction. In *Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution (LATE)*, 2007.

[55] 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.

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

[57] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The Soot framework for Java program analysis: a retrospective. In *Proceedings of the Cetus Users and Compiler Infrastructure Workshop (CETUS)*, 2011.

[58] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. In *Proceedings of the 18th Network and Distributed System Security Symposium (NDSS)*, 2011.

[59] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[60] Jerome Miecznikowski and Laurie Hendren. Decompiling Java Using Staged Encapsulation. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*, 2001.

[61] Jerome Miecznikowski and Laurie J. Hendren. Decompiling Java Bytecode: Problems, Traps and Pitfalls. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*, 2002.

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

[63] Nomair A. Naeem. Programmer-Friendly Decompiled Java. Master's thesis, McGill University, August 2006.

[64] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic Type Inference for Machine Code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[65] Muhammad Riyad Parvez. Combining Static Analysis and Targeted Symbolic Execution for Scalable Bug-finding in Application Binaries. Master's thesis, University of Waterloo, 2016.

[66] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. X-Force: Force-Executing Binary Programs for Security Applications. In *Proceedings of the 23rd USENIX Security Symposium*, 2014.

[67] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-Architecture Bug Search in Binary Executables. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 709–724. IEEE, 2015.

[68] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A Comprehensive Measurement Study of Domain Generating Malware. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[69] Nguyen Anh Quynh. OptiCode: Machine Code Deobfuscation for Malware Analysis. In *SyScan*, 2013.

[70] radare2: unix-like reverse engineering framework and commandline tools. `http://www.radare.org/`.

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

[72] H. G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.

[73] Ed Robbins, Andy King, and Tom Schrijvers. From MinX to MinC: Semantics-driven Decompilation of Recursive Datatypes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016.

[74] Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J. Dietrich, and Herbert 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.

[75] Masataka Sassa, Yo Ito, and Masaki Kohama. Comparison and evaluation of back-translation algorithms for static single assignment forms. *Computer Languages, Systems & Structures*, 35(2):173–195, 2009.

[76] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 Decompilation using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[77] M. Sharir. Structural Analysis: A New Approach to Flow Analysis in Optimizing Compilers. *Computer Languages*, 5(3-4):141–153, January 1980.

[78] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium*, 2015.

[79] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. In *Proceedings of the 2015 Network and Distributed System Security Symposium*, 2015.

[80] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[81] Doug Simon. Structuring Assembly Programs. Honours thesis, University of Queensland, 1997.

[82] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A Dynamic Excavator for Reverse Engineering Data Structures. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, 2011.

[83] 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 *Proceedings of the 4th International Conference on Information Systems Security (ICISS)*, 2008.

[84] Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *In Static Analysis Symposium, Italy*, pages 194–210. Springer Verlag, 1999.

[85] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Aug-

menting Fuzzing Through Selective Symbolic Execution. In *Proceedings of the 2016 Network and Distributed System Security Symposium*, 2016.

[86] Symantec Security Response. Regin: Top-tier espionage tool enables stealthy surveillance, 2014.

[87] The LLVM Compiler Infrastructure. `http://llvm.org`.

[88] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G. Bringas. [SoK] Deep Packer Inspection: A Longitudinal Study of the Complexity of Run-Time Packers (S&P). In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[89] Raja Vallee-rai and Laurie Hendren. Jimple: Simplifying Java Bytecode for Analyses and Transformations. Technical report, Sable Research Group, McGill University, 1998.

[90] Sebastian Vogl, Robert Gawlik, Behrad Garmany, Thomas Kittel, Jonas Pfoh, Claudia Eckert, and Thorsten Holz. Dynamic hooks: hiding control flow changes within non-control data. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 813–328, 2014.

[91] Xi Wang, Haogang Chen, Zhihao Jia, Nickolai Zeldovich, and M. Frans Kaashoek. Improving Integer Security for Systems with KINT. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2012.

[92] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

[93] M. H. Williams. Generating structured flow diagrams: the nature of unstructuredness. *The Computer Journal*, 20(1):45–50, 1977.

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

[95] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, and Engin Kirda. Automatic Network Protocol Analysis. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[96] Christian Wressnegger, Fabian Yamaguchi, Alwin Maier, and Konrad Rieck. Twice the Bits, Twice the Trouble: Vulnerabilities Induced by Migrating to 64-Bit Platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security(CCS)*, 2016.

[97] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, 2001.

[98] Zhaoyan Xu, Jialong Zhang, Guofei Gu, and Zhiqiang Lin. GoldenEye: Efficiently and Effectively Unveiling Malware's Targeted Environment. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2014.

[99] Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, and Saumya Debray. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[100] 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 *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, 2016.

[101] Khaled Yakdan, Sebastian Eschweiler, and Elmar 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.

[102] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the 22nd Network and Distributed System Security (NDSS) Symposium*, 2015. **Distinguished Paper Award**.

[103] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. In *Proceedings of the 22nd Network and Distributed System Security (NDSS) Symposium*, 2015.

[104] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[105] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, 2015.

[106] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad 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.

[107] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Obfuscation Resilient Binary Code Reuse Through Trace-oriented Programming. In *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, 2013.

A Code Snippets in the User Study

A.1 Task 1

A.1.1 Hex-Rays

Listing A.1 shows the decompiled code produced by Hex-Rays for the first task. For this task, the following questions were asked:

1. What are the data types of the parameters and what are their meanings?
2. What is the purpose of this function?

Listing A.1: Decompiled code generated by Hex-Rays for Task 1

```
unsigned int __usercall sub_10001103@<eax>(int a1@<ecx>, unsigned int a2@<edi>)
{
  unsigned int v2; // edx@2
  unsigned int v3; // eax@4
  unsigned int result; // eax@6
  signed int v5; // [sp+8h] [bp-8h]@1
  unsigned int v6; // [sp+Ch] [bp-4h]@1

  v6 = a2 >> 1;
  v5 = 4;
  do
  {
    v2 = 0;
    if ( a2 )
    {
      do
      {
        *(_BYTE *)(v2 + a1) ^= -106 * (_BYTE)v2;
```

```
19          ++v2;
20        }
21        while ( v2 < a2 );
22      }
23      v3 = 0;
24      if ( v6 )
25      {
26        do
27        {
28          *(_BYTE *)(v3 + a1) ^= *(_BYTE *)(((a2 + 1) >> 1) + a1 + v3);
29          ++v3;
30        }
31        while ( v3 < v6 );
32      }
33      for ( result = a2 - 1; result >= 1; --result )
34        *(_BYTE *)(result + a1) -= *(_BYTE *)(result + a1 - 1);
35      --v5;
36    }
37    while ( v5 >= 0 );
38    return result;
39  }
```

A.1.2 Dream

Listing A.2 shows the decompiled code produced by DREAM for the first task. For this task, the following questions were asked:

1. What are the data types of the parameters and what are their meanings?
2. What is the purpose of this function?

Listing A.2: Decompiled code generated by DREAM for Task 1

```
1   int sub_10001103(int a1, char * a2){
2     int v1 = a1 >> 1;
3     int v2 = 4;
4     int result;
5     do{
6       int v3 = 0;
7       if(a1)
8         do{
9           *(v3 + a2) ^= 150 * v3;
10          v3++;
11        }while(v3 < a1);
12      int v4 = 0;
13      if(v1 > 0)
14        do{
```

```
15          *(v4 + a2) ^= *(((a1 + 1) >> 1) + a2 + v4);
16          v4++;
17        }while(v4 < v1);
18      for(result = a1 - 1; result >= 1; result--)
19        *(result + a2) -= *(result + a2 - 1);
20      v2--;
21    }while(v2 >= 0);
22    return result;
23  }
```

A.1.3 Dream^{++}

Listing A.3 shows the decompiled code produced by DREAM^{++} for the first task. For this task, the following questions were asked:

1. What are the data types of the parameters and what are their meanings?
2. What is the purpose of this function?

Listing A.3: Decompiled code generated by DREAM^{++} for Task 1

```
1  int sub_10001103(int a1, char * a2){
2    int result;
3    for(int counter1 = 4; counter1 >= 0; counter1--){
4      for(int i = 0; i < a1; i++)
5        a2[i] ^= 150 * i;
6      for(int i = 0; i < a1 / 2; i++)
7        a2[i] ^= a2[((a1 + 1) / 2) + i];
8      for(result = a1 - 1; result >= 1; result--)
9        a2[result] -= a2[result - 1];
10   }
11   return result;
12 }
```

A.2 Task 2

A.2.1 Hex-Rays

Listing A.4 shows the decompiled code produced by Hex-Rays for the second task. For this task, the following questions were asked:

1. What are the data types of the parameters and what are their meanings?

2. What is the purpose of this function?

3. Asume that a1 points to the array `"\x7B\xAE\x7C\xAE\x74\xAE\x7D\xAE\x12\xAE"` (a1[0] = 0x7B, a1[1] = 0xAE, a1[2] = 0x7C, etc.), what will be the values in the buffer pointed to by a2?

Listing A.4: Decompiled code generated by Hex-Rays for Task 2

```
__int16 __cdecl sub_10001F81(int a1, int a2)
{
  int v2; // ecx@1
  __int16 result; // ax@2
  bool v4; // zf@3
  int v5; // edx@3

  v2 = a1;
  if ( a1 )
  {
    result = *(_WORD *)a1 ^ 0xAE12;
    v4 = *(_WORD *)a1 == -20974;
    v5 = a2;
    for ( *(_WORD *)a2 = result; !v4; *(_WORD *)v5 = result )
    {
      v2 += 2;
      v5 += 2;
      result = *(_WORD *)v2 ^ 0xAE12;
      v4 = *(_WORD *)v2 == -20974;
    }
  }
  else
  {
    result = 0;
    *(_WORD *)a2 = 0;
  }
  return result;
}
```

A.2.2 Dream

Listing A.5 shows the decompiled code produced by DREAM for the second task. For this task, the following questions were asked:

1. What are the data types of the parameters and what are their meanings?

2. What is the purpose of this function?

3. Asume that a1 points to the array `"\x7B\xAE\x7C\xAE\x74\xAE\x7D\xAE\x12\xAE"` (a1[0] = 0x7B, a1[1] = 0xAE, a1[2] = 0x7C, etc.), what will be the values in the buffer pointed to by a2?

Listing A.5: Decompiled code generated by DREAM for Task 2

```
1  short sub_10001F81(void * a1, void * a2){
2    short * v1 = a1;
3    if(!a1){
4      *a2 = 0;
5      return 0;
6    }
7    short result = *a1 ^ 0xae12;
8    short * v2 = a2;
9    *a2 = result;
10   if(result)
11     do{
12       v1 += 2;
13       short v3 = *v1;
14       v2 += 2;
15       result = v3 ^ 0xae12;
16       *v2 = result;
17     }while(result);
18   return result;
19 }
```

A.2.3 Dream^{++}

Listing A.6 shows the decompiled code produced by DREAM^{++} for the second task. For this task, the following questions were asked:

1. What are the data types of the parameters and what are their meanings?
2. What is the purpose of this function?
3. Asume that a1 points to the array `"\x7B\xAE\x7C\xAE\x74\xAE\x7D\xAE\x12\xAE"` (a1[0] = 0x7B, a1[1] = 0xAE, a1[2] = 0x7C, etc.), what will be the values in the buffer pointed to by a2?

Listing A.6: Decompiled code generated by DREAM^{++} for Task 2

```
short sub_10001F81(short * a1, short * a2){
  short * v1 = a1;
  if(a1 == 0){
    *a2 = 0;
    return 0;
  }
  short result = *a1 ^ 0xAE12;
  short * v2 = a2;
  *a2 = result;
  while(result != 0){
    v1 += 2;
    v2 += 2;
    result = *v1 ^ 0xAE12;
    *v2 = result;
  }
  return result;
}
```

A.3 Task 3

A.3.1 Hex-Rays

Listing A.7 shows the decompiled code produced by Hex-Rays for the third task. For this task, the following questions were asked:

1. For each iteration of the loop that starts at line 34, variable v7 gets a pointer to a string. What does the loop between lines 39-45 do?
2. What does the loop between lines 67-87 do?
3. What are the data types of the parameters and what are their meanings?
4. What is the condition under which line 90 will be executed?

Listing A.7: Decompiled code generated by Hex-Rays for Task 3

```
int __usercall sub_408A70@<eax>(int a1@<ebx>, int a2)
{
  int v2; // ebp@1
  int v3; // esi@1
  unsigned int v4; // eax@1
  int v5; // esi@1
  int v6; // edi@2
  int v7; // ecx@3
```

```
9    int v8; // edx@5
10   int v9; // eax@7
11   int v10; // edi@10
12   int v11; // esi@11
13   char v12; // al@12
14   int v13; // edx@14
15   int v14; // eax@16
16   bool v15; // cf@24
17   int result; // eax@25
18   int v17; // [sp+Ch] [bp-Ch]@2
19   int v18; // [sp+10h] [bp-8h]@1
20   unsigned int v19; // [sp+14h] [bp-4h]@1
21   int v20; // [sp+1Ch] [bp+4h]@1
22
23   v2 = a2;
24   v3 = *(_DWORD *)(*(_DWORD *)(a2 + 60) + a2 + 120);
25   v4 = *(_DWORD *)(v3 + a2 + 24);
26   v5 = a2 + v3;
27   v18 = v5;
28   v20 = 0;
29   v19 = v4;
30   if ( v4 )
31   {
32     v6 = v2 + *(_DWORD *)(v5 + 32);
33     v17 = v2 + *(_DWORD *)(v5 + 32);
34     while ( 1 )
35     {
36       v7 = v2 + *(_DWORD *)v6;
37       if ( v7 && a1 )
38       {
39         v8 = 0;
40         if ( *(_BYTE *)v7 )
41         {
42           do
43             ++v8;
44           while ( *(_BYTE *)(v8 + v7) );
45         }
46         v9 = 0;
47         if ( *(_BYTE *)a1 )
48         {
49           do
50             ++v9;
51           while ( *(_BYTE *)(v9 + a1) );
52         }
53         if ( v8 == v9 )
54           break;
55       }
56 LABEL_24:
57       v6 += 4;
58       v15 = v20++ + 1 < v19;
59       v17 = v6;
60       if ( !v15 )
```

```
61          goto LABEL_25;
62      }
63      v10 = v8 + v7;
64      if ( v7 < (unsigned int)(v8 + v7) )
65      {
66        v11 = a1 - v7;
67        do
68        {
69          v12 = *(_BYTE *)v7;
70          if ( *(_BYTE *)v7 < 65 || v12 > 90 )
71            v13 = v12;
72          else
73            v13 = v12 + 32;
74          LOBYTE(v14) = *(_BYTE *)(v11 + v7);
75          if ( (char)v14 < 65 || (char)v14 > 90 )
76            v14 = (char)v14;
77          else
78            v14 = (char)v14 + 32;
79          if ( v13 != v14 )
80          {
81            v5 = v18;
82            v6 = v17;
83            goto LABEL_24;
84          }
85          ++v7;
86        }
87        while ( v7 < (unsigned int)v10 );
88        v5 = v18;
89      }
90      result = v2 + *(_DWORD *)(*(_DWORD *)(v5 + 28) + 4 * *(_WORD *)(*(_DWORD *)(v5 + 36) +
            2 * v20 + v2) + v2);
91    }
92    else
93    {
94  LABEL_25:
95      result = 0;
96    }
97    return result;
98  }
```

A.3.2 Dream

Listing A.8 shows the decompiled code produced by Dream for the third task. For this task, the following questions were asked:

1. For each iteration of the loop that starts at line 12, variable v11 gets a pointer to a string. What does the loop between lines 21-25 do?

2. What does the loop between lines 36-53 do?
3. What are the data types of the parameters and what are their meanings?
4. What is the condition under which line 70 will be executed?

Listing A.8: Decompiled code generated by DREAM for Task 3

```
1  int sub_408A70(void * a1, void * a2){
2    void * v1 = a1;
3    int v2 = *(*(a1 + 60) + v1 + 120);
4    int v3 = *(v2 + v1 + 24);
5    void * v4 = v2 + a1;
6    void * v5 = v4;
7    int v6 = 0;
8    if(v3){
9      int v7 = *(v4 + 32);
10     void * v8 = v7 + a1;
11     void * v9 = v7 + a1;
12     while(1){
13       void * v10 = v8;
14       void * v11 = *v10 + a1;
15       bool cond1 = !v11;
16       int v20;
17       bool cond2 = false;
18       int v13;
19       int v12;
20       if(!cond1 && a2){
21         v12 = 0;
22         if(*v11 != v12)
23           do
24             v12++;
25           while(*(v12 + v11));
26         v13 = 0;
27         if(*a2 != v13)
28           do
29             v13++;
30           while(*(v13 + a2));
31         if(v12 == v13){
32           int v14 = v12 + v11;
33           cond2 = v11 >= v14;
34           if(!cond2){
35             int v15 = a2 - v11;
36             do{
37               char v16 = *v11;
38               int v17;
39               if(v16 > 90 || v16 < 65)
40                 v17 = v16;
41               else
42                 v17 = v16 + 32;
```

```
43              int v19;
44              char v18 = *(v15 + v11);
45              if(v18 < 65 || v18 > 90)
46                v19 = v18;
47              else
48                v19 = v18 + 32;
49              v20 = v17 - v19;
50              if(v20)
51                break;
52              v11++;
53            }while(v11 < v14);
54            if(v20 || v20){
55              v4 = v5;
56              v10 = v9;
57            } else
58              v4 = v5;
59          }
60        }
61      }
62      if(cond1 || !a2 || v12 != v13 || (v20 && !cond2) || (v20 && !cond2)){
63        int v21 = v6 + 1;
64        v8 = v10 + 4;
65        v6 = v21;
66        v9 = v10 + 4;
67        if(v21 >= v3)
68          break;
69      } else
70        return *(*(v4 + 28) + (*(*(v4 + 36) + (v6 * 2) + a1) * 4) + a1) + a1;
71    }
72  }
73  return 0;
74 }
```

A.3.3 Dream^{++}

Listing A.9 shows the decompiled code produced by Dream^{++} for the third task. For this task, the following questions were asked:

1. For each iteration of the loop that starts at line 10, variable str gets a pointer to a string. What happens at line 14?
2. What happens at line 20?
3. What are the data types of the parameters and what are their meanings?
4. What is the condition under which line 22 will be executed?

Listing A.9: Decompiled code generated by DREAM^{++} for Task 3

```
int sub_408A70(void * a1, const char * str1){
  int v1 = *(*(a1 + 60) + a1 + 120);
  int v2 = *(v1 + a1 + 24);
  void * v3 = v1 + a1;
  int counter1 = 0;
  if(v2 != 0){
    int v4 = *(v3 + 32);
    void * v5 = v4 + a1;
    void * v6 = v4 + a1;
    do{
      v6 = v5;
      const char * str = *v6 + a1;
      if(str != 0 && str1 != 0){
        size_t len = strlen(str);
        if(len == strlen(str1)){
          int v7 = len + str;
          bool cond1 = str >= v7;
          int v8;
          if(!cond1)
            v8 = strncmpi(str, str1, v7 - str);
          if(cond1 || v8 == 0)
            return *(*(v3 + 28) + (*(*(v3 + 36) + (counter1 * 2) + a1) * 4) + a1) + a1;
        }
      }
      counter1++;
      v5 = v6 + 4;
      v6 += 4;
    }while(counter1 < v2);
  }
  return 0;
}
```

A.4 Task 4

A.4.1 Hex-Rays

Listing A.10 shows the decompiled code produced by Hex-Rays for the fourth task. For this task, the following questions were asked:

1. What are the data types of the parameters?
2. What is the purpose of this function?
3. What is stored in the return value (variable v10)?

4. What are the values that the arguments a1 and a2 points to after executing the function with the input string a1 = `"The quick brown %FOX% jumps over the lazy dog"`?
5. What are the values that the arguments a1 and a2 points to after executing the function with the input string a1 = `"The %QU1CK% brown %FOX% jumps %OVER% the lazy dog"`?
6. What are the values that the arguments a1 and a2 points to after executing the function with the input string a1 = `"My %P@SSWORD% is %INFECTED%"`?

Listing A.10: Decompiled code generated by Hex-Rays for Task 4

```
int __usercall sub_404E14@<eax>(int a1@<eax>, int a2@<edx>)
{
  int v2; // ebx@1
  int v3; // eax@2
  int v4; // edi@3
  int v5; // esi@3
  int v6; // eax@3
  bool v7; // dl@10
  int v9; // [sp+Ch] [bp-Ch]@3
  int v10; // [sp+10h] [bp-8h]@1
  int v11; // [sp+14h] [bp-4h]@1

  v11 = a2;
  v2 = a1;
  v10 = 0;
  if ( !a1 )
    return v10;
  v3 = strlen(a1);
  if ( v3 < 3 )
    return v10;
  v4 = 0;
  v9 = 0;
  v5 = v3;
  v6 = 0;
  while ( 1 )
  {
    if ( *(v2 + v6) != '%' )
    {
      v7 = (*(v2 + v6) - 0x30) < 0xAu || (*(v2 + v6) - 0x41) < 0x1Au;
      if ( !v7 )
      {
        v4 = 0;
        v9 = 0;
      }
      goto LABEL_14;
    }
    if ( v4 )
```

```
      break;
    v4 = v2 + v6;
LABEL_14:
    ++v6;
    --v5;
    if ( !v5 )
      goto LABEL_15;
  }
  v9 = v2 + v6;
LABEL_15:
  if ( v4 && v9 )
  {
    *v11 = v9 - v4 + 1;
    v10 = v4;
  }
  return v10;
}
```

A.4.2 Dream

Listing A.11 shows the decompiled code produced by DREAM for the fourth task. For this task, the following questions were asked:

1. What are the data types of the parameters?
2. What is the purpose of this function?
3. What is stored in the return value (variable result)?
4. What are the values that the arguments str and a1 points to after executing the function with the input string str = `"The quick brown %FOX% jumps over the lazy dog"`?
5. What are the values that the arguments str and a1 points to after executing the function with the input string str = `"The %QU1CK% brown %FOX% jumps %OVER% the lazy dog"`?
6. What are the values that the arguments str and a1 points to after executing the function with the input string str = `"My %P@SSWORD% is %INFECTED%"`?

Listing A.11: Decompiled code generated by DREAM for Task 4

```
int sub_404E14(const char * str, void * a1){
  int result = 0;
  if(!str)
    return result;
```

```
5     size_t len = strlen(str);
6     if(len < 3)
7       return result;
8     int v1 = 0;
9     int v2 = 0;
10    int v3 = len;
11    int v4 = 0;
12    do{
13      if(*(str + v4) != 37){
14        char v5 = *(str + v4);
15        if(v5 - 48 >= 10 && v5 - 65 >= 26){
16          v1 = 0;
17          v2 = 0;
18        }
19      } else{
20        if(v1){
21          v2 = str + v4;
22          break;
23        }
24        v1 = str + v4;
25      }
26      v4++;
27      v3--;
28    }while(v3);
29    if(!v1 || !v2)
30      return result;
31    *a1 = v2 - v1 + 1;
32    result = v1;
33    return result;
34  }
```

A.4.3 Dream^{++}

Listing A.12 shows the decompiled code produced by DREAM^{++} for the fourth task. For this task, the following questions were asked:

1. What are the data types of the parameters?
2. What is the purpose of this function?
3. What is stored in the return value (variable result)?
4. What are the values that the arguments str and a1 points to after executing the function with the input string str = `"The quick brown %FOX% jumps over the lazy dog"`?
5. What are the values that the arguments str and a1 points to after executing the function with the input string str = `"The %QU1CK% brown %FOX% jumps %OVER% the lazy dog"`?

6. What are the values that the arguments str and a1 points to after executing the function with the input string str = `"My %P@SSWORD% is %INFECTED%"`?

Listing A.12: Decompiled code generated by DREAM++ for Task 4

```
int sub_404E14(const char * str, void * a1){
  int result = 0;
  if(str == 0)
    return result;
  size_t len = strlen(str);
  if(len < 3)
    return result;
  int v1 = 0;
  int v2 = 0;
  int index = 0;
  do{
    if(str[index] != '%'){
      char v3 = str[index];
      if((v3 < 48 || v3 >= 58) && (v3 < 65 || v3 >= 91)){
        v1 = 0;
        v2 = 0;
      }
    } else{
      if(v1 != 0){
        v2 = str + index;
        break;
      }
      v1 = str + index;
    }
    index++;
    len--;
  }while(len != 0);
  if(v1 == 0 || v2 == 0)
    return result;
  *a1 = v2 - v1 + 1;
  result = v1;
  return result;
}
```

A.5 Task 5

A.5.1 Hex-Rays

Listing A.13 shows the decompiled code produced by Hex-Rays for the fifth task. For this task, the following questions were asked:

1. What is the purpose of this function?
2. How many download attempts are possible?
3. What kind of files are being downloaded?
4. When will the CreateFileW function be called? (What marks a successful download)
5. What outcome results in which return code (variable v12)?

Listing A.13: Decompiled code generated by Hex-Rays for Task 5

```
1  int __userpurge sub_7FF92573@<eax>(int ebx0@<ebx>, int a1)
2  {
3    int v2; // ebp@0
4    int v3; // edi@1
5    int v4; // ebx@13
6    int v5; // eax@15
7    int v7; // [sp+8h] [bp-60h]@15
8    int v8; // [sp+4Ch] [bp-1Ch]@15
9    int v9; // [sp+50h] [bp-18h]@17
10   int v10; // [sp+5Ch] [bp-Ch]@15
11   int v11; // [sp+60h] [bp-8h]@1
12   int v12; // [sp+64h] [bp-4h]@1
13
14   v11 = 5;
15   v12 = 0x40;
16   v3 = requestUpdate(ebx0, a1, 0, 0, 1);
17   if ( v3 == -1 )
18     return v12;
19   while ( 1 )
20   {
21     if ( !v3 )
22       goto LABEL_9;
23     if ( *v3 == 'KP' )
24       break;
25     if ( *v3 == 'ZM' )
26       goto LABEL_13;
27     FreeHeap(v3);
28     v12 = 0x41;
29 LABEL_9:
30     --v11;
31     if ( v11 )
32     {
33       Sleep(3000);
34       v12 = 0x40;
35       v3 = requestUpdate(ebx0, a1, 0, 0, 1);
36       if ( v3 != -1 )
37         continue;
38     }
```

```
39      return v12;
40    }
41    v12 = 0x42;
42    ebx0 = unzipDownload(v2, v3);
43    FreeHeap(v3);
44    if ( !ebx0 )
45      goto LABEL_9;
46    v12 = 0x43;
47    if ( *ebx0 != 'ZM' )
48    {
49      FreeHeap(ebx0);
50      goto LABEL_9;
51    }
52    v3 = ebx0;
53  LABEL_13:
54    SetFileAttributesW(lpFileName, 128);
55    DeleteFileW(lpFileName);
56    v4 = CreateFileW(lpFileName, 0x40000000, 0, 0, 2, 36, 0);
57    if ( v4 == -1 )
58    {
59      v12 = 0x45;
60    }
61    else
62    {
63      v5 = ReallocHeap(v3);
64      WriteFile(v4, v3, v5, &v10, 0);
65      CloseHandle(v4);
66      memset(&v7, 0x44u);
67      v7 = 0x44;
68      if ( CreateProcessW(0, lpFileName, 0, 0, 0, 0, 0, 0, &v7, &v8) )
69      {
70        v12 = 0;
71        CloseHandle(v9);
72        CloseHandle(v8);
73      }
74      else
75      {
76        v12 = 0x46;
77      }
78    }
79    FreeHeap(v3);
80    return v12;
81  }
```

This code in this task calls other functions in the malware sample and thus they were explained as follows:

`requestUpdate(buf, url, a1, a2, a3)`. This function tries to download a file from the Internet and has the following interface:

- **Parameters**
  - buf: a pointer to the buffer that receives the downloaded file.
  - url: a pointer to a string value that contains the URL to download.
  - a1, a2, a3: irrelevant for this task.
- **Return value**: If the download succeeds, the function returns a pointer to buffer that contains the downloaded data. If there is no Internet connectivity, it returns -1. If download fails it returns 0.

`unzipDownload(a1, buf)`. This function decompresses a compressed buffer and has the following interface.

- **Parameters**
  - a1: irrelevant for this task.
  - buf: a pointer to the buffer that contains the compressed data. The function decompresses this buffer in place. That is, the original compressed data will be replaced by the decompressed data.
- **Return value**: If decompression succeeds, the function returns a pointer to the buffer containing the decompressed data (same as buf). Otherwise, it returns 0.

A.5.2 Dream

Listing A.14 shows the decompiled code produced by DREAM for the fifth task. For this task, the following questions were asked:

1. What is the purpose of this function?
2. How many download attempts are possible?
3. What kind of files are being downloaded?
4. When will the CreateFileW function be called? (What marks a successful download)
5. What outcome results in which return code (variable result)?

Listing A.14: Decompiled code generated by DREAM for Task 5

```
1  int sub_7FF92573(void * a1, int a2){
2    int v1 = 5;
3    int result = 64;
4    void * v2 = requestUpdate(a1, a2, 0, 0, 1);
5    LPCVOID lpBuffer = v2;
6    if(v2 == -1)
7      return result;
8    while(1){
9      bool cond1 = !lpBuffer;
10     bool cond2 = false;
11     void * v4;
12     short v3;
13     if(!cond1){
14       v3 = *lpBuffer;
15       if(v3 == 0x4b50){
16         result = 66;
17         v4 = unzipDownload(lpBuffer);
18         a1 = v4;
19         FreeHeap(lpBuffer);
20         if(v4){
21           result = 67;
22           cond2 = *v4 == 0x5a4d;
23           if(cond2)
24             lpBuffer = v4;
25           else
26             FreeHeap(v4);
27         }
28       } else if(v3 != 0x5a4d){
29         FreeHeap(lpBuffer);
30         result = 65;
31       }
32     }
33     if(!cond1
34         && (v3 != 0x4b50 || cond2)
35         && (v3 != 0x4b50 || v4)
36         && (v3 == 0x5a4d || v3 == 0x4b50)){
37       SetFileAttributesW(lpFileName, 128);
38       DeleteFileW(lpFileName);
39       HANDLE hFile = CreateFileW(lpFileName, 0x40000000, 0, 0, 2, 36, 0);
40       if(hFile != -1){
41         DWORD NumberOfBytesWritten;
42         WriteFile(hFile, lpBuffer, ReallocHeap(lpBuffer), &NumberOfBytesWritten, 0);
43         STARTUPINFOW StartupInfo;
44         HANDLE ProcessInformation;
45         CloseHandle(hFile);
46         memset(&StartupInfo, 68);
47         BOOL v5 = CreateProcessW(0, lpFileName, 0, 0, 0, 0, 0, 0, &StartupInfo, &
               ProcessInformation);
48         if(v5){
```

```
49          result = 0;
50          HANDLE hObject;
51          CloseHandle(hObject);
52          CloseHandle(ProcessInformation);
53        } else
54          result = 70;
55      } else
56        result = 69;
57      FreeHeap(lpBuffer);
58      return result;
59    } else{
60      v1--;
61      if(!v1)
62        return result;
63      Sleep(3000);
64      result = 64;
65      void * v6 = requestUpdate(a1, a2, 0, 0, 1);
66      lpBuffer = v6;
67      if(v6 == -1)
68        break;
69    }
70  }
71  return result;
72 }
```

The same explanation for the called internal functions was provided as in Section A.5.1.

A.5.3 Dream^{++}

Listing A.15 shows the decompiled code produced by Dream^{++} for the fifth task. For this task, the following questions were asked:

1. What is the purpose of this function?
2. How many download attempts are possible?
3. What kind of files are being downloaded?
4. When will the CreateFileW function be called? (What marks a successful download)
5. What outcome results in which return code (variable result)?

Listing A.15: Decompiled code generated by Dream^{++} for Task 5

```
1 const int ZIP_FILE_SIGNATURE = 0x4b50;
2 const int EXE_FILE_MZ_HEADER = 0x5a4d;
3
```

```
int sub_7FF92573(void * a1, int a2){
  int counter1 = 5;
  int result = 64;
  LPCVOID lpBuffer = requestUpdate(a1, a2, 0, 0, 1);
  if(lpBuffer != -1){
    bool cond2 = false;
    short v2;
    bool cond1 = false;
    void * v1;
    do{
      cond1 = lpBuffer == 0;
      if(!cond1){
        v2 = *lpBuffer;
        if(v2 != ZIP_FILE_SIGNATURE){
          if(v2 == EXE_FILE_MZ_HEADER)
            break;
          FreeHeap(lpBuffer);
          result = 65;
        } else{
          result = 66;
          a1 = unzipDownload(lpBuffer);
          FreeHeap(lpBuffer);
          if(a1 != 0){
            result = 67;
            cond2 = *a1 == EXE_FILE_MZ_HEADER;
            if(cond2){
              lpBuffer = a1;
              break;
            }
            FreeHeap(a1);
          }
        }
      }
      counter1--;
      if(counter1 != 0){
        Sleep(3000);
        result = 64;
        v1 = requestUpdate(a1, a2, 0, 0, 1);
        lpBuffer = v1;
      }
    }while(v1 != -1 && counter1 != 0);
    if(!cond1
        && (v2 != ZIP_FILE_SIGNATURE || cond2)
        && (v2 != ZIP_FILE_SIGNATURE || a1 != 0)
        && (v2 == EXE_FILE_MZ_HEADER || v2 == ZIP_FILE_SIGNATURE)){
      SetFileAttributesW(lpFileName, 128);
      DeleteFileW(lpFileName);
      HANDLE hFile = CreateFileW(lpFileName, 0x40000000, 0, 0, 2, 36, 0);
      if(hFile != -1){
        DWORD NumberOfBytesWritten;
        WriteFile(hFile, lpBuffer, ReallocHeap(lpBuffer), &NumberOfBytesWritten, 0);
        STARTUPINFOW StartupInfo;
```

```
56          HANDLE ProcessInformation;
57          CloseHandle(hFile);
58          memset(&StartupInfo, 68);
59          BOOL v3 = CreateProcessW(0, lpFileName, 0, 0, 0, 0, 0, 0, &StartupInfo, &
                ProcessInformation);
60          if(v3 != 0){
61            result = 0;
62            HANDLE hObject;
63            CloseHandle(hObject);
64            CloseHandle(ProcessInformation);
65          } else
66            result = 70;
67        } else
68          result = 69;
69        FreeHeap(lpBuffer);
70      }
71    }
72    return result;
73  }
```

The same explanation for the called internal functions was provided as in Section A.5.1.

A.6 Task 6

A.6.1 Hex-Rays

Listing A.16 shows the decompiled code produced by Hex-Rays for the sisth task. For this task, the following questions were asked:

1. What happens in lines 68-75 (dwSeed is a global variable of type int)?
2. Asume that the pointer &v25 at line 108 points to the string `".eu;1l;U*m"` (&v25 = `".eu;1l;U*m"`). What happens in lines 108-111?
3. How many times is the loop in lines 118-131 executed?
4. What happens in this loop?
5. What is stored in the return value (variable result)?
6. What is the purpose of this function?

Listing A.16: Decompiled code generated by Hex-Rays for Task 6

```
void *__cdecl sub_10006390()
{
  int v0; // edx@1
  char v1; // al@1
  char *v2; // ecx@1
  unsigned int v3; // esi@3
  char *v4; // edi@3
  char v5; // dl@4
  signed int v6; // eax@5
  signed int v7; // ecx@5
  signed int v8; // ecx@7
  char *v9; // eax@11
  void *result; // eax@12
  const char *v11; // esi@13
  char *v12; // eax@13
  __int32 v13; // eax@14
  int v14; // esi@15
  unsigned int v15; // ecx@15
  int v16; // edx@16
  char *v17; // edi@18
  bool v18; // zf@18
  unsigned int v19; // edx@18
  char v20; // dl@21
  LPVOID v21; // eax@24
  void *v22; // esi@24
  char v23; // [sp+0h] [bp-338h]@1
  char v24; // [sp+1h] [bp-337h]@1
  char v25; // [sp+104h] [bp-234h]@1
  char v26; // [sp+105h] [bp-233h]@1
  char v27; // [sp+208h] [bp-130h]@1
  char v28; // [sp+209h] [bp-12Fh]@1
  __int16 v29; // [sp+308h] [bp-30h]@1
  int v30; // [sp+30Ch] [bp-2Ch]@1
  int v31; // [sp+310h] [bp-28h]@1
  int v32; // [sp+314h] [bp-24h]@1
  int v33; // [sp+318h] [bp-20h]@1
  int v34; // [sp+31Ch] [bp-1Ch]@1
  char v35; // [sp+320h] [bp-18h]@1
  __int32 v36; // [sp+324h] [bp-14h]@14
  int v37; // [sp+328h] [bp-10h]@1
  __int16 v38; // [sp+32Ch] [bp-Ch]@1
  char v39; // [sp+32Eh] [bp-Ah]@1
  int i; // [sp+330h] [bp-8h]@1
  char v41; // [sp+334h] [bp-4h]@4
  char v42; // [sp+335h] [bp-3h]@4
  char v43; // [sp+336h] [bp-2h]@1

  v30 = *"qwrtpsdfghjklzxcvbnm";
  v32 = *"ghjklzxcvbnm";
  v33 = *"lzxcvbnm";
  v31 = *"psdfghjklzxcvbnm";
  v35 = aQwrtpsdfghjklz[20];
```

```
53  v37 = *"eyuioa";
54  v34 = *"vbnm";
55  v38 = *"oa";
56  v39 = aEyuioa[6];
57  v23 = 0;
58  memset(&v24, 0, 0x103u);
59  v25 = 0;
60  memset(&v26, 0, 0x103u);
61  v27 = 0;
62  memset(&v28, 0, 0xFFu);
63  v0 = dwSeed;
64  v1 = ’1’;
65  v29 = ’\0’;
66  v43 = ’\0’;
67  i = 0x45AE94B2;
68  v2 = "1670cf215403c56d8859a0636ffc74";
69  do
70  {
71    ++v2;
72    v0 += v1;
73    v1 = *v2;
74  }
75  while ( *v2 );
76  dwSeed = v0;
77  v3 = 0;
78  v4 = &v25;
79  do
80  {
81    v5 = a1670cf215403c5[v3 + 1];
82    v41 = a1670cf215403c5[v3];
83    v42 = v5;
84    *v4 = strtol(&v41, 0, 16);
85    v3 += 2;
86    ++v4;
87  }
88  while ( v3 < 0x1E );
89  v6 = 0;
90  v29 = 1;
91  v7 = 0;
92  do
93  {
94    *(&v27 + v7) = v7;
95    ++v7;
96  }
97  while ( v7 < 256 );
98  v8 = 0;
99  do
100 {
101   *(&v27 + v8) ^= *(&i + v6++);
102   if ( v6 >= 4 )
103     v6 = 0;
104   ++v8;
```

```
105     }
106     while ( v8 < 256 );
107     decrypt(&v27, 15, &v25, &v25);
108     v9 = strstr(&v25, ";"); //&v25 = ".eu;11;U*m";
109     if ( v9
110       && (*v9 = 0, v11 = v9 + 1, (v12 = strstr(v9 + 1, ";")) != 0)
111       && (*v12 = 0, v13 = strtol(v11, 0, 10), (v36 = v13) != 0) )
112     {
113       v14 = 0;
114       v15 = 3;
115       if ( v13 > 0 )
116       {
117         v16 = 1 - &v23;
118         for ( i = 1 - &v23; ; v16 = i )
119         {
120           v17 = &v23 + v14;
121           v19 = (&v23 + v14 + v16) & 0x80000001;
122           v18 = v19 == 0;
123           if ( (v19 & 0x80000000) != 0 )
124             v18 = ((v19 - 1) | 0xFFFFFFFE) == -1;
125           v20 = v18 ? *(&v37 + dwSeed / v15 % 6) : *(&v30 + dwSeed / v15 % 0x14);
126           ++v14;
127           v15 += 2;
128           *v17 = v20;
129           if ( v14 >= v36 )
130             break;
131         }
132       }
133       v21 = HeapAlloc(hHeap, 8u, 0x110u);
134       v22 = v21;
135       if ( v21 )
136       {
137         memset(v21, 0, 0x110u);
138         memset(v22, 0, 0x104u);
139         snprintf(v22, 0x104u, "%s%s", &v23, &v25);
140       }
141       result = v22;
142     }
143     else
144     {
145       result = 0;
146     }
147     return result;
148   }
```

A.6.2 Dream

Listing A.17 shows the decompiled code produced by DREAM for the sixth task. For this task, the following questions were asked:

1. What happens in lines 16-21 (dwSeed is a global variable of type int)?
2. Asume that the pointer &str1 at line 48 points to the string `".eu;11;U*m"` (&str1 = `".eu;11;U*m"`). What happens in lines 48-59?
3. How many times is the loop in lines 63-79 executed?
4. What happens in this loop?
5. What is stored in the return value (variable result)?
6. What is the purpose of this function?

Listing A.17: Decompiled code generated by DREAM for Task 6

```
1  LPVOID sub_10006390(){
2    int v1 = *"qwrtpsdfghjklzxcvbnm";
3    int v2 = *"eyuioa";
4    int v3 = 0;
5    int v4;
6    memset(&v4, 0, 259);
7    int str1 = 0;
8    int v5;
9    memset(&v5, 0, 259);
10   int v6 = 0;
11   int v7;
12   memset(&v7, 0, 255);
13   int v8 = dwSeed;
14   int v9 = 0x45ae94b2;
15   char v10 = 49;
16   char * v11 = "1670cf215403c56d8859a0636ffc74";
17   do{
18     v11++;
19     v8 += v10;
20     v10 = *v11;
21   }while(v10);
22   dwSeed = v8;
23   int v12 = 0;
24   char * v13 = &str1;
25   do{
26     char v14;
27     int str = v14;
28     v14 = *(v12 + (&a1670cf215403c5));
29     *v13 = strtol(&str, 0, 16);
30     v12 += 2;
31     v13++;
32   }while(v12 < 30);
33   int v15 = 0;
34   int v16 = 0;
```

```
35    do{
36      *(v16 + (&v6)) = v16;
37      v16++;
38    }while(v16 < 256);
39    int v17 = 0;
40    do{
41      *(v17 + (&v6)) ^= *(v15 + (&v9));
42      v15++;
43      if(v15 >= 4)
44        v15 = 0;
45      v17++;
46    }while(v17 < 256);
47    decrypt(&str1, &str1);
48    char * retStr = strstr(&str1, ";"); //&str1 = ".eu;11;U*m";
49    if(!retStr)
50      return 0;
51    *retStr = 0;
52    const char * str2 = retStr + 1;
53    char * retStr1 = strstr(str2, ";");
54    if(!retStr1)
55      return 0;
56    *retStr1 = 0;
57    long int num = strtol(str2, 0, 10);
58    if(!num)
59      return 0;
60    int v18 = 0;
61    int v19 = 3;
62    if(num > 0)
63      do{
64        char * v20 = v18 + (&v3);
65        int v21 = v18 + 1;
66        int v22 = v21;
67        int v23 = v21 & 0x80000001L;
68        bool v24 = !v23;
69        if(v23 < 0)
70          v24 = !((v23 - 1) | 0xfffffffeL) + 1;
71        char v25;
72        if(!v24)
73          v25 = *(((dwSeed / v19) % 20) + (&v1));
74        else
75          v25 = *(((dwSeed / v19) % 6) + (&v2));
76        v18++;
77        v19 += 2;
78        *v20 = v25;
79      }while(v18 < num);
80    LPVOID result = HeapAlloc(hHeap, 8, 272);
81    if(result){
82      memset(result, 0, 272);
83      memset(result, 0, 260);
84      _snprintf(result, 260, "%s%s", &v3, &str1);
85    }
86    return result;
```

```
87 }
```

A.6.3 Dream^{++}

Listing A.18 shows the decompiled code produced by DREAM^{++} for the sixth task. For this task, the following questions were asked:

1. What happens in lines 19-25 (dwSeed is a global variable of type int)?
2. Asume that the pointer str1 at line 44 points to the string `".eu;11;U*m"` (str1 = `".eu;11;U*m"`). What happens in lines 44-55?
3. How many times is the loop in lines 57-61 executed?
4. What happens in this loop?
5. What is stored in the return value (variable result)?
6. What is the purpose of this function?

Listing A.18: Decompiled code generated by DREAM for Task 6

```
1  LPVOID sub_10006390(){
2    char * v1 = "qwrtpsdfghjklzxcvbnm";
3    char * v2 = "eyuioa";
4    char * v3;
5    *v3 = 0;
6    char * str1;
7    int v4;
8    char * v6;
9    memset(&v4, 0, 259);
10   *str1 = 0;
11   char * v9;
12   int v5;
13   memset(&v5, 0, 259);
14   *v6 = 0;
15   int v7;
16   memset(&v7, 0, 255);
17   int v8 = dwSeed;
18   *v9 = 0x45ae94b2;
19   char counter1 = 49;
20   char * v10 = "1670cf215403c56d8859a0636ffc74";
21   while(counter1 != 0){
22     v10++;
23     v8 += counter1;
24     counter1 = *v10;
```

```
25    }
26    dwSeed = v8;
27    char * v11 = str1;
28    for(int i = 0; i < 30; i += 2){
29      char v12 = a1670cf215403c5[i];
30      char * str = v12;
31      *v11 = strtol(&str, 0, 16);
32      v11++;
33    }
34    int index = 0;
35    for(int i = 0; i < 256; i++)
36      v6[i] = i;
37    for(int i = 0; i < 256; i++){
38      v6[i] ^= v9[index];
39      index++;
40      if(index >= 4)
41        index = 0;
42    }
43    decrypt(str1, str1);
44    char * retStr = strstr(str1, ";"); //str1 = ".eu;11;U*m";
45    if(retStr == 0)
46      return 0;
47    *retStr = 0;
48    const char * str2 = retStr + 1;
49    char * retStr1 = strstr(str2, ";");
50    if(retStr1 == 0)
51      return 0;
52    *retStr1 = 0;
53    long int num = strtol(str2, 0, 10);
54    if(num == 0)
55      return 0;
56    int v13 = 3;
57    for(int i = 0; i < num; i++){
58      char v14 = i % 2 == 0 ? v1[(dwSeed / v13) % 20] : v2[(dwSeed / v13) % 6];
59      v13 += 2;
60      v3[i] = v14;
61    }
62    LPVOID result = HeapAlloc(hHeap, 8, 272);
63    if(result != 0){
64      memset(result, 0, 272);
65      memset(result, 0, 260);
66      _snprintf(result, 260, "%s%s", v3, str1);
67    }
68    return result;
69  }
```

List of Figures

1.1 Compilation vs Decompilation 3

2.1 Overview of the DREAM^{++} decompiler. In the remainder of this thesis, we refer by DREAM to the version of the decompiler consisting of the first four steps (highlighted in green). The name DREAM^{++} refers to the complete decompiler (highlighted in blue). 10

2.2 SSA Form. 13

2.3 Handling global variables for SSA back translation. 14

2.4 Expression Propagation. 17

2.5 A trivial ϕ function 18

3.1 Exemplary code sample 25

3.2 Abstract Syntax Tree 25

3.3 Control Flow Graph 26

3.4 Example of structural analysis. 26

3.5 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). 29

3.6 Decompiled code generated by DREAM (left) and by Hex-Rays (right). The arrows represent the jumps realized by `goto` statements. 30

3.7 $S_G(d_1, n_9)$ of the running example 32
3.8 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). 36
3.9 Loop structuring rules. The input to the rules is a loop node n_ℓ. 40
3.10 Example of loop type inference of region R_1. 41
3.11 Decompiled code generated by Hex-Rays. 42
3.12 Decompiled code generated by DREAM. 42
3.13 Aliasing example 43
3.14 Transforming abnormal entries: multi-entry loops (left) are transformed into semantically equivalent single-entry loops (right). Tags c_n represent the logical predicates $i = n$. 45
3.15 Transforming abnormal exits: loops with multiple successors (left) are transformed into semantically equivalent single-successor loops (right). 46
3.16 The five basic structures which cause unstructured flow diagrams. 47
3.17 Structured form of overlapping loops shown in Figure 3.16d. In this case, the condition that causes control flow to exit the loop through edge (c_1, n_1) is $c_s = c_1$. 48

4.1 Excerpt from the decompiled code generated by Hex-Rays of the domain generation algorithm of the Simda malware family. This example shows the main loop where the domain names are generated. At a high level, letters are picked at random from two arrays. Choosing the array from which to copy a letter is based on whether the loop counter is even or odd. 66
4.2 Decompiled code generated by DREAM for the same sample as Figure 4.1. 67
4.3 Decompiled code generated by DREAM^{++} for the same sample in Figure 4.1. . . 68
4.4 Congruence Analysis 69
4.5 Excerpt from the decompiled code from a Stuxnet sample. The code checks the version of the Windows operating system. 73
4.6 Code representations. 77
4.7 Sample search patterns 78

4.8 Sample transformation rule 78
4.9 Excerpt from the code of the Cridex malware family showing the code inlining technique. 81

5.1 Counterbalancing the order of decompiler and difficulty levels. Nodes in each horizontal sequence represent the tasks performed by one participant. Letters denote the used decompiler for the task and colors represent task difficulty level: medium (black) or hard (red). 97
5.2 Aggregated participant agreement with the statements related to usability perception (6 questions) and trust in correctness (2 questions). 105
5.3 Participant agreement with the statements from Table 5.2. The positive statements are marked in green and the negative statements are marked in gray. . . . 106
5.4 Boxplot for Rating 107

List of Tables

3.1 AST nodes that represent high-level control constructs 27

3.2 Correctness results. 52

3.3 Structuredness and compactness results. For the `coreutils` 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. 53

4.1 Logic-based predicates for the Dream IR. Each predicate has an *id* to uniquely represent the corresponding statement or expression. The second argument of each code fact is the parent id *pid* that represent the id of containing AST node. For a statement or expression e, we denote by $\#e$ the id of e. 76

5.1 Test User Study. The third column denotes the result of performing the task: ● task is completely solved, ◐ = task is partially solved, and ○ = task is not solved. Tasks are ordered according to difficulty level as shown by the pre-study. 95

5.2 Questions after each task. 98

5.3 Aggregated Experiment Results 102

5.4 Detailed study results for all tasks for the students group. 103