REcompile: A Decompilation Framework for Static Analysis of Binaries

Khaled Yakdan
University of Bonn
Institute of Computer Science 4, Germany
yakdan@cs.uni-bonn.de

Sebastian Eschweiler, Elmar Gerhards-Padilla
Fraunhofer FKIE, Germany
sebastian.eschweiler@fkie.fraunhofer.de
elmar.gerhards-padilla@fkie.fraunhofer.de

Abstract

Reverse engineering of binary code is an essential step for malware analysis. However, it is a tedious and time-consuming task. Decompilation facilitates this process by transforming machine code into a high-level representation that is more concise and easier to understand. This paper describes REcompile, an efficient and extensible decompilation framework. REcompile uses the static single assignment form (SSA) as its intermediate representation and performs three main classes of analysis. Data flow analysis removes machine-specific details from code and transforms it into a concise high-level form. Type analysis finds variable types based on how those variables are used in code. Control flow analysis identifies high-level control structures such as conditionals, loops, and switch statements. These steps enable REcompile to produce well-readable decompiled code. The overall evaluation, using real programs and malware samples, shows that REcompile achieves a comparable and in many cases better performance than state-of-the-art decompilers.

1 Introduction

Malware analysis helps to acquire detailed information about malware functionality. This knowledge is essential for developing countermeasures to protect computer systems. A thorough analysis is performed using reverse engineering which is a tedious and time-consuming process. With the rapid increase in the number of malware samples and the level of sophistication, techniques to speed up this process are needed. Decompilation, or reverse compilation, offers a promising solution. It is a program transformation from a low-level to a high-level representation making it easier to understand. By removing machine-specific details and reducing the amount of code an analyst has to read, the decompiled code becomes a good representation for malware analysts; they would analyze the high-level decompiled code of the program instead its disassembly.

During the compilation process several types of information are lost, including variable names, types, and high level expressions. Control changes are expressed in terms of branch instructions. Arguments are passed to functions in registers or pushed on the stack depending on the used calling convention. Therefore, a decompilation framework for supporting analysts must solve a large spectrum of problems in order to generate easily readable high-level code. To this end, the paper makes the following contributions:

- The architecture of REcompile, an efficient and extensible decompilation framework is developed.
- The framework extends existing control flow analysis algorithms to correct some flaws and introduces optimization techniques to achieve better readability.
- The framework is evaluated using real malware samples and compared to a state-of-the-art decompiler.

The rest of the paper is structured as follows: in Section 2, the related work is presented. Section 3 illustrates the main design ideas and describes REcompile's structure. Following this, REcompile's intermediate representation (IR), namely the Static Single Assignment (SSA) form, is presented in Section 4. Section 5 describes the employed data flow analyses. Section 6 provides an overview of type analysis implemented by REcompile. Control flow analysis that constructs high-level control structures is presented in Section 7. In Section 8, the performance of REcompile is evaluated and compared to the de facto standard decompilation tool, namely the Hex-Rays decompiler [2]. Section 9 presents the conclusion and future work.

2 Related Work

There have been several research activities aimed at solving the problems related to decompilation and several approaches have been developed. Cristina Cifuentes' PhD thesis [7] provides several techniques to deal with decompilation of binary files. She uses data flow analysis algorithms

978-1-4799-2535-3/13/$31.00

to identify function parameters and return values, eliminate dead code, and recover high-level expressions. Graph structuring algorithms are used to identify standard control structures. These techniques are implemented in the *dcc* decompiler [8] which only handles small Intel 80286/DOS programs.

The Hex-Rays Decompiler is a proprietary decompiler plugin for the Interactive Disassembler Pro (IDA) [2]. It decompiles one function at a time and generates C-like pseudo code. An application programming interface (API) is provided to access its analysis and develop custom analysis methods. Hex-Rays is the de facto standard decompilation tool. However, it is closed-source and no access is possible to the analysis algorithms it applies.

The SSA form is an intermediate representation (IR) currently used in many compilers. The use of SSA for decompilation was proposed by M. van Emmerik in his PhD thesis [11]. His work demonstrates the advantages of the SSA form for several data flow components of decompilers, such as expression propagation, identifying parameters and return values of functions, and eliminating dead code. His approach was tested using the Boomerang open-source decompiler [1] which supports several architectures and binary formats. The REC decompiler [6] also supports several architectures and binary file formats and handles complex control constructs.

Ďurfina et al. designed a retargetable decompiler in [17]. The decompiler is based on the architecture description language ISAC and the LLVM compiler system. First, machine code is translated into LLVM IR. Then, several optimization and analysis passes are applied before transforming the optimized LLVM IR into a Python-like code. Although this is an interesting idea, the decompiler is evaluated using only simple functions and is not compared to other decompilers.

3 System Design

REcompile is designed to meet the challenges of the increasing growth and level of sophistication of malware. Therefore, it is built to fulfill the following goals:

- **Readability** aims at generating easily readable and concise decompiled output. Machine-specific details should be replaced by equivalent high-level representations. Changes in the control flow should be expressed in terms of high-level control structures.
- **Extensibility** facilitates modifying the system and adding new functionality which is essential in order to easily adapt the decompiler to cope with the new techniques used in future malware.
- **Modularity** assists in the development process by dividing the system into independent components with well-defined input, output, and function.

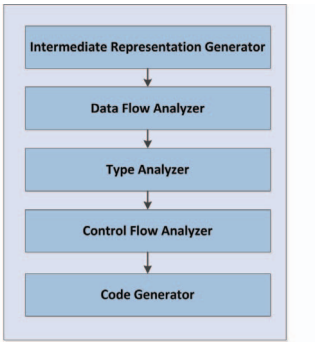


Figure 1: REcompile's architecture.

3.1 Overview of the System

In order to satisfy the extensibility and modularity requirements, an object-oriented design of REcompile's IR and core analysis algorithms is used. The IR's semantics are defined using class hierarchies that represent the IR instructions and the expressions used by each instruction. Figure 1 illustrates the main components of REcompile

- **Intermediate Representation Generator** transforms the input program into REcompile's IR that guarantees independence between analysis phases and the underlying architecture and binary format of input programs.
- **Data Flow Analyzer** performs code optimizations to reconstruct high-level expressions and conditions. It also removes dead code and identifies the parameters and return values of functions.
- **Type Analyzer** recognizes the types of variables based on how they are used in the program.
- **Control Flow Analyzer** reconstructs high-level language control structures such as loops and conditionals. This is accomplished by graph structuring algorithms that determine the underlying control structures of the control flow graph (CFG).
- **Code Generator** generates the decompiled code after all analysis phases have been performed.

REcompile is implemented as a plugin for IDA. It relies on IDA for initial disassembly, and hence the correctness of decompiled code depends on the correctness of IDA disassembly. Due to the modular layout, REcompile can be easily adapted to use other disassemblers. This only requires rewriting the IR generation phase. The term *input code* will be used in this paper to refer to disassembly code that REcompile accepts as input. Also, the term *decompiled code* refers to the high-level code generated by the decompiler.

4 Intermediate Representation

This section introduces REcompile's IR, namely the static single assignment (SSA) form. It is chosen due to the various advantages it offers to the data flow and type analysis steps.

4.1 Static Single Assignment

The SSA form is an IR in which each variable has only one definition in the program text [5]. The property of unique definitions allows for efficient implementation of data flow analyses. Transforming a program into SSA is done by adding subscripts to its variables with each new definition. ϕ functions are used to merge the different definitions of a given variable that reach join points of the CFG as shown in Figure 2.

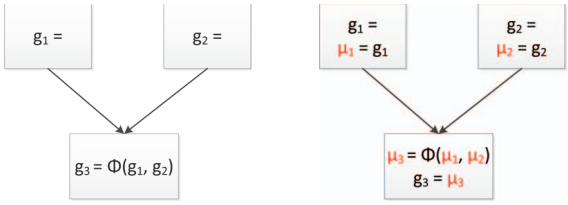
4.2 Transforming code into SSA

REcompile uses the SSA generation algorithm proposed by Cytron et al [10]. The algorithm efficiently constructs the SSA form based on a graph property called the *dominance frontiers* [5] and computes the minimal SSA form in terms of inserted ϕ functions. Moreover, REcompile constructs the following data structures.

- **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.

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. REcompile uses Sreedhar's algorithm [16] since it produces fewer copies in general [15]. 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.



(a) Before. (b) After.

Figure 2: Handling global variables.

Recompile solves this problem by breaking the live ranges of interfering global variables participating in a ϕ-function. Figure 2 shows an example of this case. If the live ranges of global variables g_1, g_2 and g_3 shown in Figure 2a interfere, copy instructions using local variables μ_1, μ_2 and μ_3 are inserted as illustrated in Figure 2b. 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.

5 Data Flow Analysis

In this section, we describe and discuss the second analysis phase, as depicted in Figure 1. Here, REcompile performs several data flow analyses to restore high-level statements corresponding to the input code.

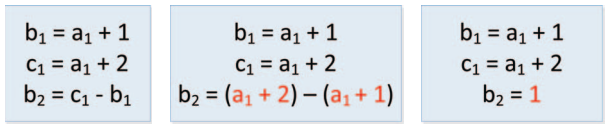
5.1 Expression Propagation

Machine code instructions can only represent simple expressions directly. Moreover, instruction sets impose restrictions on the number and type of operands. Therefore, compilers break high-level expressions into a sequence of simpler sub-expressions that can be represented by machine instructions. Expression propagation reverses this process by propagating variable definitions into the instructions using them. Figure 3a 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 3b.

This propagation may result in superfluously complex expressions. After propagation, REcompile performs a mathematical simplification phase in order to transform expressions into equivalent but simpler forms. This phase is analogous to that of common compilers, hence, it is not allowed too much space in the paper. But its effect is illustrated in Figure 3c where the third instruction is simplified.

5.2 Dead Code Elimination

A variable is *dead* if it is defined by a given instruction but not used afterwards. If the defining instruction only



(a) Sample code. (b) Code after propagation. (c) Code after mathematical simplification.

Figure 3: Expression propagation.

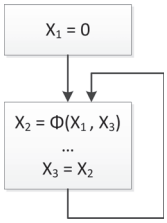


Figure 4: A trivial ϕ-function.

defines the dead variable, it can be safely removed. Dead code is common after expression propagation as illustrated in Figure 3c 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.

$$v \text{ is dead} \iff \textbf{usesMap}\,[v.name]\,[v.subscript] = \emptyset \quad (1)$$

Certain types of variables cannot be removed even if they satisfy statement 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, REcompile does not eliminate global variables. Combining expression propagation and dead code elimination enables REcompile to overcome obfuscation techniques that insert junk code and semantic NOPs.

5.2.1 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 1. This is particularly relevant for variables participating in ϕ functions. Figure 4 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 *a*) a ϕ function of variables in ϕ_t; or *b*) a copy assignment of the form $a_i = a_j$ defining a variable contained in ϕ_t . 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.

5.3 High-Level Condition Reconstruction

At machine code level, the notion of condition is represented using flags. Each flag represents a given status information about the result of executing the most recent instruction. High-Level Condition Reconstruction deduces the corresponding high-level conditions from the flags tested by branch instructions. Each flag is represented in REcompile's IR as independent variable. The setting of flags is represented by specific macros that are explicitly inserted right after each instruction affecting them. These macros summarize the information regarding which flags are defined and the variables based on which the flags are set. Treating flags this way allows to reconstruct high-level conditions without any assumptions regarding the positions of the defining and using instructions in code. For example, a subtraction instruction `sub` $\texttt{op}_1, \texttt{op}_2$ affects, among others, the zero flag (ZF). This is represented in the IR as a new statement:

$$(ZF_n, CF_n, ..., SF_n) = \texttt{SUBFLAGS}\,(op_1, op_2, result)$$

If the zero flag is tested by a `jz` instruction (jump if zero), the instruction is represented in the IR by:

$$\texttt{If}\,(ZF_n)\texttt{,BranchType:Equal,goto node}_m$$

where `BranchType` represents conditional jump type (here `jz`). The two statements summarize the information required to deduce the high-level condition: branch type, tested flags, and variables based on which the flags are set.

5.4 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. REcompile constructs function parameters based on the following equation

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

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.

6 Type Analysis

Type analysis addresses the problem of assigning types to variables. Certain machine instructions take operands of fixed and known types. These instructions include:

- *string instructions* that deal with pointers. This set includes `movs, lods, stos, cmps`.
- *integer instructions* that deal with integer values. This set includes `mul, div` , etc.
- *floating-point instructions* that operate on floating-point numbers. This set includes `fadd, fdiv` , etc.
- *API function calls* where the types of parameters and return values can be easily acquired.

REcompile uses these instructions as reliable starting points for performing type unification. That is, it deduces 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 the same type. 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 that allows to efficiently get, for each variable, the defining instruction and the list of using instructions.

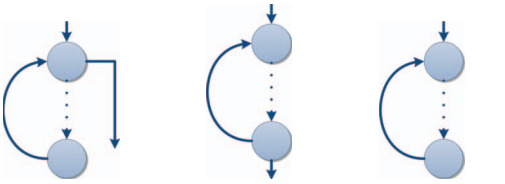
7 Control Flow Analysis

In this section, we describe the techniques used by REcompile to recover high-level control structures such as loops and conditionals. This is the fourth phase as shown in Figure 1. REcompile applies graph structuring algorithms on the CFG to find the subgraphs corresponding to high-level control structures. It extends the algorithms proposed by Cifuentes in [7]. Although these algorithms achieve a good performance, they fail to handle certain cases because the assumptions involved do not hold in the general case.

7.1 Exit Basic Blocks Splitting

An exit basic block in a function is a block at which the execution terminates. It usually contains code to restore saved registers and return to the calling function. The code generation step walks the CFG and prints its blocks' code. If a block is already printed, a goto statement is generated. Therefore, blocks with a small number of incoming edges are desirable. Often, many exit paths lead to a common exit block which may lead to generating goto statements.

In order to minimize the number of goto statements, REcompile duplicates exit blocks with more than two incoming edges so that each preceding block has one exit block with one incoming edge. Although this technique slightly increases the size of decompiled code, it decreases the number of goto statements which severely decrease code readability. Duplicating large exit blocks radically increases the volume of decompiled code. Therefore, only exit blocks whose instructions number is below a threshold are splitted.



(a) Pre-tested Loop. (b) Post-tested Loop. (c) Endless Loop.

Figure 5: Loops.

7.2 Structuring Loops

REcompile applies interval theory and derived sequence of graphs $G^1 \dots G^n$ introduced by F.Allen and J.Cocke [9, 3, 4]. Given a node h, An interval $I(h)$ is the maximal, single-entry subgraph in which h is the only entry node and in which all closed paths (cycles) contain h. The interval structure satisfies two important conditions: *a*) there exists at most one loop per interval; and *b*) a nesting order is provided by the derived sequence of graphs [12]. This provides a precise loop definition in terms of graph representation. Figure 5 shows subgraphs of loops structured by REcompile: pre-tested (while) loops, post-tested (do-while) loops, and endless loops.

7.2.1 Loop Detection

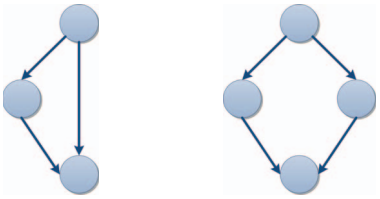
A loop in a given interval $I(h)$ is detected by the existence of a back edge from a node $l \in I(h)$, called the latching node, to the header node h.

7.2.2 Loop Nodes

Cifuentes stated in [7] that nodes in a loop defined by the back edge (l, h) in interval $I(h)$ are defined as follows:

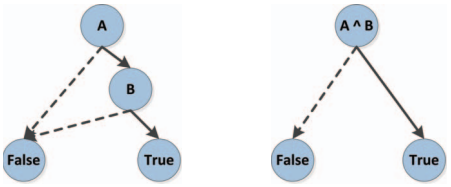
$$\forall n \in \text{Loop}(l,h) : n \in (I(h) \cap \text{Candidates}) \qquad (2)$$

Candidates is the set of nodes $\{n\}$ satisfying $\texttt{rPostorder}(h) \leq \texttt{rPostorder}(n) \leq \texttt{rPostorder}(l)$ where $\texttt{rPostorder}(n)$ is the reverse postorder of node n. However, the reverse postorder is not unique; several different reverse postorderings of graph nodes may exist. This depends on the order in which the depth-first search algorithm traverses graph nodes. This flaw may result in erroneously marking some nodes as loop nodes.



(a) If-then. (b) If-then-else.

Figure 6: Conditionals.



(a) Original subgraph. (b) Equivalent subgraph.

Figure 7: Compound conditionals.

REcompile corrects this flaw by adding a new condition to cope with all possible orderings. This condition states that if node n belongs to Loop (l, h), there must exists a path from n to the loop header h that contains the latching node l. Combining this with statement 2 results in correct detection of loop nodes.

After identifying loop nodes, REcompile detects loop type and exit nodes at which the execution continues after the loop is terminated. Those exits are analyzed to check if they correspond *break* or *continue* statements.

7.3 Structuring Conditionals

REcompile recognizes both if-then and if-then-else conditionals whose corresponding subgraphs are shown in Figure 6. Both conditionals are characterized by a two-way header node and a common follow node that is immediately dominated by the header node and has two predecessors. The domination property may not hold in the case of nested conditionals. In this case, the header of the inner subgraph is added to a set of unresolved conditionals and finding the follow node is delayed until the outer one is analyzed. This is done by traversing the nodes according to their postorder since node n is last visited by DFS before node m iff $\texttt{postorder}(n) < \texttt{postorder}(m)$. n-way nodes corresponding to switch statements are handled similarly.

High-level languages perform short-circuit evaluation for compound conditionals where several conditions are combined using logical operators. This technique results in an unstructured graph which includes several basic blocks each of which is dedicated to test an individual condition. Figure 7a shows an example consisting of a compound condition $A \wedge B$. Dashed arrows represent false paths and solid arrows represent true paths. Handling each 2-way block independently results in a goto statement. This is due to the fact that each conditional would have either an abnormal exit or entry. Therefore, REcompile searches for compound conditionals in the CFG and translates them into equivalent graphs as illustrated in Figure 7b. This graph has one block in which the compound condition is tested and can then be structured by REcompile without goto statements.

8 Evaluation

The evaluation of REcompile is performed using two tests. The first test includes samples whose source code is available, which allows to compare the decompiled code to the original source code. The second test compares the performance of REcompile with the Hex-Rays decompiler which is the de facto standard decompiler. The used Hex-Rays version is v1.8.0.130306. Both decompilers are used to decompile a set of 19824 functions chosen from real malware samples including Waledac, SpyEye, Conficker, and Flame. The set also includes Mozilla Firefox open-source web browser. Malware samples constitute 87.5% of this set. We first manually unpacked packed samples so that IDA can disassemble the actual malware code.

In this evaluation, the focus is on readability. Although this requirement is subjective and can be rated differently by different analysts, it can be argued that a good decompiler should generate fewer lines of code compared to the corresponding machine code. It should also represent the control flow using high-level control structures. Therefore, the following performance metrics are chosen:

1. **Functional equivalence** between the input code and its decompiled code means that they follow the same behavior and produce the same results when executed using the same set of parameters.
2. **Structuring efficiency** represents the ability of expressing the control flow in terms of high-level control structures. It is measured by the inverse of the number of generated goto statements in the decompiled code.
3. **Reduction ratio** represents the decrement of the volume that occurred to the input code as a result of the analysis phases. It is defined as follows:

$$\textit{Reduction ratio} = \left(1 - \frac{\#\textit{lines in decompiled code}}{\#\textit{lines in input code}}\right) * 100$$

8.1 Functional Equivalence

To test functional equivalence, we created 10 test programs including functions of varying complexity ranging

from functions with one basic block to functions with several nested control structures. These programs were compiled and then provided to REcompile as input. We then manually compared the source and decompiled code. In all tested cases, the decompiled code and source code are functionally equivalent. Data flow analysis algorithms removes all machine-specific details from the input code. All high-level conditions are restored and replaced the low-level conditions expressed in terms of processor flags. These flags are removed from the decompiled code.

8.2 Structuring Efficiency

For the majority of decompiled functions, no goto statements are generated. Table 1 presents the percentage of the functions for which no goto statements are generated. Here, REcompile performs slightly better than Hex-Rays.

Tool	% of cases without goto statements
REcompile	91.2%
Hex-Rays	89.3%

Table 1: Goto statements.

Figure 8 presents a comparison between REcompile and Hex-Rays with regard to the number of lines and goto statements in the decompiled code. As illustrated in Figure 8(a) Both decompilers generate an equal number of goto statements in 86.7% of the cases. This is dominated by the functions for which no goto statements are generated as shown in Table 1. REcompile generates less goto statements in 8% of the cases. Hex-Rays performs better for 5.3% of the tested functions. This shows that REcompile achieves a slightly better structuring efficiency than Hex-Rays. Here, the exit basic block splitting step plays an important role in avoiding unnecessary goto statements.

8.3 Reduction Ratio

It is difficult to optimize for the three chosen metrics simultaneously. This can be seen when testing REcompile's performance against the reduction ratio metric. Achieving a smaller number of goto statements has the impact of reducing the achieved reduction ratio. Interestingly, there are functions with negative reduction ratio, i.e., the decompiled code is larger that the input code. This is particularly relevant for small functions where only few lines have a huge impact on the reduction ratio.

Figure 8(b) shows that for 41.2% of the cases REcompile generates smaller decompiled code, while Hex-Rays performs better in 46% of the cases. The penalty of having a smaller reduction ratio while being able to produce less goto statements is justified by the fact that such statements radically decrease code readability.

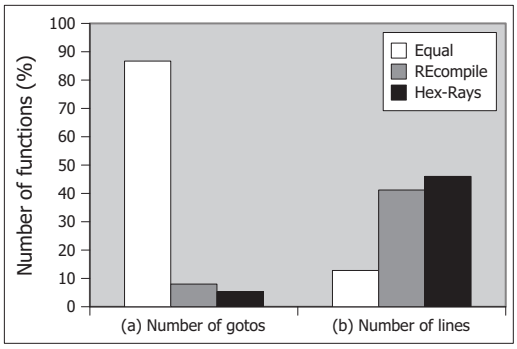


Figure 8: Comparison between REcompile and Hex-Rays according to the number of lines and gotos in decompiled code. The bars represent the percentage of functions where 1) equal performance is achieved; 2) REcompile achieves a better performance; and 3) Hex-rays achieves a better performance.

8.4 Comparison with Hex-Rays

Figure 9 shows a function, with a relative virtual address (RVA) of 0x1598, from the Waledac malware decompiled using REcompile and Hex-Rays. The sample's md5 hash is `19af4bd0ef0579d632c6096b0d3de4bd`. Both outputs are functionally equivalent but REcompile's decompiled code is more compact and easier to read. Hex-Rays interprets the loop as an endless loop including a break and a goto statement. REcompile, on the other hand, interprets the loop as a post-tested loop. The exit basic block splitting optimization eliminates the need for goto and break statements. Also, Hex-Rays' output has two unnecessary assignments: `v1=lpMem` and `v4=v3`. Such instructions are removed by REcompile's data flow analysis. Simplifying ternary operators are not yet supported by REcompile, and hence the first *return* instruction in Figure 9b is not simplified.

Readability is a subjective matter, and therefore it should be evaluated using user experiments. However, the focus of this paper is on presenting the design of REcompile. A deeper evaluation including a questionnaire to get reversers' feedback is planned for future work.

9 Conclusion and Future Work

Reverse engineering of binary code is essential for malware analysis. This paper presents REcompile, a decompilation framework designed to fulfill the requirements of readability, extensibility, and modularity. It solves a large set of problems related to decompilation. REcompile uses the Static Single Assignment (SSA) form as its intermediate representation and performs several analyses. Data flow analysis replaces low-level details of the code by equivalent high-level representations. Type analysis finds variable types. Control flow analysis recovers high-level control

```
bool __cdecl sub_401598(int a1)
{
  LPVOID v1; // ebx@1
  int v2; // eax@3
  int v3; // eax@5
  int v4; // esi@5
  int v5; // eax@6
  int v7; // [sp+Ch] [bp-4h]@1

  v7 = 0;
  v1 = lpMem;
  while ( 1 )
  {
    v3 = dword_4031C0(a1, dword_4031D4, 102400, 0);
    v4 = v3;
    if ( !v3 )
      return sub_401448() == 0;
    if ( v3 < 0 )
    {
LABEL_7:
      LOBYTE(v5) = 0;
      return v5;
    }
    sub_401115((char *)v1 + v7, dword_4031D4, v3);
    v7 += v4;
    v2 = sub_401448();
    if ( !v2 )
      break;
    if ( v2 != 1 )
      goto LABEL_7;
  }
  LOBYTE(v5) = 1;
  return v5;
}
```

(a) Hex-Rays.

```
sub_401598(arg_0) {
    int  a;
    int  eax_7;
    int  eax_2;
    int  eax_15;

    a = 0;
    do{
        eax_2 = dword_4031C0(arg_0, dword_4031D4, 102400, 0);
        if((eax_2 == 0)) {
            eax_15 = sub_401448();
            return (((-eax_15) != 0) ? 0 : 1);
        }
        if((eax_2 < 0)) {
            return 0;
        }
        sub_401115((a + lpMem), dword_4031D4, eax_2);
        a = (a + eax_2);
        eax_7 = sub_401448();
        if((eax_7 == 0)) {
            return 1;
        }
    } while(((eax_7 + -1) == 0));
    return 0;
}
```

(b) REcompile.

Figure 9: Decompilation example.

structures. REcompile can also reconstruct complex nested control structures and compound conditionals.

The evaluation of REcompile shows that it produces easily readable decompiled code. It generates smaller number of goto statements than Hex-Rays and achieves a comparable performance regarding the size of decompiled code. One limitation in the current implementation is that REcompile relies on IDA for disassembling machine code. In the presence of indirect branches as well as anti-disassembly and obfuscation techniques, IDA may produce incorrect or incomplete disassembly which leads to incorrect decompiled code by REcompile. This issue is an important step for future work. Several interesting approaches in this area already exist and can help. They include the work of Kinder et al. [13] for accurate disassembly and the method of Lakhotia et al. [14] to detect obfuscated calls in binary code. Future work also includes inter-procedural analysis and deeper alias analysis.

References

[1] Boomerang decompiler. Retrieved from `http://boomerang.sourceforge.net`.

[2] Hex-rays decompiler. Retrieved from `https://www.hex-rays.com/products/decompiler/`.

[3] F. E. Allen. Control flow analysis. In *Proceedings of a symposium on Compiler optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[4] F. E. Allen and J. Cocke. Graph theoretic constructs for program control flow analysis. Technical Report 3923, IBM, Thomas J. Watson Research Center, Yorktown Heights, New York, July 1972.

[5] J. P. Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

[6] G. Caprino. Reverse engineering compiler. Retrieved from `http://www.backerstreet.com/rec/rec.htm`.

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

[8] C. Cifuentes. The dcc decompiler. GPL licensed software. Retrieved from `http://www.itee.uq.edu.au/~cristina/dcc.html`, 1996.

[9] J. Cocke. Global common subexpression elimination. In *Proceedings of a symposium on Compiler optimization*, pages 20–24, New York, NY, USA, 1970. ACM.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS*, 13:451–490, 1991.

[11] M. J. V. Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, The University of Queensland, 2007.

[12] B. C. Housel, III. *A study of decompiling machine languages into high-level machine independent languages*. PhD thesis, Purdue University, 1973.

[13] J. Kinder and H. Veith. Jakstab: A static analysis platform for binaries. In *Proceedings of the 20th international conference on Computer Aided Verification*, pages 423–427, 2008.

[14] A. Lakhotia, E. U. Kumar, and M. Venable. A method for detecting obfuscated calls in malicious binaries. *IEEE Transactions on Software Engineering*, 31:955–968, 2005.

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

[16] V. C. Sreedhar, R. D. ching Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *In Static Analysis Symposium, Italy*, pages 194–210. Springer Verlag, 1999.

[17] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna. Design of a retargetable decompiler for a static platform-independent malware analysis. *International Journal of Security and Its Applications*, 5:91–106, 2011.