Language Model Directed Decompilation

Kai H. Michaelis

May 23, 2018

Contents

Abstract xi

Prologue xiii

Scooping The Loop Snooper

A proof that the Halting Problem is undecidable xiii

1. Introduction 1

1.1. Motivation 1

1.2. Related Work 1

1.3. Outline 2

2. Decompilation 3

2.1. Accuracy 3

2.2. Disassembly And Lifting 4

2.3. Variable Recovery 6

2.4. Abstract Syntax Tree Construction 9

2.5. Type Inference 12

2.6. Summary 14

3. A Language Model For C 17

3.1. Neural Networks 18

3.2. Recurrent Neural Networks 19

3.3. Readability Scoring 20

3.4. Summary 20

4. Language Model Directed Decompilation 21

4.1. Training the Language Model 21

4.2. An Iterated Refinement Algorithm 21

4.2.1. Deriving the Initial Abstract Syntax Tree 22

4.2.2. Available Expressions, Dead Code Elimination and Constraint Propagation 25

4.2.3. Transformation Selection 27

4.2.4. Measuring 27

4.2.5. Metaheuristics 29

4.3. Transformation Rules For Control-Flow 30

4.4. Transformation Rules For Expressions 32

4.5. Identifier Selection 35

4.6. Summary 36

5. Results 37

5.1. Test Set 37

5.2. Overview 37
5.2.1. Running Time 37
5.2.2. Memory Consumption 38
5.2.3. Completeness of Disassembly 39
5.2.4. Compactness of Decompilation 40
5.3. Language Model Performance 41
5.4. Transformation Rules 43
5.5. Decompilation Quality 44
5.5.1. Our Implementation 44
5.5.2. RetDec 47
5.5.3. Hexrays 48
5.6. Identifier Selection 48
5.7. Summary 49

6. Conclusion 51
6.1. Further Directions 52
6.1.1. Algebraic Subtyping for Machine Code 52
6.1.2. Abstract Syntax Tree Based Machine Learning 52
6.1.3. Faster Condition Based Refinement 53

Appendices 59

A. Complete Figures 61

List of Figures

2.1. Arbitrary control transfer on AMD64 using `ret`. 4
2.2. Example of Gulianovs Method. The basic block on the left has a call to `f` with unknown effect on `ESP`. The effect can be determined by observing that $in_B = in_C$ and $out_C = out_B$. 8
2.3. Typical patterns used in structural analysis. 9
2.4. Example of common subexpression elimination. Left before running the algorithm, right after. 10

3.1. Markov chain for words in *Look at you, hacker*. We ignored white space and punctation. 17
3.2. Schematic of a LSTM cell. The input gate receives the cells output from the previous character, combines it with the current network input and sends it to the reset gate. The reset gate combined the new value with the old and sends it to the output gate. The output gate combines the value with the next character. Adapted from [30] 20

4.1. Lack of inlining causes excessively large `if`-condition before `return` statements. It also makes it hard to recognize that control flow never falls through the second `if`-statement. 22
4.2. Results of exit node inlining. 23
4.3. Example of a dead code elimination. 25
4.4. Definition of the AST sum type. The *Comparison* and *Block* are used for the pre-AST. After the control flow constructs have been inferred, they are removed. 26
4.5. Example of a available expressions analysis and subsequent forward expression propagation. 27
4.6. Example of condition propagation removing superfluous if-statements. 28
4.7. Language model-directed refinement algorithm. 31
4.8. Rule for flattening early exits. 31
4.9. Rule for simplifying `if`-statements. 31
4.10. Rule for swapping `if` branches. 32
4.11. Rule for creating `switch`-statements. 32
4.12. Rule for expression simplifications. 33
4.13. Rule for integer logic equivalences. 33
4.14. Rule for reversing strength reduction optimizations. 35

5.1. Time spent decompiling a subset of `coreutils` in seconds. 38
5.2. Peak memory consumption while decompiling a subset of `coreutils` in bytes. 39
5.3. Number of functions not found by the decompilers. 40
5.4. Average lines of code in the decompiled function. 40
5.5. Final readability scores of all three implementations. Lower is better. 41

5.6. Flame graph of the critical function in out decompiler. Bars higher up give a finer grained profiling of the bar below. All refinement loop iterations are folded. 42
5.7. Popularity of AST transformation rules. 44
5.8. Redundant redundancy in an if-condition. A typical decompilation failure. 45
5.9. Example of an overly complex `if`-condition. 46
5.10. RREIL code for `mov RBX, RDI`. 46
5.11. Example of an over approximated read/write set. Both `RSI` and `ESI` are listed despite them being the same register. 46
5.12. Example of a stray Φ node. 46
5.13. Example of a correctly handled non-returning function. 47
5.15. Example of a stray `goto`-statement left by RetDec. 48
5.17. Sample of identifiers generated by the language model. 49

6.1. Different AST node sequences resulting from different AST traversal orders. 53
6.2. Using dominators to decide path condition subsumption. 54

A.1. Time spent decompiling a subset of `coreutils` in seconds. Complete figure. 62
A.2. Number of functions not found by the decompilers. Complete figure. 63
A.3. Average lines of code in the decompiled function. Complete figure. . 64
A.4. Final readability scores of all three implementations. Lower is better. Complete figure. 65

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Ich erkläre mich damit einverstanden, dass die digitale Version dieser Arbeit zwecks Plagiatsprüfung verwendet wird.

Official Declaration

Hereby I declare, that I have not submitted this thesis in this or similar form to any other examination at the Ruhr-Universität Bochum or any other Institution of University.

I officially ensure, that this paper has been written solely on my own. I herewith officially ensure, that I have not used any other sources but those stated by me. Any and every parts of the text which constitute quotes in original wording or in its essence have been explicitly referred by me by using official marking and proper quotation. This is also valid for used drafts, pictures and similar formats.

I also officially ensure, that the printed version as submitted by me fully confirms with my digital version. I agree that the digital version will be used to subject the paper to plagiarism examination.

Not this English translation, but only the official version in German is legally binding.

Datum/Date

Unterschrift/Signature

Acknowledgements

The author thanks, in no particular order: Cornelius Aschermann, Marcus Brinkmann and LABOR .e.V.

Abstract

This Master thesis develops a decompiler from AMD64 machine code to C that is supported by a recurrent neural network. We use conventional disassembly and static analysis algorithms to recover an initial abstract syntax tree (AST). This AST is then modified using transformations randomly sampled from a fixed set. After each transformation, the neural network is used to extract a readability score. If the transformation improved the score we keep it, otherwise the results are discarded and another transformation is chosen. We continue applying transformations until the readability score can no longer be improved The prototype implementation was evaluated against two existing decompilers.

Prologue

Before we dive into this 76 page thesis, lets loosen up with this little poem. We will refer to its moral later.

Scooping The Loop Snooper
A proof that the Halting Problem is undecidable

No general procedure for bug checks will do.
Now, I won't just assert that, I'll prove it to you.
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called P
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and P gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs.

If there will be no looping, then P prints out 'Good.'
That means work on this input will halt, as it should.
But if it detects an unstoppable loop,
then P reports 'Bad!' — which means you're in the soup.

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,
I could use it to set up a logical bind
that would shatter your reason and scramble your mind.

Here's the trick that I'll use — and it's simple to do.
I'll define a procedure, which I will call Q,
that will use P's predictions of halting success
to stir up a terrible logical mess.

For a specified program, say A, one supplies,
the first step of this program called Q I devise
is to find out from P what's the right thing to say
of the looping behavior of A run on A.

If P's answer is 'Bad!', Q will suddenly stop.
But otherwise, Q will go back to the top,

and start off again, looping endlessly back,
till the universe dies and turns frozen and black.

And this program called Q wouldn't stay on the shelf;
I would ask it to forecast its run on itself.
When it reads its own source code, just what will it do?
What's the looping behavior of Q run on Q?

If P warns of infinite loops, Q will quit;
yet P is supposed to speak truly of it!
And if Q's going to quit, then P should say 'Good.'
Which makes Q start to loop! (P denied that it would.)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

I've created a paradox, neat as can be —
and simply by using your putative P.
When you posited P you stepped into a snare;
Your assumption has led you right into my lair.

So where can this argument possibly go?
I don't have to tell you; I'm sure you must know.
A reductio: There cannot possibly be
a procedure that acts like the mythical P.

You can never find general mechanical means
for predicting the acts of computing machines;
it's something that cannot be done. So we users
must find our own bugs. Our computers are losers!

Geoffrey K. Pullum[1]

[1]Copyright ©2008, 2012 by Geoffrey K. Pullum. Permission is hereby granted to reproduce or distribute this work for non-commercial, educational purposes relating to the teaching of computer science, mathematics, or logic, provided this attribution is included.

1. Introduction

1.1. Motivation

"What does this program do?" is a question that arises across various disciplines of computer science. In the context of security we areas often interested how malware got into the system, how it spreads and how it communicates with its creator. Free and Open Source developers that seek to write a replacement for proprietary software need to understand how a closed file format or protocol works. Ignoring the legal ramifications of reverse engineering software, the technical challenges are still huge. Chromium, one of the most common applications on desktop computers as well as mobile phones is more than 100 MB in size. While advancements in automatic search and exploitation of software vulnerabilities have been made, reverse engineering of more complex behaviour like protocols and infection routines remains manual work.

Decompilation is a misnomer as it implies that the compilation step is reversed in the sense of recovering the original C code. This is impossible as compilation is a lossy process. We can resolve this dilemma by reformulating the problem. Instead of recovering the original code, we create a C program that has roughly the same runtime semantics as the original binary. We will then do our analysis on the C code. The assumption is that all conclusions we draw from that are also true for the binary. Despite all this we will use the terms decompiler and decompilation in this thesis.

Decompilers prove to be a significant step forward for manual reverse engineering [50]. While they are not able to perfectly recover the original code, they're still able to provide the analyst with a representation of the program that is easier to reason about. Additionally, C can be used as input into other static analysis tools [52], allowing them to analyse binary applications without implementing the complex disassembly routines themselves.

This thesis concentrates on decompiling binaries to C, but that does not mean that this is the only sensible target language. mean that this is the only sensible target language. In recent years the focus has moved towards analysis of C++ applications [42, 45]. Naturally, decompiling to C++ makes sense in this context [26]. Another interesting target language is LLVM [36]. LLVM bitcode can be recompiled to machine code allowing ports of binaries to different processor architectures. Lately, LLVM has become the go-to input language for static analysis tools [49, 14]. As with the static analysis tools for C a decompiler targeting LLVM would allow using these tools on binaries.

1.2. Related Work

The idea of decompilers is not a new one. Historically the first published decompiler was *dcc* [15]. After this followed a string of other takes on the problem like Boomerang [25], Phoenix [46], Dream(++) [51, 50], RetDec [35], SmartDec [26],

Snowman [4], *fcd* [2], *radeco* [44] and Hex-Rays [3] (in no particular order). All of these are either academic prototypes attempting to improve the state-of-the-art or commercial tools aimed at professional analysts.

Dream and Phoenix are particular important for this thesis as they are fairly new and also employ iterative improvement of an initial AST. What sets our approach apart is that we apply transformation rules only if they improve the AST. Previous tools have no such measure and thus can only use rules that improve readability regardless.

1.3. Outline

This thesis develops a decompiler for ELF [5] binaries for AMD64 [1] processors and tries to generate more readable code than existing projects by utilizing a statical model of manually written C code. In chapter 2, we will give an overview of the current state-of-the-art of decompiling binaries. We will employ most of the techniques mentioned there to generate a first Abstract Syntax Tree (AST) of a function. In chapter 3, we present to a language model we developed to measure the readability of abstract syntax trees. Chapter 4 presents the iterative refinement of the decompiled AST, directed by the language model. Lastly in chapter 5 we compare our results with those of commercial offerings as well as academic prototypes. We close with our conclusion in chapter 6. Chapters 3 and 4 present novel work.

2. Decompilation

In order to create a C program from assembly it makes sense to take a look at both to see how they differ, these are:

1. Lack of variables,
2. lack of types and
3. lack of structured control flow.

In this chapter we describe the process of disassembling a binary, recovering function boundaries and the stack frame layout. Then we will infer the types locally and generate an initial abstract syntax tree which will serve as a starting point for the iterative improvement algorithm described in Chapter 4.

2.1. Accuracy

Before any decompilation can happen we need to find and disassemble all code inside the binary. Deciding if a byte is code is equivalent to asking whether its address inside a binary is reachable. General reachability questions can be reduced to the Halting Problem. [1] This means complete and accurate disassembly is undecidable in general. Binaries regularly include indirect jumps, i.e. jumps with targets that are computed at run-time. Even when we ignore the reachability problem, an algorithm computing exact jump targets may not terminate as it involves executing the program. This again introduces the Halting Problem. Again we will solve this dilemma by lowering our standards and accepting approximate solutions. Approximations introduce the question whether we favour Soundness or Completeness. Soundness means all facts our algorithms derive are true. Completeness means we derive all facts (accepting that some may be false). Academic research often favours Soundness, i.e. deriving only facts we know for sure are true and ignoring everything else. While this is reasonable, it also serverly limits how useful our analysis is in real-world conditions. For example, AMD64 has instructions for inter procedural (meaning between functions) control flow. The `call` instruction calls into a function, the `ret` and `retn` instructions return from it. Sadly, the concrete semantics of these instructions are looser defined. The `call` instruction pushes the address of the following instruction onto the stack and jumps to the supplied function address. The `ret/retn` instructions read an address from the top of the stack, pop it and jump to this address. Assuming that `ret/retn` instructions always mark the end of the current function is not sound. Listing 2.1 shows how to use `ret` as dynamic jump instruction. In order to make sure we correctly mark a `ret/retn` instruction as function end, we need to ensure the top

[1] Deciding whether the address after a function call is reachable includes deciding whether the function returns.

of the stack is the same as it was at function entry. This involves tracking the value of the stack register throughout the function (and it is callees). This, again, introduces the Halting Problem.

```
push $DEADBEEF
ret
```

Figure 2.1.: Arbitrary control transfer on AMD64 using `ret`.

That and similar challenges prompt us to favour Completeness, forgoing Soundness, which is common practice in commercial tools like Hex-Rays that employ heuristics to find functions in binaries. This has led to users of disassemblers and decompilers not blindly trusting their results [12]. To mitigate failures of the algorithms, most commercial reverse engineering tools allow analysts to correct the disassembly. Another way to look at the Soundness issues is to see the decompiler as a function that is only defined on the output of a compiler.

2.2. Disassembly And Lifting

Aside from deciding where code lies, disassembly is a straight forward process. There exists a one-to-one mapping between assembly instructions and byte sequences. Depending on the CPU architecture instructions can be constant or vary in size. AMD64 is a variable-size instruction set with instruction lengths ranging from one to 15 bytes [1]. The disassemble function is given the address of the start of an instruction and will return its human readable name (also called *mnemonic*), its arguments and its length. The function will be called for each address where we expect to see code and that is not already covered by another instruction. We call the resulting list of mnemonic, argument and address assembly listing. Disassemblers can be categorized by how they decide which address to disassemble next.

The *Linear Sweep* disassembler simply start at the lowest code address and continue sequential towards the highest address, expecting all code to be in one continuous address range. Linear sweep disassembler require some degree of cooperation from the compiler. Data like jump tables or padding bytes that are interleaved with code will be treated as code. Worse, for variable-length instruction sets the data may be decoded as an instruction that partly overlaps with different code address (and thus an actual instruction) causing disassembly to fail completely. An advantage of linear sweep is their speed and accuracy. If the compiler does not put data into code sections, a linear sweep will disassemble all code despite indirect jumps.

The second category are the *Control Flow Aware* disassembler. These have additional knowledge about how control flow proceeds after each instruction. If an instruction branches, both the next code address and the branch target will be disassembled. Same with function call instructions and unconditional jumps. This technique does not require code to be in one continuous address range and will skip all data that may be between instructions. The downside is that it avoids the Halting Problem by only disassembling reachable addresses (s.a). In order to follow control flow the disassembler needs to resolve indirect jumps. While jump targets cannot be computed perfectly in general without executing the program, static analysis can be

used to approximate the set of possible values. A popular technique to do this is Abstract Interpretation [18]. Abstract Interpretation approximates run time behaviour by executing an abstraction of the program, replacing values with sets of values, following both paths at branch points and merging value sets at join points. Abstract Interpretation is parametrized with an *Abstract Domain* which defines the exact way value sets are represented as well as how they are merged. Abstract domains exists for either sound [33] or complete [18] analysis.

Disassembly only deals with the encoding of instructions, either as binary code or textual assembly listing, i.e. its syntax. For Abstract Interpretation as well as type inference and later decompilation stages we are more interested in what an instruction does, i.e. its semantics. In order get them we use the same technique as compilers do and introduce an *Immediate Language* (IL). The idea is to generate an IL snippet for each disassembled instructions that describes its run time semantics. The immediate language is designed to be easy to analyse by limiting the number of instructions. Analysis stages that work on the disassembly semantics only need to understand a few, simple IL instructions instead of the ~500 instructions of the AMD64 CPU architecture. This also enables reuse of the analysis code for other CPU architectures. The only part that needs to be changed is the disassembler. This immediate code is also useful for communicating analysis results from one stage to the other by rewriting parts of it. A dead code elimination pass could remove parts of the code. A stage that partly recovers the stack frame can replace stack slots with fresh variables and so on. Translating machine code to an immediate language is sometimes called lifting. The idea of using an immediate language was introduced by REIL [24] as part of the commercial tool BinNavi [7], later RREIL [47] built on this. Other applications of this concept are VEX and the LLVM language. VEX was initially part of the Valgrind [40] dynamic analysis tool suite and was later used for the binary analysis toolkit *angr* [48]. LLVM was designed for the compiler suite with the same name but is now used in various decompilers and static analysis projects [49, 14, 11].

After we derived an assembly listing from the binary static analysis starts. The first step is to determine function boundaries. This means recovering a mapping from code addresses to functions. Each function is a collection of code addresses with one special address called the entry point. Each code address belongs to one or more functions. As each function has a single entry point, we can use it to identify functions. To find function entries we just look at all `call` instructions. The function extends from the entry point to the `ret` instruction. All code addresses between are assigned to the function. Optimizing compilers can reuse code. This is why a single code address can be part of multiple functions. This is problematic for performance when we analyse each function independently, all shared code addresses are processed multiple times without yielding more information. The most common type of code reuse are leaf calls [2] and calls to functions that do not return. In both cases the compiler replaces the `call` instruction with an unconditional jump. In both cases runtime behaviour isn't effected by this change. A simple analysis pass that turns all unconditional jumps to another functions' entry into a call instruction can correct this problem.

With function boundaries determined we proceed to split the assembly code of a function into *basic blocks*. Each basic block is a sequence of instructions that do not

[2]Leaf calls are function calls that occur directly before returning from the function.

have any jumps or branches in between [3]. Additionally, we require that no jump into the basic block occur. This means that each instruction inside a basic block is executed if the block is jumped to. With the basic blocks in place we create a *Control Flow Graph*, a directed, potential cyclic graph, of all basic blocks with edges between blocks if a jump (also called *Control Transfer*) can happen.

Construction of the control flow graph concludes the disassembly process.

2.3. Variable Recovery

After disassembly finishes and all instructions have been lifted we are left with a *Control Flow Graph*. The next step is to recover the concept of variables. Most functions reserve space on the stack to temporarily save values and pass arguments to other functions. Values flow between memory cells of the stack and registers. Our goal is to remove all usage of the stack by introducing new variables in the immediate language code of the function. For our proposes we define the stack as all addresses that are accessed relative to the stack pointer register (`RSP` for AMD64). These are found using Abstract Interpretation. The abstract domain we used was introduced in [33] as *Bounded Address Tracking*. It represents register values as $R + n$, where R is a symbolic value, also called memory region and $n \in \mathbb{Z}_{2^{64}}$ a linear offset from the start of the region. A special region GLOBAL is used for absolute values. At entry the value of `RSP` is `RSP` $+ 0$. After the Abstract Interpretation finishes we make a pass through the function and collect all memory load and store operations with target addresses of the shape of `RSP` $+ n$. Each is replaced by a copy to or read from a fresh variable. All memory operations inside the GLOBAL region are replaced with global variables. Ideally this removes all memory operations.

The first use of Abstract Interpretation for variable recovery was by Balakrishnan and Reps in their commercial tool CodeSurfer/x86. Their *Value Set Analysis* [10] pioneered this approach. Their abstract domain used intervals instead of integers for describing offsets. While this is more accurate it also requires interval arithmetic that correctly models overflow and boolean logic operation. While these exist [32] they are hard to implement correctly and slower. Intervals improve accuracy if the stack contains arrays that are processed iteratively inside the function. In this case using a single integer as offset fails. If the stack is only used to spill values when the compiler runs out of registers, intervals have no advantage.

Variable recovery depends on the ability to correctly track the (symbolic) values of the stack across the function. This is only possible if we know the effect every instruction has on the stack register. This can be difficult for function calls. In the multitude of C calling conventions there are some, namely `stdcall` were a function removes more values from the stack than it pushes. If a call to an unknown function is encountered, the analysis cannot infer the new value of the stack register. A solution for this problem was presented by Gulianov and implemented in the commercial decompiler Hex-Rays. Gulianov computes a "stack effect" for each basic block of a function. The stack effect is the number of bytes the value stack register changes after the basic blocks concludes. If the basic block i contains a call to an unknown function, the stack effect is represented by a fresh variable a_i. Gulianovs algorithm

[3]Some exclude unconditional jumps in the definition of basic blocks and only require them free of intermittent branches.

assumes that the value of the stack register is equal before each return instruction of the function. From this and the control flow graph we can derive a linear programming problem. First we introduce two fresh variables out_i and in_i for each basic block i. For each basic block we derive the equation

$$in_i - out_i = \text{stack effect of } i.$$

For each control transfer from i to j we derive

$$out_i - in_j = 0,$$

for each exit block e.g. basic blocks that have no successors we derive

$$out_i = x$$

and for the entry basic block we derive

$$in_i = 0.$$

The objective function we want to be minimized is

$$\sum^{i} a_i + x.$$

The solution for this linear programming problem provides us with the missing stack effects. There is no reason why this solution is the right one, except for the observation that it works well in practice. Gulianov argues that selecting a solution that minimizes the used stack space makes sense as compilers tend to optimize code to use as few memory as possible. Figure 2.2 shows a simple example.

Summarizing Function

Recovering local variables also involves recovering function arguments – Arguments to the function we analyse as well as arguments passed to functions called inside. If we assume the source of the binary is a C program, the C ABI for the platform tells us how to identify arguments to exported functions. C compiler are freer in how to send argument to functions that are not exposed to other programs. When decompiling non-C/C++ binaries, we also do not know how arguments are passed. A common way to handle this situation is to take a hint from program analysis and use data flow analysis. Computing the upwards exposed variables of the entry node, i.e. the registers that are read before being written to, gives us the set of arguments to the function that are passed via registers. After recovering stack slots, we can also apply this analysis to arguments passed through the stack. For this the recovered stack slots of the calling function are copied to the called function. Data flow analysis cannot determine the order of the arguments in the functions' signature. The C ABI can be taken as a hint if the function obeys it, otherwise any ordering is applicable. Return values are more complicated. A straight forward way to get the set of return values is to compute the set of written registers. Sadly this would include the registers that are saved on the stack and recovered before function exit by `push`/`pop`. Instead, we want the list of registers that may have different values after the function returns compared to when it is called. As this analysis is about the value of variables (registers), this is not a data flow analysis. Here, Abstract Interpretation helps. All

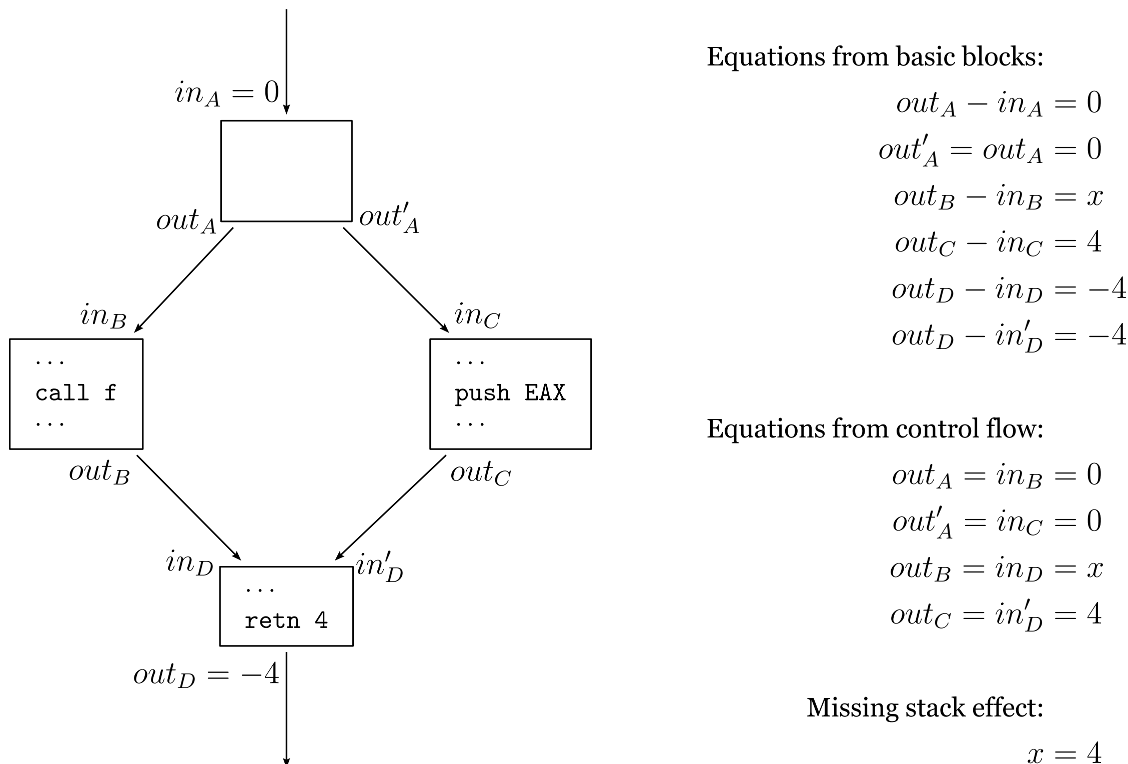


Figure 2.2.: Example of Gulianovs Method. The basic block on the left has a call to `f` with unknown effect on `ESP`. The effect can be determined by observing that $in_B = in_C$ and $out_C = out_B$.

registers are initialized with a symbolic "start" value. The final value of a register is the join of all its reaching definitions across all exit nodes. If the final value of a register is not "start", it contains a return value. The result of both analysis steps can be represented by read and a write-set. The read-set contains all function arguments, the write-set all return registers. Running this analysis in post-order on the call graph recovers the arguments for each known function. What remains are all functions that are not part of the binary: imported library functions and system calls. As long as we are able to identify the libraries used and acquire them read and write-sets can be computed separately and copied to the original binary. Same can be done for system calls by identifying the operating system. In case the library cannot be found the C ABI provides us with a default. All caller saved registers are assumed to be written. If Gulianovs method concludes a non-zero stack effect for the function call, the stack registers are also written.

A more exact method of handling function effects is the use of function summaries like they were developed Dahse et al. for their static analysis tool RIPS [21]. Function summaries allow for a more fine-grained approximation of functions. While a read/write-set can only tell us that a registers contents is trashed (i.e. written with an undefined value), function summaries could tell us it is set to either `1` or `0`. Also, function summaries can include memory operations. A function that uses the `RAX` register as a pointer to a four byte integer would include a load operation using RAX as address in its summary. Function summaries are strictly more powerful than read/write-sets. Every read/write-set can be turned into a function summary by inserting a copy operation from a register to itself for each read register and an

assignment with an undefined value to any written register into the summary. The read/write-set can be recovered by simple data flow analysis.

Ideally, we now converted all usage of the stack frame into fresh variables and identified function call arguments and their return values.

2.4. Abstract Syntax Tree Construction

For now, a function is represented by a control flow graph of immediate language code. We now want to convert this into an *Abstract Syntax Tree*, which is a tree of expressions and control flow constructs. There are two approaches to structuring the CFG as AST. Historically the first is structural analysis. Here a set of patterns are searched for in the control flow graph. If one matches, the matching nodes are replaced by a C control flow construct. A diamond is turned into an `if/else` statement, a back edge [4] and all nodes in between are replaced with a loop and so on. Figure 2.3 list these common patterns and their corresponding control flow constructs. The algorithm continues until no more matches are found or all edges in the control flow graph has been replaced. All remaining edges in the control flow graph are replaced with `goto` statements. This technique was used in the Phoenix decompiler [46]. The main advantages of structural analysis are its speed and simplicity. The downside is that in the presence of optimizing compilers simple pattern match often fails and the resulting AST contains a lot of `goto` statements. For example, it is common for loops to have multiple exits and entries and `return` statements are often merge by compilers.

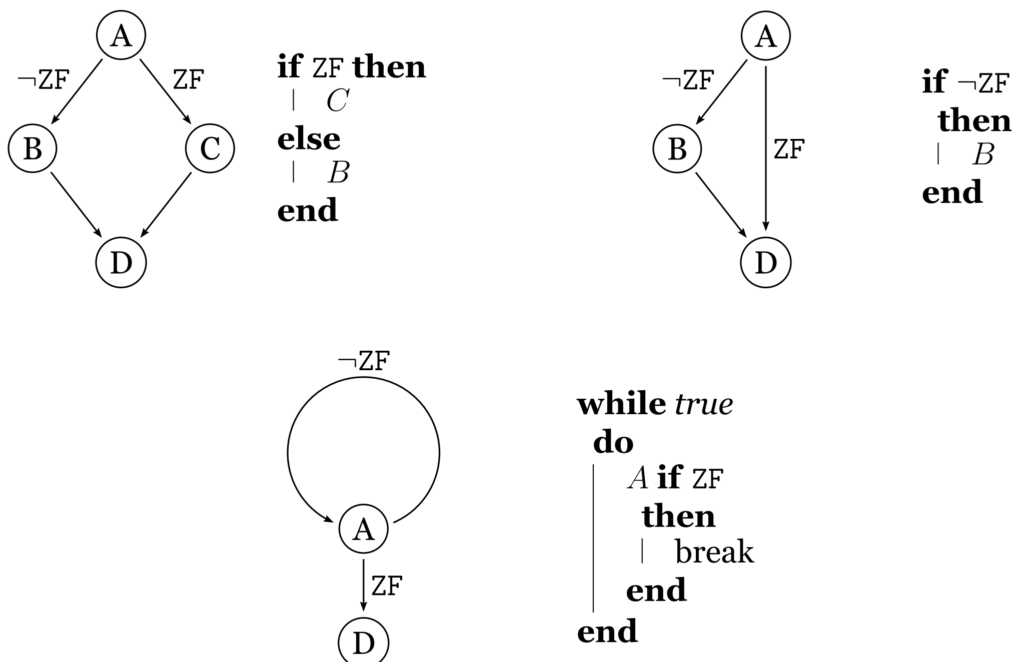


Figure 2.3.: Typical patterns used in structural analysis.

A different approach took the authors of the Dream [51] decompiler. Their algorithm starts by running a depth first search on the control flow graph starting at

[4] A back edge is an edge which source nodes position in depth first order is later than its target node.

the entry point. This identifies back edges that induce loops, dubbed cyclic regions. Loops with multiple entries or exits are transformed into one with a single entry and exit node. For a loop with multiple exits a fresh variable is introduced. Each exit node save one is replace with one that assigns a unique value to this variable and a `break` statement. After the remaining exit node code is inserted that checks the fresh variable and jumps to the original exit nodes successor. A similar method is used to get rid of multiple entry nodes. Now, the back edge is removed, replaced with a guarded `break` and nodes inside the loop are replaced with a `while(true) {…}` statement. After all cyclic regions have been replaced, the algorithm starts processing each single-entry-single-exit [5] (SESE) region of the control flow graph. SESE regions include the loop bodies created in the previous step. The idea is to structure the node based on their reachability condition. That is the conjunction of all branch conditions that need to be taken to reach that particular node from the SESE regions entry. This already is enough to turn a SESE region into an AST. Simply sort the nodes in reverse post-order and put an `if` statement around them. The statements condition is the reachability condition. Obviously this is not particularly readable. The algorithm then starts to group `if` statements by factoring out common conditions. See for example figure 2.4. Here the first condition is contained in the second. We can put the second condition into to first's body and remove the common factor. The third condition is the negation of the first. This means we can turn it into an `else` branch. Factoring common conditions proceeds in post-order over the control flow graph and tries to simplify sequences of `if` statements reachable from that node.

```
if(A) {
  z();
}

if(A && B) {
  x();
}

if(!A) {
  y();
}
```

```
if(A) {
  z();
  if(B) {
    x();
  }
} else {
  y();
}
```

Figure 2.4.: Example of common subexpression elimination. Left before running the algorithm, right after.

For now all loops in the reconstructed AST are endless `while` loops with one or more `breaks` guarded by `if` statements. To transform these into more "natural" looking loops Dream uses a pattern matching mechanism they call *Loop Structuring* rules. The authors provide six of these rules that describe how to transform certain AST patterns to make more readable loop constructs.

Our decompiler uses both, structural analysis and the condition based refinement

[5] A SESE region is exactly what the name suggests. A set of nodes in the control flow graph that are only reachable through a single entry. All paths leaving the region must go through a single exit node. There is a more exact (and complicated) definition involving the graph dominance relationship between nodes we will omit here.

implemented in Dream. The structural analysis runs first and replaces three control flow graph patterns (c.f. figure 2.3). First, it looks for nodes that have a single successor where the successor only has one predecessor and combines them to a single node. Second it looks for simple branches that either result in an `if`- or `if/else` statement. After the structural analysis stops making progress, our implementation of the Dream algorithm takes over.

Our disassembler implements the algorithm described in the paper without any substantial changes. What the authors omit in their paper is the exact mechanism they use to structure conditions especially deciding whether a condition subsumes another and how they "factor out" the common parts. This is why we describe our approach here in detail. We assume conditions are in *First Order* logic. This means we only consider expressions consisting of variables x_i or their negation $\neg x_i$. As connectives, we allow logical conjunction (AND) $\wedge$ and logical disjunction (OR) $\vee$. We now want to decide for first order logic expressions $A(\vec{x})$ and $B(\vec{x})$ whether A subsumes B, i.e. A implies B, written

$$A \implies B.$$

This logical implication can also be written as

$$\neg(A \wedge \neg B).$$

Deciding

$$\forall \vec{x}\ \neg(A(\vec{x}) \wedge \neg B(\vec{x}))$$

is equivalent to deciding

$$\nexists \vec{x}\ A(\vec{x}) \wedge \neg B(\vec{x}).$$

This is a *Boolean satisfiability problem* (SAT) which is known to be $\mathcal{NP}$-Hard. In practice this means the (asymptotically) fastest algorithm needs to enumerate all possible $\vec{x}$ and check if the formula evaluates to true. Factoring out common terms, written $A \setminus B$ is not a SAT problem as it involves transforming the formulae instead of proving satisfiability. In order to solve both problems without introducing dependencies to complex tools or libraries we convert formulae into *disjunctive normal form* (DNF). A formula is in DNF if it is a disjunction of conjunctions e.g. $\bigvee \bigwedge x_i$. One way to see the DNF is as an enumeration of all solution to the formula. To make a formula in DNF true one of its conjunctions must be true. For the conjunction to be true all non-negated variables must be set to true and all negated variables to false [6]. Additionally, the DNF expects that for each conjunction a variable can only occur once, either negated or non-negated. This means each formula can be represented as a set of sets of variables. A subsumes B if all conjunctions of B are subsets of conjunctions of A. Factoring out A with a single conjunction from B means computing the set difference of all conjunctions of B with that of A. This limits condition based structuring to simple conjunctions. In practice complex disjunctions are very rare as common terms.

After all control flow constructs have been inferred and the control flow graph collapsed to a single AST node we still need to turn the assembly code listing into C

[6] This implies that converting a general formula to DNF is $\mathcal{NP}$-Hard.

expressions. This is entirely done on the IL code, mnemonics are discarded at this point. First, IL operations are turned into C expressions, then an *Available Expressions* [17, p.490] analysis forward propagates computed expressions. Finally, we remove all dead code, this is necessary as the immediate language code implements all instruction side effects like setting flag registers. An AMD64 `test` instruction sets the flag registers `PF`, `ZF`, `SF` and `OF`. Not all of them will be used in subsequent branches. The IL code that computes the unneeded flags can be removed. To do this we proceed with a *Backwards Program Slice*. This removes all instructions that are not needed to compute values that are in our *Slicing Criteria*. Our slicing criteria contains all flags that are used in branching, all arguments to other functions, return values and addresses as well as values used in memory load and store operations. Intuitively these are all values that contribute to observable behaviour of the function we are decompiling.

After our combined structural and condition based AST recovery algorithm concludes and all assembly code was converted to C expressions we have an initial AST that will serve as a starting point for the language model directed refinement described in chapter 4.

2.5. Type Inference

The last concept missing from our generated C code are types. After AST recovery we have an untyped program, a set of types that C provides us with and a small set of typing constraints (i.e. if the variable `a` is dereferenced we know it must be some kind of pointer). What we now look for is an algorithm that "fills in the gaps". This problem is called *Type Inference* and has been studied since to 1960s in form of the simply typed lambda calculus [43].

An early and rather influential general solution to the type inference problem was developed by Hindley [28] and Damas-Milner [22, 39], dubbed algorithm W. Hindley-Milner poses type inference as unification problem. Their algorithm traverses the programs AST and generates equality constraints between concrete type (*type constants*) and variables. For example the expression $a + b$ generates the constraints

$$a \equiv \texttt{int}$$
$$b \equiv \texttt{int},$$

the expression $a = b$ generates the constraint $a \equiv b$ and $c = *a$ generates

$$a \equiv \text{Ptr}(\alpha)$$
$$c \equiv \alpha,$$

where α is a fresh variable and Ptr is a type constructor. The constraint set is then solved using unification. Here a substitution is maintained mapping variables to other variables or types. Constraints are processed in random order and each constraint updates the substitution or, if the constraint cannot be unified returns a special error value. Hindley-Milner is syntax-directed meaning typing constraints are generated based on the shape of the operations and in isolation. This means

that it cannot be directly applied to type systems with overloading where operations with multiple type signatures exist (e.g. the + operation could mean integer addition if both arguments are integers, floating-point addition if both are floats and string concatenation if applied to strings).

TIE [37] is a type inference algorithm for reverse engineering that is meant to be directly applied to machine code. It extends the unification based framework of Hindley-Milner with structural subtyping. For this TIE defines a subtype relation $\leq$ between types. $A \leq B$ means that in all contexts where B is applicable, A also is [7]. This relation induces a partial order on all types that form a *Lattice* [8]. TIE is syntax directed and models overloading using intersection types. An intersection type has the shape $A \wedge B \wedge C$, meaning its behaviour at runtime can be described by types A, B and C. TIE overloads the + operator, $c = a + b$ produces the constraint set

$$
\begin{aligned}
(a \leq \text{num32}, b \leq \text{num32}, \text{num32} \leq c) \vee \\
(a \leq \text{Ptr}(\alpha), b \leq \text{num32}, \alpha \leq c) \vee \\
(a \leq \text{num32}, b \leq \text{Ptr}(\alpha), \alpha \leq c).
\end{aligned}
$$

These constraints express that if both a and b are numbers, the result is a number but if either a or b is a pointer, the result will be a pointer. Conceptionally this the same as overloading the + operator with three definitions. Traditional Hindley-Milner type inference cannot handle overloading, nor subtyping. To solve the constraint set the TIE authors developed a greedy algorithm with backtracking that infers type intervals. A type interval consists of an upper and a lower type that represent the most general and the most specific type of expression respectively. Intersection types also cannot be inferred by Hindley-Milner, to rectify this TIE tries to solve the constraint system with one of the types in the intersection and backtracks if that fails. In that case it continues with the next until no more remain. In case the constraint system can be solved, all inferred type intervals are mapped to C types and displayed to the user. While Hindley-Milner has a runtime of $\mathcal{O}(n)$ in the size of the program, TIEs use of backtracking makes its runtime exponential in the worst case.

What is curious about TIE is their use of subtyping to infer C types. C has only extremely limited subtyping [9] why would we need a more powerful type system to infer C types? There are two reasons for this. First, C uses manifest typing exclusively. Every variable needs to have its type declared before use. A C compiler does only very limited type inference on expressions [10]. For example,

$$\texttt{void} * \texttt{memcpy}(\texttt{void} * \texttt{restrict s1}, \texttt{const void} * \texttt{restrict s2}, \texttt{size_t n});$$

does type erasure by the use of `void` pointer. This is not a problem in C as `s1` and `s2` have their types ascribed in the function. In the type inference case we first have to recover this information. A more exact function signature for `memcpy` is one where both pointer arguments are polymorphic i.e.

[7] In TIE all type constructors are covariant.

[8] That is one of the mathematical concepts called lattice. Do not confuse this with the discreet linear equation system that bears the same name.

[9] Longer integers are subtypes of smaller one and `void` pointers are a supertype of all other non-constant pointers.

[10] The type τ of `x ? a : b` is $\tau \leq \alpha \wedge \tau \leq \beta$, where α and β are the types of `a` and `b` respectively.

$$\forall\tau\ \mathrm{Ptr}(\tau)\ \texttt{memcpy}(\mathrm{Ptr}(\tau)\ \texttt{s1}, \mathrm{Ptr}(\tau)\ \texttt{s2}, \texttt{size_t}\ n).$$

Second, real-world C code make copious use of a special case of structural subtyping. Here two types are compatible if the types of their initial fields are equal, for example a function expecting a pointer to an integer also works with a pointer to a structure which first field is an integer. This means that in C, and assembly code, access through pointers must be modelled as a subtype relationship. A read from a pointer like

$$\alpha\ \texttt{a} = *(*\beta)\texttt{p};$$

induces the constraint $\alpha \leq \beta$ and a write operation to a pointer like

$$\beta\ *\texttt{p} = \texttt{a};$$

creates the constraint $\alpha \geq \beta$.

This and other complications of real-world C prompted the development of Retypd [41]. Here only subtype constraints exists that are solved by converting the constraint set into a Weighted Push-down System [31] (PDS) that is queried for every type in the type lattice.

Although their approach is more powerful in principle, implementing it from the automata-theoretic description in the paper proved far too difficult.

Thus, we will use a simple unification algorithm to infer C types that uses the subtype relation from TIE to infer pointers in a purely intra procedural manner.

2.6. Summary

In this chapter we gave a short overview on the different steps of translating binary code to a C program with similar runtime characteristics, also called decompilation. We started with disassembling machine code to an assembly listing by following the control flow. When encountering dynamic jumps, we used Abstract Interpretation to approximate the jump targets. Given the choice of complete or sound abstractions we chose completeness, hoping the human analyst would spot and correct incorrect conclusions. Similarly, we expect the code to use `call` and `ret` instructions to realize inter procedural control flow. Aside from listing mnemonics and their operands we also emit immediate language code that describes their semantics. Data flow analysis and another Abstract Interpretation pass is used to gain insights in how the stack frame is used in a function, again with the assumption that the binary was generated by a “sane” compiler. We accept the fact that we are only processing partial programs that do not include all code that is executed like dynamic libraries and operating system parts. We used unsound algorithms like Gulianovs method and hardcoded read/write-sets to approximate the behaviour of missing functions as good as possible. After we were able to recover a functions’ behaviour at runtime as well as its interaction with the rest of the program we convert it to C syntax. Constructing an C AST from the assembly control flow graph is a mix of graph pattern matching and reachability based refinement that is followed by more pattern matching. We accept a less than perfect solution as we are only interested in getting a starting point for later language model based refinement of the AST. As a last step we infer type for the

expressions in our AST. The practise of playing hard and loose with typing rules in real-world C programs requires us to use a more powerful type system for inference than one would expect. All steps except for disassembly is done for each function in isolation. While this limits the accuracy of our algorithms this also helps to parallelize the problem and limit the size of a single problem instance. The result is a set of C functions that now can be further manipulated.

3. A Language Model For C

We want to use a language model to guide decompilation. We will model the C programming language, but the techniques we use can be applied to any other language. The language model will tell us how similar to C a given text is. The similarity is expressed as a real number. This score will help the decompilation processes to decide whether applying a certain transformation improves readablity.

Language models can be categorized roughly into symbolic and probabilistic models. Symbolic models are based on formal languages and production rules derived from mathematical logic. These models are part of symbolic Artificial Intelligence (AI), that was predominant from the 1950s until the rise of Machine Learning in this century. Symbolic AI has been falling out of favour since computers became fast enough to simulate Neural Networks that turned out to be very effective at various pattern recognition tasks. We follow this trend and will also use a neural network. These are part of the probabilistic models that understand language as a sequence of tokens that have a certain probability distribution that depends on the previous tokens.

A simple probabilistic language model is a *Markov Chain*. A Markov Chain defines a transition system consisting of states and transitions between them. Each state is labelled with a token (there is exactly one state per token) and transitions with probabilities attached to them. This transition system defines the probability distribution for the next token, depending on the current one (c.f. figure 3.1).

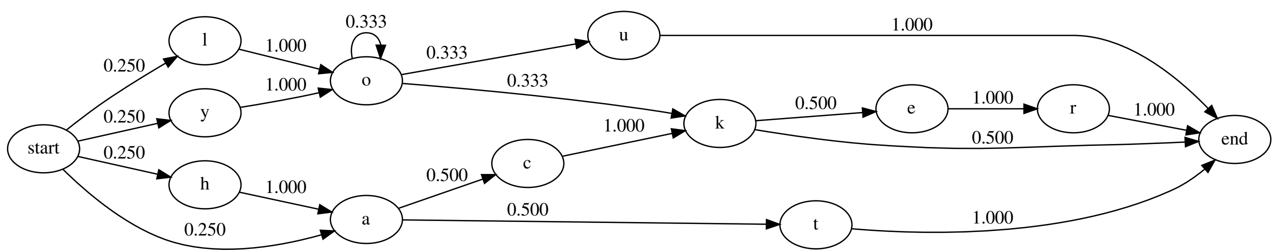


Figure 3.1.: Markov chain for words in *Look at you, hacker*. We ignored white space and punctation.

The problem with Markov Chains is their lack of understanding for context. While the concept of Markov Chains can be extended to consider the previous $n \in \mathbb{N}$ tokens, this "window" is still fixed. For C code we expect our model to remember what variables were defined in the current function and what their types are in order to make more educated decisions whether this code is similar to man-made C.

3.1. Neural Networks

Neural Networks are a collection of artificial neurons. Each neuron has multiple inputs and one output. Each input to a neuron is multiplied with a value $w_i \in \mathbb{R}$ called weight. The neuron sums all inputs and produces an output by running the sum through a possibly non-linear function. Neurons are organized in layers. The outputs of one layer are connected to the inputs of the layer above. The last layer is called the output layer, the first is the input layer. Between are the hidden layers. A network with multiple hidden layers is called deep neural network. These neural networks are able to approximate every continuous function [20]. In order to approximate a function the network needs to "learn" it. For this we need a set of inputs with their expected output. We first partition these into two sets. One is the training set, the other the test set. The former is used to teach the network to approximate our function, the latter is used to verify whether the network is able to abstract from the concrete training samples. If not, the network has been *over fitted*. This means the network fails to work on inputs that even slightly derivate from the training set.

Learning is done by first initializing all weights with random values. Then, the network is presented an input from the training set. We compute the output of the network and compare it to the expected output. We then adjust the weights to move the actual output of the network to the expected from the training set. The difference between computed and expected outputs is called *loss* or *error* and is determined by a loss function. A simple loss function is the sum of squares (TSS)

$$\sum_i (t_i - y_i)^2,$$

where t_i are expected and y_i are the computed outputs. Another popular loss function is the *cross entropy*

$$-\frac{1}{n}\sum_i [t_i \ln y + (1 - t_i)\ln(1 - t_i)].$$

The cross entropy takes into account how well the neuron approximates the function when computing the loss. This helps to slow down learning when approaching the (local) minimum to prevent overshooting it.

In order to know how to change the weights to improve the output we compute the derivation of the whole network in respect to the weights. This is called a backward pass. The chain rule for derivation means we can propagate partial derivates from the output layer, backwards to the input layer. The backward pass will give us a gradient vector. We use it to change the weights in the direction that minimizes the loss. The amount of change is governed by the learning rate. A higher learning rate speeds up convergence, but also can introduce oscillation while smaller learning rates slow down learning. After adjusting the weights, the algorithm proceeds with the next training sample. After all samples in the training set have been processed, the loss for all samples in the test set is computed. This value is then used to judge whether the network was able to learn the function we are interested in. The expectation is that the loss of the network is the same on the training set as well as on the test set.

The simple feed-forward networks described above are extremely useful for image and speech recognition tasks as well as classification, but they lack a crucial element of real neural networks. They have no concept of short term memory.

3.2. Recurrent Neural Networks

Instead of a fixed size image or voice snippet, we are interested in applying a neural network to a sequence of characters. The sequence does not have a fixed length and every character can possibly depend on all previous characters. For example, the set of allowed identifiers in C code depends on what variables, functions and types have been defined before. For this the network needs some kind of short term memory [1].

One type of neural network that has memory is a *Recurrent Neural Network*. Here, some output of hidden layers feeds back into the network. One way to visualize this are loops in the networks hidden layers. A more accurate way to represent this is to unfold the network while feeding it characters. Now, one networks hidden layer connects to the next one. The first network gets the first character as output and the second network the second character. This repeats until all characters have been consumed. The expected output of the network is the next character in the sequence. The main hurdle to applying this naive model of memory is the *Vanishing Gradient Problem* [29]. When deriving the network for the backward pass, the portion of the gradient that models the influence of the recurrent part of the network has the form

$$f'(\vec{x})^s.$$

where s is the number of characters consumed since. Depending on the gradient of the error function this value approaches either infinity or becomes 0 very fast.

The solution is to use *Long-short term memory* (LSTM) cells [30] (c.f. figure 3.2). Here each cell has a separate input, output and reset gate. Each gate is a neuron. The input gate gets the cells content and the current input. The output gate gets the output of the input gate and the current contents in the LSTM cell and decides the output of the cell. The reset gate gets the current input and the previous contents of the cell and decides what the new cell contents are. While this architecture is more complex it also avoids the vanishing gradient problem and allows the gradient descent based learning described previously.

Recurrent Neural Network neurons expect inputs to be in the real numbers, so we need to transform characters into vectors in $\mathbb{R}^n$. We use the so called *One-Hot Encoding*. We represent each character as a natural number n and set the nth element of the input vector to one. All other elements remain 0. The same encoding is used for the expected output. After training, the neural network will be making predictions for the next character given an input sequence. The output is a vector of the relative probability for the respective character.

Now we have a recurrent neural network that makes predictions about the successors of a character sequence. This can be used to generate random C code. To do this we feed it a partial program as character sequence and sample from the distribution for the next character. We append this character to the partial program and repeat.

[1] Note that this is different from long term memory that is embedded in the weights we learn. While short term memory remembers specific facts about the current function, long term memory knows about universal facts of C source code.

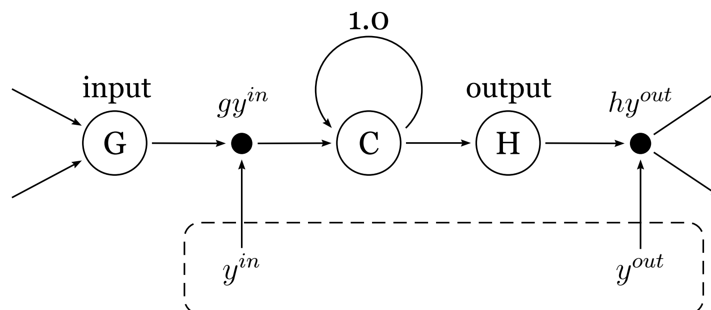


Figure 3.2.: Schematic of a LSTM cell. The input gate receives the cells output from the previous character, combines it with the current network input and sends it to the reset gate. The reset gate combined the new value with the old and sends it to the output gate. The output gate combines the value with the next character. Adapted from [30]

3.3. Readability Scoring

We are interested in using the neural network to compute a score for the readability of a given piece of decompiled C code. The following aspects influence readability:

1. Formatting of white space.
2. Naming identifiers (functions, variables, types, preprocessor defines).
3. Number of comments and whether they correctly explain intent.
4. Appropriate use of language features.

For our readability score white space formatting is not of concern as the C code will be formatted automatically. The decompiler is not able to understand intent of functions, so will not add comments. What remains is identifier naming and choice of language features. Here the neural network will help. We will reuse the loss function used while training the network as a score for readability. The loss function is a measure for how well the neural network was able to predict a given character sequence. If we assume that the code used to train the model is readable, the networks prediction is also readable code. As close as the decompiled code is to the prediction of the network, as more readable it is.

3.4. Summary

In this chapter we gave a rough overview of how recurrent neural networks work. They are presented with a sequence of tokens and will produce a prediction for the next token. Internally, they consist of LSTM cells that model short term memory. This way the network "remembers" previous tokens. We use this prediction property of the RNN to build a readability score for C code. The network is trained on real source code and then queried to make a prediction based on decompiled code. The closer the prediction is to the actual next token, the better the score.

4. Language Model Directed Decompilation

Now that we covered the theoretical underpinnings of both decompilation and recurrent neural networks it is time to develop an iterated refinement algorithm for recovering C code. First, we describe the exact hyper parameters, topology and training procedure of the neural network. We show how to embed this into the decompilation process to guide C code transformations. We close with a description of all transformations used.

4.1. Training the Language Model

The Recurrent Neural Network (RNN) is build using the Torch7 framework [16], written in Lua. As a basis we used the high-performance, reusable RNN and LSTM modules provided by J. Johnson [1]. The exact network we used has four hidden layer and accepts sequences of length 50. The network was trained with a batch size of 50 sequences over the course of 3 months. For training, we used the Linux kernel version 4.15. No special preprocessing was done, we simply concatenated all C code files into a 460 MiB large sequence. After training for 3 months the average loss on the training set was 1.754 and 2.014 on the test set.

In order to connect the RNN to the decompiler we implemented a version that accepts sequences from the local network. The RNN reads C code from a TCP connection, computes the cross entropy loss and sends the value back. We call this the measure server. As the RNN can only handle sequences of exactly 50 characters while inputs into the measure server have arbitrary length, some processing has to be done. Inputs larger than 50 characters are cut into smaller pieces and measured in batches, inputs that are smaller than 50 characters are repeated until they are large enough.

4.2. An Iterated Refinement Algorithm

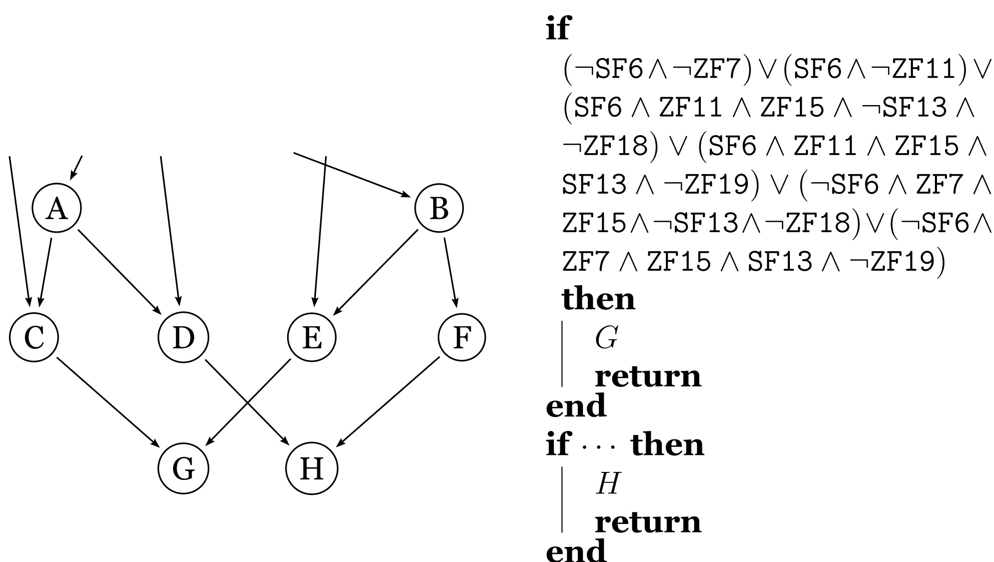
Now, we have everything to develop the iterated refinement of decompiled C code. The algorithm disassembles a given binary and produces a set of functions. A function is represented as a sequence of RREIL instructions. From this, an initial abstract syntax tree (AST) is derived. The AST is then fed into the refinement algorithm. In each iteration the algorithm first forward propagates all expressions, then removes all dead code from the AST. It then randomly picks a transformation to try on the AST, applies it and sends the result to the measure server. If the readability did not

[1] https://github.com/jcjohnson/torch-rnn

degrade too much or improved, the transformation is kept. This loop continues until the AST stops improving.

4.2.1. Deriving the Initial Abstract Syntax Tree

Before we derive the initial AST some transformations have to be done in order to make the CFG more digestible for the algorithm outlined in section 2.4. First we "inline" common exit nodes. Most compiler emit CFG that only have one exit node. This simplifies cleaning up the stack before executing `ret`. Sadly, the AST construction algorithm we use tends to produce superfluous if-conditions in the presence of shred exit nodes, as it is unable to determine that the path condition of the exit node cannot be falsified. Figure 4.2 show a typical example with two shared exit nodes.



(a) Exit nodes of the control flow graph without inlining.

(b) Resulting pseudo code.

Figure 4.1.: Lack of inlining causes excessively large `if`-condition before `return` statements. It also makes it hard to recognize that control flow never falls through the second `if`-statement.

While it would be possible to check whether the if-condition is needed using a boolean satisfiability (SAT) solver, this would incur a $\mathcal{O}(2^n)$ time-complexity. A far simpler way is to copy the exit node for each additional incoming control flow edge so that every edge points to its unique exit node.

The inlining algorithm first collects all inlinable sub graphs in the CFG and sorts them by their node count in ascending order. It then selects the first one, inlines it and iterates. This process stops as soon as the smallest inlineable sub graph has more than two nodes, avoiding too much inlining. Without it the decompiler would copy nodes until every loop-less path through the CFG has its unique set of nodes. This would increase the function size by roughly n^2. Special care needs to be taken when copying basic blocks. The RREIL instructions are in SSA form which demands that only a single definition for every variable plus subscript exists in a function. If we would simply copy the basic block, every definition would exist twice. To prevent this

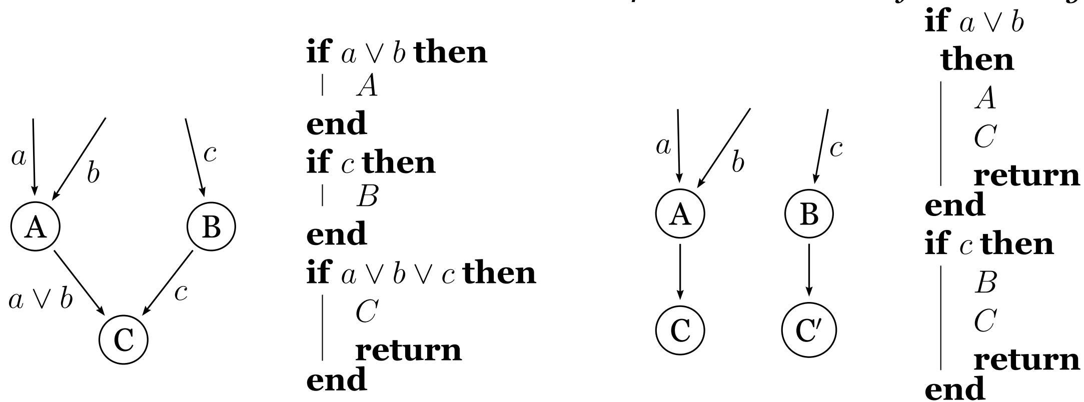


Figure 4.2.: Results of exit node inlining.

we rewrite all subscripts in the copy to unique values. Another complication arises with copying Φ operations. [2] Both the copy and the original basic block have less incoming edges and thus some arguments of the Φ operations have vanished. These need to be removed, otherwise data flow analysis could determine that the register values of one branch depend on those from the other branch.

After this, we remove all unresolved jumps from the CFG and run the algorithm described in section 4.2.1. This produces a *pre-AST* where each node in the AST is either a C control flow construct, a dummy conditional node or a reference to a basic block.

This pre-AST is then converted to the actual AST by replacing all basic blocks with sequences of statements. For the most part, this is a straight forward process as the RREIL instructions inside the basic blocks are in SSA form. Each RREIL instruction of the form

$$A_n := op(B_m, C_o)$$

is converted into a C statement of the form

$$\texttt{a_n} = \texttt{b_m}\ op\ \texttt{c_o};.$$

The subscripts of the SSA variable names are simply omitted. The AST has a separate node for the RREIL Φ instruction. While C is not in SSA form, this data flow information is useful for later transforms. This is why we keep it for now.

Special care needs to be taken when handing procedure returns and calls to other functions. While our RREIL dialect has a return instruction it does not return a value, it simply marks the end of the procedure. This is due to the fact that RREIL is optimized to be emitted directly from the disassembler without previous static analysis. The concept of return values does not exist on the assembly level. While it is common to use `EAX` or `RAX` to return values [3] nothing on the ISA level enforces this. Also, in AMD64 instruction sets the register depends on the size of the return value, e.g. a 16 bit value would be returned in `AX` while a compiler will use `EAX` for a 32 bit value. To correctly infer return values we again use the reaching definitions analysis to find a set of final values for `AL/AH/AX/EAX/RAX`. We then introduce a fresh variable with the

[2] Φ operations “tie” data flow together where control flow joins. For example, after an if/else statement with both branches setting the variable x a Φ operation is inserted to encode that both values are possible for x.

[3] For example the System V ABI returns most values via `EAX/RAX`.

size of the largest reaching definition of the A register and insert Φ instruction like this:

$$\mathtt{v44} = \mathtt{PHI}(\mathtt{EAX_22}, \mathtt{RAX_12});$$
$$\mathtt{return\ v44};$$

Similar, the RREIL call instruction has no defined return value but a set of registers it writes. In order to get the return value of the function, we look at the write-set. If members of the A register are in it, we introduce a fresh variable, assign the return value of the function to it and copy its contents to all written registers:

$$\mathtt{v45} = \mathtt{func}(\mathtt{a}, \mathtt{b}, \mathtt{c});$$
$$\mathtt{EAX_22} = \mathtt{v45};$$
$$\mathtt{RAX_23} = \mathtt{v45};$$

For the function's arguments we select only the largest member of a register group in the read-set. For example, a read-set of `{AX, EAX, RAX, CX, ECX, ESP}` the resulting function call is `func(RAX,ECX)`. Note that we filter out the stack pointer. We do not handle functions with multiple return values. One way to do this is to add all members of the write-set that are not `*SP` or `*AX` as out parameters to the functions arguments.

When converting basic blocks to statements, we simultaneously run a dead code elimination pass. This prevents the initial AST from becoming too large. Dead code is one of the main problems for the decompiler. This is largely because the disassembler cannot know whether a register or flag is needed later on, so it has to generate code for everything. Common operations like additions, multiplications and bitwise AND set various flag registers reflecting the result. The majority of these flags are never read before they are written again. Similar, with setting "sub-registers" like `AH` or `CL`. This results in a huge percentage of the RREIL code being useless. In order to find all operations that contribute to observable behaviour of the function we remember all SSA variables that are used to read or write from or into memory, used as function arguments, are return value of the function being decompiled or are used in control flow constructs. Starting from this set we collect all SSA variables that contribute to their values. For example the code

$$\mathtt{a} := \mathtt{b} + 1;$$
$$\mathtt{c} = *\mathtt{a};$$

would add `a` and `b` to the set of needed SSA variables. As the SSA form means every operation in the function is uniquely identified by a SSA variable, we can skip every RREIL instruction that is not in the "live" set. We run the conversion/dead-code-elimination algorithm backwards and in post-order across the AST and thus can do both in a single walk over the tree. Sources of live variables for dead code elimination are

- variables read and written by functions called by us,

- downwards exposed variables, i.e. return values,
- variables used in conditional statements and loops,
- addresses of memory operations,
- values written to memory outside of the stack frame and
- values necessary to compute other live values.

Figure 4.3 shows an example run of the dead code elimination algorithm. The majority of the dead code comes from AMD64 flag registers being computed but never used for branching and moving results into sub-registers.

```
res = RSP + 8
SF = res < 0
ZF = res == 0x0
CF = res < RSP
AF = res < RSP
af1 = af1 & AF
of1 = RSP ^ 8
of1 = of1 ^ 2^64 − 1
of2 = RSP ^ res
OF = of1 & of2
val = res
SPL = val
SP = val
ESP = val
RSP = val
if ZF then
|   ESP = ESP + 4
|   ...
end
```

(a) Before dead code elimination.

```
res = RSP + 8

ZF = res == 0x0







val = res


ESP = val
RSP = val
if ZF then
|   ESP = ESP + 4
|   ...
end
```

(b) After dead code elimination.

Figure 4.3.: Example of a dead code elimination.

Finally, we replace the DNF formula inside control flow constructs with C expressions. The result is an initial AST that serves as a starting point for the refinement loop. We use a heterogeneous AST type to allow easy use of pattern matching. Each AST node type has a unique identifier to allow attachment of additional data to nodes. The full definition of the AST data structure is show in Figure 4.4.

4.2.2. Available Expressions, Dead Code Elimination and Constraint Propagation

Now that we have an initial AST we start the iterated refinement. The loop consists of three phases. First, we clean up the AST by running an available expression analysis and subsequent dead-code elimination pass. This is necessary as the previous iteration may have removed uses of variables it deemed superfluous.

```
pub enum Ast {
  Sequence{ id: usize, nodes: Vec<Box<Ast>> },
  Block{
    id: usize, statements: Vec<Statement>,
    reaching_defs: Vec<Lvalue> },
  Comparison{
    id: usize, marker: Str,
    vertex: Option<ControlFlowRef> },
  Condition{
      id: usize, condition: Bool2,
      condition_expr: Option<Box<Ast>>,
      yes: Box<Ast>, no: Option<Box<Ast>> },
  Loop{
      id: usize, invariant: Bool2,
      invariant_expr: Option<Box<Ast>>,
      body: Box<Ast>, head_controlled: bool },
  Call{ id: usize, name: CallTarget, arguments: Vec<Box<Ast>> },
  Return{ id: usize, value: Option<Box<Ast>> },
  Assignment{ id: usize, lvalue: Box<Ast>, rvalue: Box<Ast> },
  BinaryOp{ id: usize, operation: Str, left: Box<Ast>, right: Box<Ast> },
  UnaryOp{ id: usize, operation: Str, value: Box<Ast> },
  Variable{ id: usize, name: Str, width: usize, subscript: usize },
  Constant{ id: usize, value: u64, width: usize, base: usize },
  Break{ id: usize },
  MemoryRef{ id: usize, address: Box<Ast> },
  MemoryDeref{ id: usize, address: Box<Ast>, field: Str },
  Phi{ id: usize, lvalue: (Str,usize), arguments: Vec<(Str,usize)> },
  Decl{ id: usize, typ: Str, name: Str, initializer: Option<Box<Ast>> },
}
```

Figure 4.4.: Definition of the AST sum type. The *Comparison* and *Block* are used for the pre-AST. After the control flow constructs have been inferred, they are removed.

The fact that all variables in the AST are still in single static assignment form makes the available expression analysis trivial. It is implemented as a single recursive function that walks the AST as depth-first search, registering for every assignment expression on the form $a_n :=$ *expr* that expression *expr* is available as a_n. Figure 4.5 gives a simple example of an available expression analysis run. We avoid putting function calls into this expression list because the following forward propagation of expressions could potentially duplicate the function call. If the function call has side effects this would change the observable behaviour of the program.

The following dead code elimination pass works similar to the one done while converting RREIL code to AST nodes. Sources of observable behaviour are marked and needed expressions are marked working back from there. Everything not marked is removed from the AST.

Then, we run a simple path constraint propagation. Here we collect all conditions necessary for each node the in AST to be reached. These conditions are propagated from if- and while statements downwards the AST. Conditions are represented as integer intervals for variables. If an AST node has the path constraint

$$\{a_1 \in [0;10], b_2 \in [50;2^{64}-1]\}$$

```
res = RSP + 8
ZF = res == 0x0
val = res
ESP = val
RSP = val
if ZF then
|  ESP = ESP + 4
|  ...
end
```

(a) Before available expressions.

```
res = RSP + 8
ZF = res == 0x0
val = RSP + 8
ESP = RSP + 8
RSP = RSP + 8
if RSP + 8 == 0x0 then
|  ESP = RSP + 8 + 4
|  ...
end
```

(b) After available expressions.

Figure 4.5.: Example of a available expressions analysis and subsequent forward expression propagation.

which means that if- and while statements leading to that node made sure that the variable a_1 is not larger than 10 and variable b_2 is larger than 49. These path constraints are then used to eliminate superfluous if-conditions. These happen either due to transforms on the AST during the refinement loop or from the inlining of exit nodes. See Figure 4.6 for an example of how the algorithm proceeds.

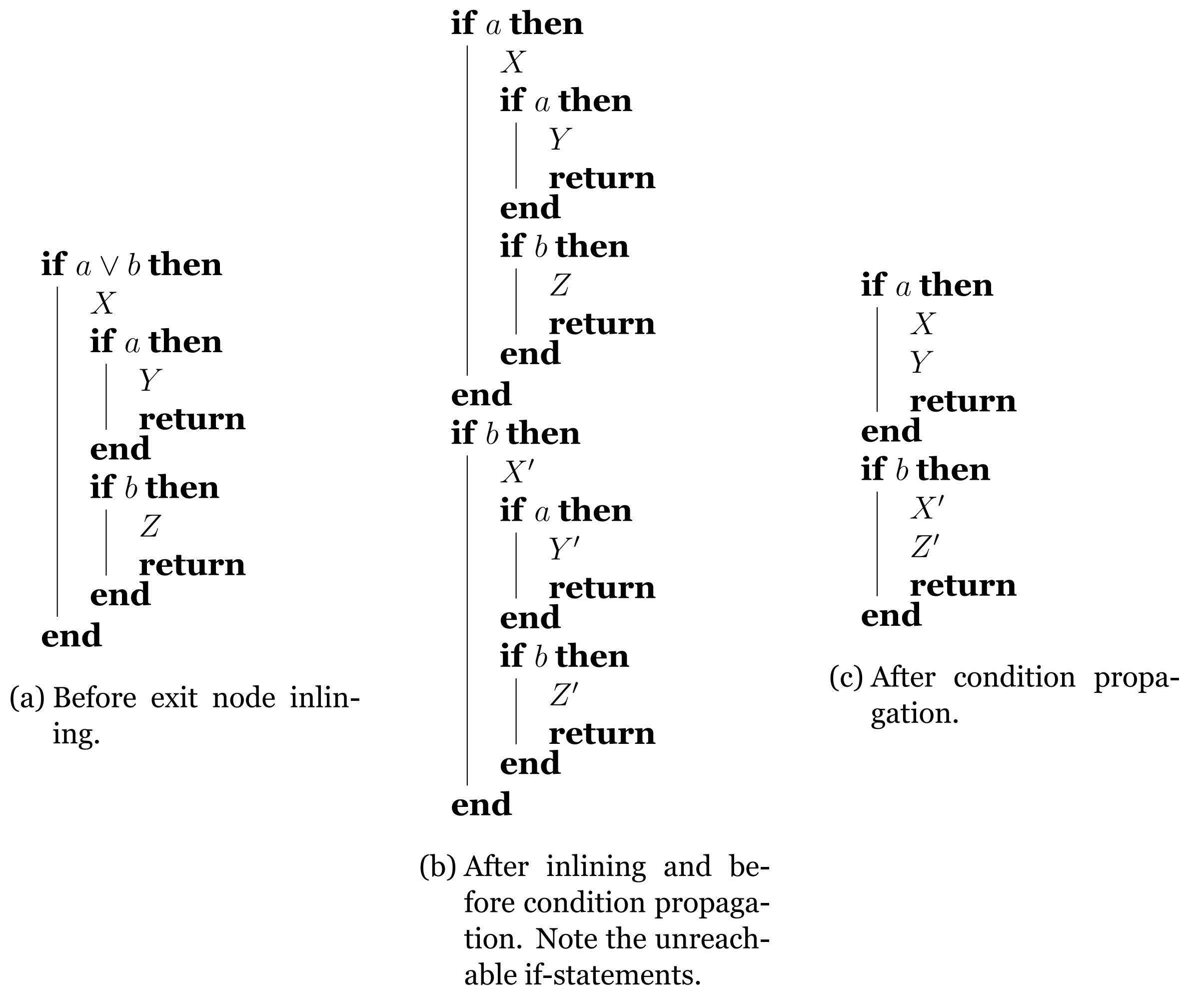
4.2.3. Transformation Selection

Second, we run a random transformation on the AST. The selection process is completely unguided and only works on the syntax of expressions. We also do not transform expressions to a normal form. While this could potentially improve the results, implementing this kind of computer algebra system (CAS) is out of scope for this thesis. Transformation rules are tree patterns matched against the AST. The list of patterns is processed one by one, if one pattern matches we note this by incrementing an a integer variable. The result is the number of matching patterns. Now we sample a number between zero and the number of patterns found (excluding). We run the same function again, this time with the sampled transformation number as an additional argument. The function will traverse the AST in the same order keeping track of the applicable transforms. If it reaches the transform in the argument it applies it to the AST and returns. This has the advantage that the logic for matching and applying transforms live in the same function and share code. New transforms only need to be added to a single function and there is no risk mismatching code for matching and applying transforms.

Before the second run of the function we copy the AST as the transformation is done destructively.

4.2.4. Measuring

Applying a transform produces a new AST, and we measure the readability of the AST using our neural network. For this, we first convert the AST into C source code. Before we can do that, we need to infer all local variables and their scopes. One of the downsides of using SSA form for the AST is the lack of lexical scoping. All variables definitions are visible at every point in the function, without prior declaration. This



(a) Before exit node inlining.

(b) After inlining and before condition propagation. Note the unreachable if-statements.

(c) After condition propagation.

Figure 4.6.: Example of condition propagation removing superfluous if-statements.

makes sense for assembly where all "variables", i.e. registers, are globals. In C we need to declare them inside to function. We heed the rules of C89 [6] and declare all local variables at function entry. This is done by collecting all used variables, ignoring their SSA subscript and prepending the functions body with their declarations. We run a simple unification-based type inference to figure out what type to use. In parallel, we note all variables that are read before written (e.g. upwards exposed from the entry point). These are the arguments into the function.

Now we have enough information to emit C code. Because we used the Linux kernel source code to train our network we emulate the Linux kernel coding style by indenting with tabs and putting the opening brace on the last line. After we turned the AST into a single character string we open a TCP connection to the measure server and send the string. The measure server splits the string into batches and computes the cross entropy loss and sends it back.

The cross entropy loss is used as the readability score that decides whether we keep the transform.

4.2.5. Metaheuristics

Two things are still missing from the algorithms' description: how to interpret the readability score and when to stop improving? A simple solution for the first question is to always favour the AST with the smaller readability score [4]. This has the problem that especially in the first few iterations, readability decreases. Only accepting strictly better ASTs causes the algorithm to be caught in a local maximum early on. Instead, we use a metaheuristic called simulated annealing [34]. Simulated annealing emulates the solidifying process of metals. While the metal is still hot, molecules move larger distances. When the metal cools molecules move less and less until they stop and the material becomes solid. The speed of cooling governs if the molecules are able to form larger or smaller lattice structures. The size of these lattices (called "grains") influences the material's hardness. Applied to decompilation, we allow changes to be large (and possible worsening readability) when we start with the refinement loop. While iterating, we "cool" the algorithm and only allow less drastic changes to the AST.

Modelling the rate of change over time is done by using a temperature and a cooling schedule. The temperature is a positive number that limits how much worse the readability score can get after a transformation. At every iteration the new ASTs score is compared against the current. If readability increased or decreased by less than the current temperature, the transform is kept. The temperature decreases over time towards zero. This limits exploration of the space of possible ASTs as time goes on and our confidence rises that we found the global maximum.

The exact rate of cooling is governed by the `cool()` function. The function is called for every iteration of the refinement algorithm. It gets the current temperature as input argument and returns the new temperature. We experimented with two cooling functions. The first cools at a steady state i.e.

$$\text{cool}(t) = \max(0, t - c)$$

and one that halves the time at each iteration i.e.

$$\text{cool}(t) = \frac{t}{2}.$$

Even when the temperature is zero, the refinement algorithm does not stop. The stop condition is independent from the cooling cycle and thus keeps running as long as it finds transforms that improve readability. A zero temperature only causes the algorithm to stop accepting transforms that do not improve readability. Still, in order for the algorithm to terminate we need a way to recognize that we reached a (possibly global) maximum. For this we keep track of the number of iterations the algorithm failed to make an improvement. If we stop improving for a fixed number n of iterations, the algorithm terminates. Choice of n is critical to the performance of the algorithm. If it is too small we may miss transforms that could have improved the AST, on the other hand a too large n causes a unnecessary slowdown because the algorithm tries to improve a AST that is already a local maximum. As our transformation selection strategy is completely random, the number of available transforms

[4] Remember the score is the cross entropy loss of the recurrent neural network. The smaller the value the closer the networks predictions where to the code we sent. As our assumption is that the network predicts perfect C code, *lower is better*.

need to take into account. Assuming of all x transformations y apply to the AST and improve readability we have a $\frac{y}{x}$ chance of selecting a transform that is successful and thus resets the termination counter. Assuming temperature is already zero and all iterations are independent experiments, the chance of selecting a successful transform over i iterations is

$$\frac{iy}{x.}$$

For a probability of 90% we need

$$\frac{ny}{x} \geq 0.9.$$

The worst case is that only a single transform exists, so

$$n \geq \frac{9}{10}x.$$

With 16 implemented transforms this leads to a stop condition of

$$n = 15.$$

After the refinement loop has terminated we run the same final processing as we did before sending the AST to the measure server with the difference that we output the C code into a file.

This concludes the discussion of the refinement loop's overall structure. Figure 4.7 shows the complete refinement algorithm as pseudocode. In the next sections we discuss the transformations we implemented in detail. We categorize transforms into those that change the control flow constructs and those that change only expressions. The former tend to have a larger impact on the readability score than the latter, thus we expect to see fewer changes in control flow as the temperature decreases.

4.3. Transformation Rules For Control-Flow

The following transformation rules work on C statements that deal with control flow. Figures 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4.14 describe the exact semantics as logical inference rules.

Flatten early exits

The initial AST tends to be deeply nested `if/else` statements with one branch ending in a `return`. This transformation tries to flatten this by exploiting the fact that the `return` statement is a dead end. An `if/else` construct with a `return` statement at the end of the `if` branch can be turned into an `if`-Statement for the without `else` branch. The code inside the original `else` branch is simply put after the `if`-Statement.

Simplifying `if`-Statements

Due to exit node inlining outlined in section 4.2.1 and the propagation of path conditions (same section) some `if`-statements have uncommon structure: a condition

```
lastImprov ← 0
temperature ← InitialTemperature()

ast ← InitialAst(cfg)
ConvertBasicBlocks(ast)
score ← Measure(ast)

while lastImprov < Patience() do
    ForwardPropagateExprs(ast)
    ForwardPropagateBounds(ast)
    DeadCodeElimination(ast)

    transform ← SelectRandomTransformation(ast)

    if transform = null then
        return
    end

    newAst ← ApplyTransformation(ast, transform)
    newScore ← Measure(ast)

    if score − newScore < temperature then
        ast ← newAst
        score ← newScore
        lastImprov ← 0
    else
        lastImprov ← lastImprov + 1
    end
    Cool(temperature)
end
```

Figure 4.7.: Language model-directed refinement algorithm.

$$\frac{Cond[c, n_t, n_f] \wedge n_f \text{ ends in } \texttt{return}}{Seq[Cond[c, n_t, -], n_f]}$$

Figure 4.8.: Rule for flattening early exits.

that is always true or always false as well as empty `if` or `else` branches. We implement two transforms that remove the `if` around statements that are always true and delete statement that never are. Also, we remove branches that are empty. If the statement only has an `else` branch, we invert the condition and turn it into a simple `if` statement.

$$\frac{Cond[\top, n_t, n_f]}{Seq[n_t]}$$

Figure 4.9.: Rule for simplifying `if`-statements.

Swapping `if` and `else` Branches

One of the less obvious transforms is to swap the `if` and the `else` branches of conditional statements. In man-made C code, the true branch often handles the "common" case whereas the `else` branch is for the error handling. While our decompiler cannot infer intent, the neural network may have a limited understanding of what error handling code looks like. The idea of this transform is that putting the error handling into the `else` branch should improve the readability score and the refinement algorithm will keep the "correct" order. This is also one of the transforms that is an involution. One of the `if/else` orders must have the better readability score and thus the fact that the transform always applies does not cause an infinite loops.

$$\frac{Cond[c, n_t, n_f]}{Cond[\neg c, n_f, n_t]}$$

Figure 4.10.: Rule for swapping `if` branches.

Converting cascading `if`-Statements to `switch`

The `switch` statement is probably the only control flow statement that has no direct correspondence in machine code. Compilers employ various strategies to translate a `switch` to efficient machine code. For dense values, i.e. values that are close to each other with very little "gaps" a table mapping values to addresses is used. The addresses point to the bodies of the `case` statements. Most compilers interleave the table with the code to improve cache coherency. With the table in place, the `switch` is simply a memory load from the table and an indirect jump to the right basic block. Statements that cannot be implemented using tables degenerate to a sequence of `if/else` statements. The latter can easily be converted back into a `switch` by looking for cascades of `if/else` statements whose conditions have the form a_i `==` c with a_i being a variable and c an integer constant.

The transformation rule is rather simple. All statements of the form `if (`a_i `==` c`) {` b `}` are converted to `case` c`:` b`; break;`. If the cascade ends with an `else` statement, this is converted into the `default` case.

$$\frac{Cond[v = i_1, n_1, Cond[v = i_2, \cdots, n_2]]}{Switch[\forall j \; (v = i_j, n_j)]}$$

Figure 4.11.: Rule for creating `switch`-statements.

4.4. Transformation Rules For Expressions

The second group of transformation rules handle expressions.

Simplifications

Two transforms remove assembly code artefacts from the C expressions. To compare a register to zero compilers often emit an AMD64 `test` r`,` r instruction, comparing

the register with itself. It is lifted to the RREIL snippet `and tmp, r, r; cmpeq ZF, tmp, 0`. This results in the C expression `if (r & r == 0) { ··· }` ant further simplifies to `if (!r) { ··· }`.

$$\frac{Cond[a \wedge a, n_t, n_f]}{Cond[a, n_t, n_f]} \qquad \frac{Cond[a = a, n_t, n_f]}{Cond[a, n_t, n_f]}$$

Figure 4.12.: Rule for expression simplifications.

Integer Logic Equivalences

Similar, complex integer inequalities produce certain artefacts due to their translation to machine code. The "jump if greater or equal to zero" instruction is lifted to a logical OR of the zero and the sign flag. This causes the C expression

$$\texttt{if}(r >= 0)\{\cdots\}$$

to be decompiled to

$$\texttt{if}(r == 0 || r > 0)\{\cdots\}.$$

We implement transforms to look for expressions of the form `r == 0 || r > 0` and transform the to `r >= 0`. Comparisons with non-zero constants also introduce artefacts. AMD64 has a "compare" instruction. Instead of comparing directly it subtracts the constant from the register and compares the result with zero. Thus, `cmp r, c` that is translated to the RREIL snippet

```
sub tmp, r, c
cmpeq ZF, tmp, 0
cmple SF, 0, tmp
```

This results in the C expression $r - c == 0 || r - c \diamond 0$. Translating expression of the form $r - c \diamond 0$ to $r \diamond 0$ reverses this.

A "less than or equal to" comparison is decompiled to `!(r - c > 0)`. This happens because the sign bit is inverted when checking if a value is less than [5]. To make the expression more readable we also introduce a transformation that "pushes" negations into the expression and inverts the relational operator. So the expression above is transformed to `r - c <= 0`.

$$\frac{Cond[a - x \diamond 0, n_t, n_f]}{Cond[a \diamond x, n_t, n_f]} \qquad \frac{Cond[\neg(a - x \diamond 0), n_t, n_f]}{Cond[a \bar{\diamond} x, n_t, n_f]}$$

$$\frac{Cond[a - x \circ 0 \vee a = 0, n_t, n_f]}{Cond[a \circ = x, n_t, n_f]}$$

Figure 4.13.: Rule for integer logic equivalences.

[5]The comparison instruction cannot be reversed like `cmp c, r` because AMD64 demands the first argument to be a register or memory location.

Number Base Conversions

Integer literals in C can be written in different number bases: octal, decimal and hexadecimal. What base to use for representation depends largely on personal preference and the conceptional meaning of the literal. If the literal is used as a bit field, mask or as constants in logical operations hexadecimal popular. For integer arithmetic decimal is used more. UNIX file permissions are often written in octal. Inferring the purpose of a particular literal in the programmers mental model of the function being decompiled is difficult, especially automating it. Our solution is to change the number base of integer literals and use the neural network to tell us what looks "right".

Aside from switching between octal, decimal and hexadecimal we also try to convert literals from signed to unsigned values as well as to characters if the value is between zero and 127. All the previous transforms had only one "direction", i.e. applying them repeatedly will cause the AST to settle in a state where no more transformable patters exists. The number base conversion is different, it always applies to every literal. We depend on the metaheuristic to make sure we do not get trapped in an infinite loop changing the number base of one literal from decimal to octal and back.

Reversing Strength Reductions

The final group of transformations try to reverse common strength reduction optimizations compilers employ. Strength reduction [17] is the process of changing expensive ("strong") operations into less expensive ("weak") equivalents. The disadvantage is that these changes may make the code less readable. We implement two transforms reversing strength reductions to give the neural network a chance to see whether this optimization was applied.

First, we transform integer multiplications into divisions. All integer arithmetic on AMD64 is done in the ring $\mathbb{Z}/2^n\mathbb{Z}$ where $n \in \{8, 16, 32, 64\}$. Division in rings is defined for divisors that are co-prime to 2^n. The division $\frac{a}{b}$ can be written as ab^{-1} with $bb^{-1} \equiv 1$. Being able to reformulate division as multiplication opens an avenue for optimization of machine code. Many CPUs lack division instructions and even if they have it, they are often slow. Compilers can turn division by constants into multiplications as long as they are co-prime and do not cause an underflow [6]. Same as the number base transforms no there is not particular reason why a given multiplication could be an optimized division, so we blindly rewrite all multiplications with one of the factors being co-prime as division, keeping those that improve readability.

Second, we can rewrite bit shifts as multiplication or division by powers of two by exploiting the two's complement representation of integers used in modern CPUs. For left shifts this transform is straight forward, for right shifts i.e. divisions the arithmetic right shift is used. While shifting left always moves a zero bit into the lowest position, right shift has two definitions. One, the logical right shift also moves a zero bit into the most significant bit position. Second, the arithmetic right shift duplicates the bit previously in MSB position. This is necessary when right shifting negative numbers. Assuming 16 bit integers

[6] An easy fix would be to check before dividing if the divisor is less or equal than the dividend.

$$a \equiv -8_{10} \equiv 1111111111111010_2$$

shifted right by one bit using logical shift is

$$01111111111111101_2 \equiv 32765_{10},$$

using arithmetic shift it is

$$11111111111111100_2 \equiv -4_{10}.$$

Replacing divisions by two with arithmetic right shifts is only correct if the remainder is zero. Shifting -7 by one produces -4 which is not correct as integer division in C rounds towards zero. As we already mentioned we care more about readability than correctness, so we try the transform anyway.

$$\frac{n >>_u m}{n/2^m} \qquad \frac{n << m}{n * 2^m}$$

$$\frac{n * m \quad \exists a : a * m \equiv 1}{n * a}$$

Figure 4.14.: Rule for reversing strength reduction optimizations.

4.5. Identifier Selection

For now, we used the recurrent neural network only to compute a readability score. There are another ways to utilize its understanding of C source code. Remember that an RNN is a predictor for the next character in a given sequence, this can be exploited to construct an algorithm that generates identifiers for functions and variables. Identifier names are among the source level information that is lost during compilation. RetDec and our implementation generate new identifiers using a simple schema. While this is easy to implement is also produces rather hard to distinguish names like `v12`, `v1` and `g3`. Hexrays uses cues from the type inference to select identifier names that match up with their semantics. If Hexrays determines that a local variable is only used as an argument for a function, it is named after that argument. This means if the parameter of a function flows directly into the first argument `fopen`, this parameter will be named `pathname`. Obviously, this fails if the local variable or parameter is not used with external, known functions.

The idea of language model directed identifier selection is to use the RNN to select the most "readable" identifier for a local variable based on the preceding source code.

We implemented this by building another TCP server that accepts C source code and runs it through the RNN. Instead of returning the loss it uses the prediction to sample an identifier.

When the decompiler introduces a new local variable it generates the C source up to the point where the variables name starts, sends this text to the server which computes a vector with the distribution for the next character. The server samples from that, appends the sampled character to the source code and iterates. This process repeats until a non-identifier character is sampled. Obviously, the sampling

is restricted to characters allowed in identifiers and whitespace. We also limit the number of sampled characters to 10.

The sampled identifier is then returned to the decompiler.

4.6. Summary

In this chapter we described an iterated refinement algorithm for C ASTs. The algorithm consists of three parts. One, a set of transformations on the AST that may improve readability by rewriting control structures and expressions to (largely) semantically equivalent elements. Two, a recurrent neural network trained on man-made C code. Predictions of the model are used to create a readability metric for ASTs. This metric is the loss of the AST when converted to a character string and fed into the RNN. The metric is used to determine whether a transformation improved readability. The assumption is that more readable C code is closer to the prediction of the RNN and thus produces a smaller loss. Three, a metaheuristic based on simulated annealing to help the refinement algorithm settling on a global maximum without being forced to enumerate all possible transforms.

The transformations are local and derived from typical compiler optimizations and artefact of the disassembly, lifting and control flow recovery processes that produced the initial AST. The RNN and simulated annealing allows us to have very little requirements for the transformations employed. We only expect them to preserve semantics and not yield invalid ASTs, nothing else. In particular, we do not expect them to always improve readability or to have a fixed point (some of them are involutions).

The algorithm refines every function of a program in isolation. The only inter-procedural information needed is the set of registers read and written by functions that are called. As this information can be gathered before running the refinement algorithm and does not change while the algorithm proceeds it is extremely easy to parallelize it.

5. Results

In this chapter we evaluate the language model based refinement algorithm against two existing decompilers RetDec and Hexrays. We will do both a *qualitative* and *quantitative* comparison. First, we compare run-time and memory consumption as well as size of the decompiled functions. Second, we will compare the quality of the output by a random sample of decompiled functions. We will discuss common and specific shortcomings of our decompiler as well as RetDec and Hexrays. Third, we compare relative completeness of disassembly.

5.1. Test Set

In order to evaluate our approach to traditional decompiler we used the GNU `coreutils` package. The package contains various GNU/Linux utilities like `cp` and `ls`. We choose it because it provides a set of reasonable sized test cases. All 105 programs in the package are between 30 and 137 KiB large. This helps as our decompiler as well as RetDec scale poorly for large programs. The `coreutils` package is Free Software and thus we are provided with the source code. This means we have a “ground truth” to compare our decompilation results to. To improve reproducibility of our results we use the binaries built by the Debian [1] project. All Debian packages are built with a standardized setup for all instruction set architectures. Because RetDec only supports x86 and our implementation only AMD64 using the coreutils package helps to make the result comparable. The exact version used for evaluation is `8.26`.

All tests were run on an Intel Core i5–3320M running at 2.60 GHz, with 16 GiB memory. RetDec was run under GNU/Linux compiled from source [2] using commit `d1090644`. We used the Windows version of Hexrays shipped with IDA Pro 6.8. Our disassembler was also run under GNU/Linux. The binaries were decompiled in one batch using a single process so decompilers had a chance to use both cores (four Hyper Threads).

5.2. Overview

5.2.1. Running Time

Running times were measured using the `time` utility on GNU/Linux and the `Measure-Command` function of Powershell on Windows. Both measure the total running time including time spend in the kernel.

Figure 5.1 shows the runtime of the three decompiler on a subset of the test set. The complete version of this and the following figures were moved to Appendix A

[1] `https://debian.org/`
[2] `https://github.com/avast-tl/retdec`

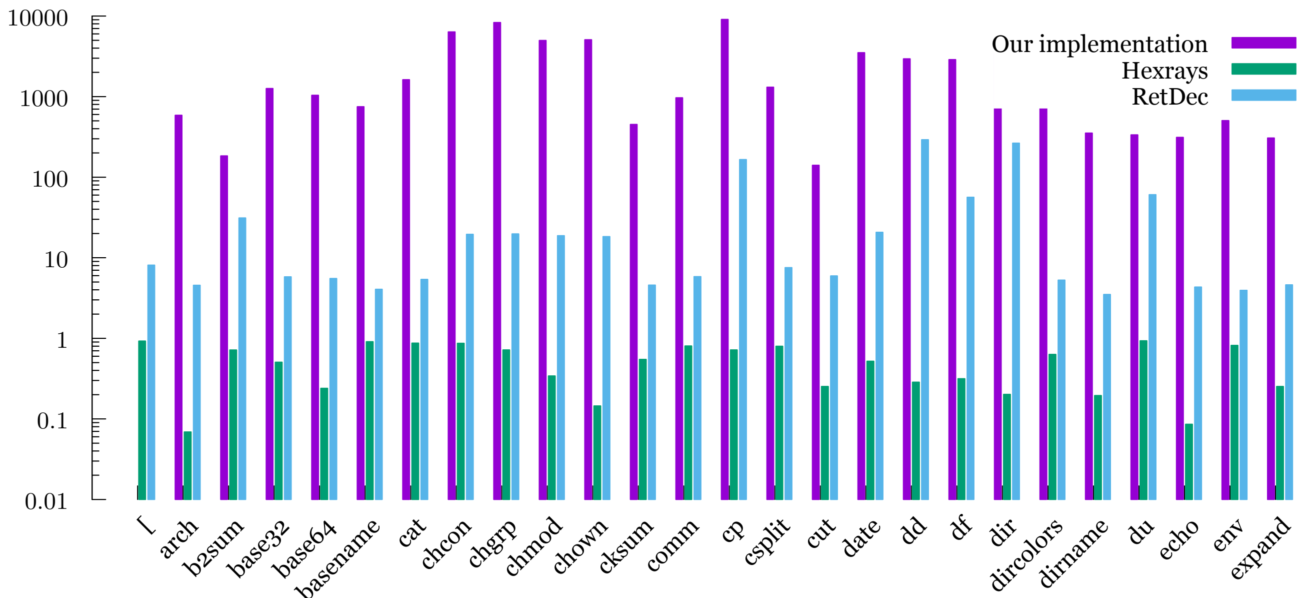


Figure 5.1.: Time spent decompiling a subset of `coreutils` in seconds.

due to size constraints. Please note that the y-axis is logarithmic. Hexrays is the fastest decompiler, spending between 23 ms and 986 ms with an average of 445 ms per program. RetDec comes in second with 3.5 s to 359 s and 31.5 s on average. This is roughly two orders of magnitude slower than Hexrays. Our implementation is the slowest averaging out at 1455 s, spending between 1.1 s and 27 523 s. Again, two to three orders of magnitude behind RetDec.

Each implementation has a different set of binaries for which decompilation takes particularly long. Hexrays took the longest on `nproc` and `who`, whereas RetDec spend the most time on `dd`, `ls` and `vdir`. Our implementation took a dispropriate amount of time decompiling `md5sum`. The second to last — `cp` — only took 9100 s.

5.2.2. Memory Consumption

Aside from running time, memory consumption is a critical metric for decompilers. Depending on the used data structures keeping all functions, lifted assembly code and associated data flow information in memory can become difficult, even on machines with tens of gigabytes of RAM. RetDec has multiple issues on their public tracker dealing with high memory consumption [3][4][5]. Our implementation is plagued by the same problem. It failed to decompile `sha1sum`, `sha224sum` and `sort` because is exhausted all 16 GiB memory available plus the 4 GiB swap space.

Figure 5.2 shows the peak memory consumption while decompiling the test set. In order to track memory usage of RetDec and our implementation over time we sampled the total number of bytes mapped into the process memory using the `pmap` tool every 500 ms. After decompilation has finished and the process is destroyed we note the largest observed process size as peak memory consumption. For Hexrays

[3] https://github.com/avast-tl/retdec/issues/13
[4] https://github.com/avast-tl/retdec/issues/36
[5] https://github.com/avast-tl/retdec/issues/16

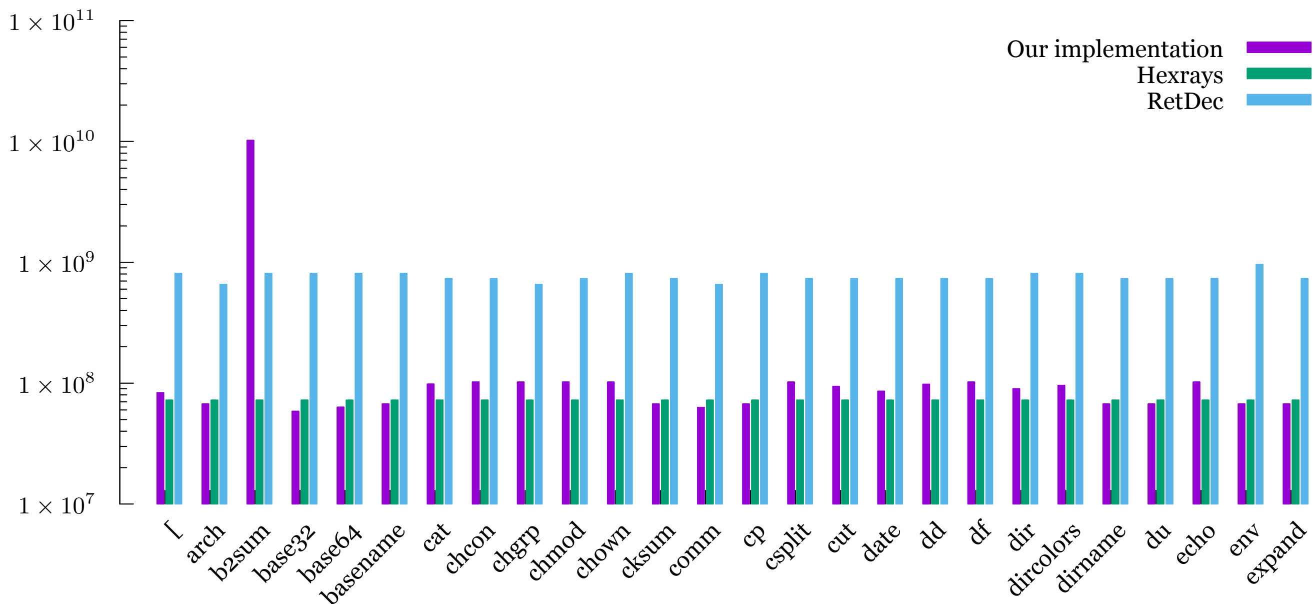


Figure 5.2.: Peak memory consumption while decompiling a subset of `coreutils` in bytes.

we employed the Process Monitor tool from the Sysinternals suite [6]. Process Monitor reports the peak size of memory mapped into the process address space (the Working Set in Windows parlance).

In terms of memory usage the situation is less bleak for our implementation than in terms of running time. The y-axis is again logarithmic and shows the number of bytes consumed. Hexrays allocates between 74 MiB and 75 MiB, our implementation between 56 MiB and 97 MiB with one runaway case of `b2sum` consuming 9.52 GiB. Our implementation also consumed more than the available 20 GiB while trying to decompile `sha1sum`, `sha224sum` and `sort`. As mentioned before, RetDec has particular high memory requirements, consuming one order of magnitude more memory than the others. RetDec consumes between 627 MiB and 916 MiB and was able to decompile all the test set without exhausting system memory.

5.2.3. Completeness of Disassembly

The prerequisite for decompiling successful is finding and disassembling all functions in the binary. As we already explained in Section 2.1, deciding whether a byte in the binary is code can be reduced to the halting problem. Nevertheless, approximating reachable addresses and functions is possible.

To measure how much all code the decompilers find we use the debugging informations present. They list the majority of functions in the binary and addresses they occupy. Not all executable bytes have debugging informations, for example inlined code and functions that are inserted by the compiler do not have this data. In figure 5.3, we list the number of functions the decompiler were not able to find it the test set. The results are similar for all three, except our implementation found significantly fewer functions in `chgrp`, `chroot` and a few other binaries. It found more

[6] `www.sysinternals.com`

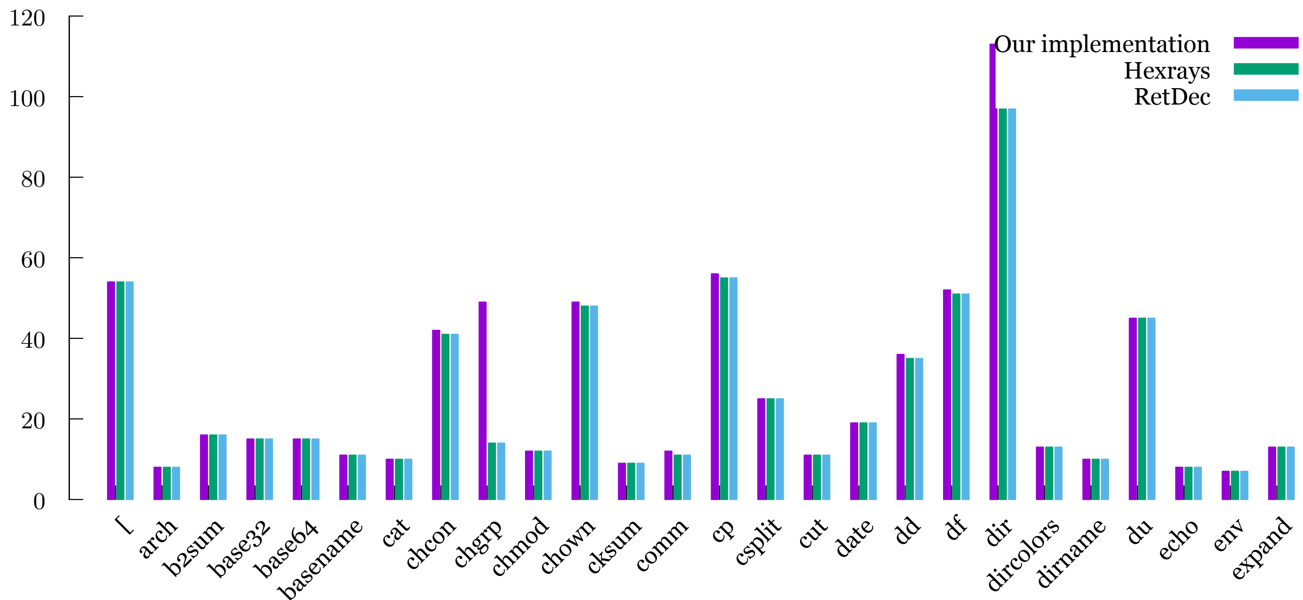


Figure 5.3.: Number of functions not found by the decompilers.

functions in `cut`, `env` and `dirname`. Also, RetDec finds nearly the same amount of functions in all binaries which shows that, with a lot of work, FOSS disassembler can approach the quality of IDA Pro.

5.2.4. Compactness of Decompilation

Looking at the decompiled functions, one simple metric for the readability and thus quality of the decompiled function is their size. We expect the decompiled function to have a low number of lines.

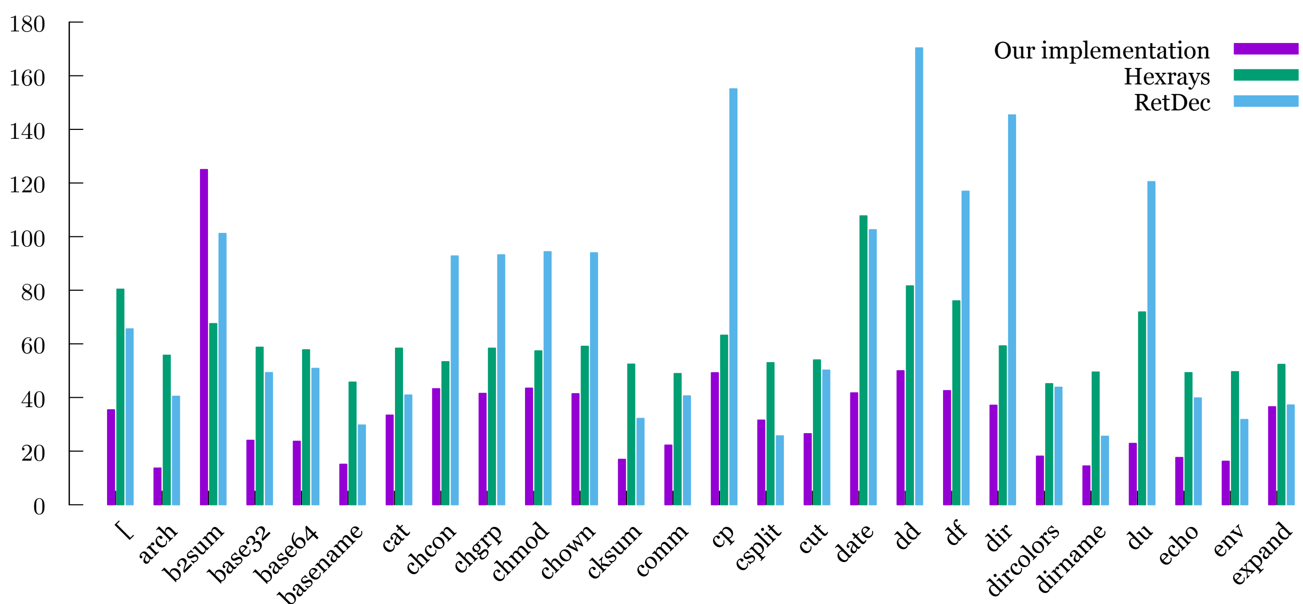


Figure 5.4.: Average lines of code in the decompiled function.

Figure 5.4 shows the average lines of code (LOC) of a function across all three decompilers. Our implementation has the lowest LOC count in most binaries in the test set. Hexrays comes in second. RetDec nearly always produces the largest functions. While the LOC count can be use as a measure for readability, it should be taken with a grain of salt. Hexrays formats functions using Allman style, putting opening and closing braces into their own line. This increases the number of lines linear with the number of control flow statements in the function. Our implementation as well as RetDec formats using the more common Linux C style that pus the opening brace on the same line as the control flow statement.

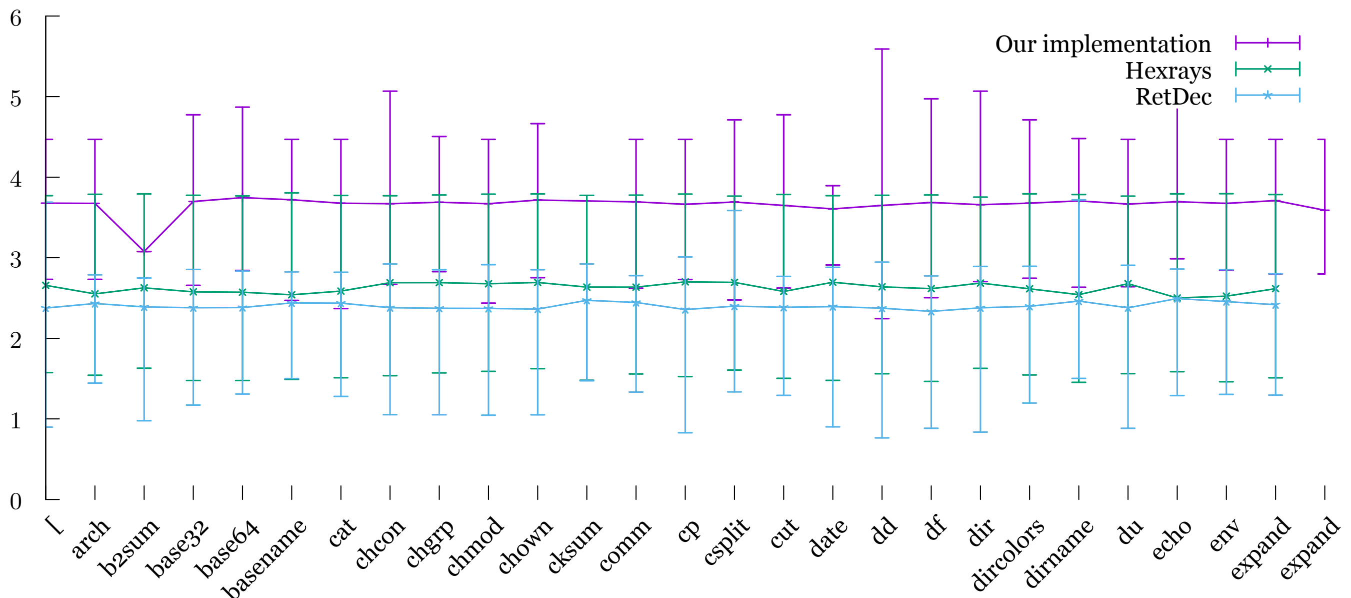


Figure 5.5.: Final readability scores of all three implementations. Lower is better.

Figure 5.5 shows the average readability scores of each binary in the test set. The error bars mark the variance (worst vs. best score). Unsurprisingly the more mature decompiler RetDec and Hexrays beat our implementation in terms of readability. Still, the average difference is ≈ 1 point which is not much considering Hexrays has been in development since 2007 and RetDec since 2011. Interesting is the huge variance of the score across implementations. Out guess is that this reflects that variance in decompiled functions length. Larger functions tend to have worse readability scores as they often contain decompilation failures (c.f. section 5.5).

All previous metrics treated our implementation as a black box. To gain more insights we will now dive deeper and examine the two important components in isolation: the language model and the transformation rules.

5.3. Language Model Performance

In Section 5.2.1 we saw that our implementation takes two to three orders of magnitude longer than the other two decompilers. To learn more about why our approach takes so long we conduct a more fine-grained performance analysis by measuring time spent in the language model. Figure 5.6 shows a precise CPU profile of our decompiler. The profile is visualized as a *flame graph* [27]. The x-axis represents time

and each bar a function. A longer bar means the program spends more time inside that function. A set of bars on top of another bar further details how that function spends its time. Our flame graph shows only functions that contribute more than 0.01 ms to the running time. Also, bar widths are not to scale to improve readability of the graph.

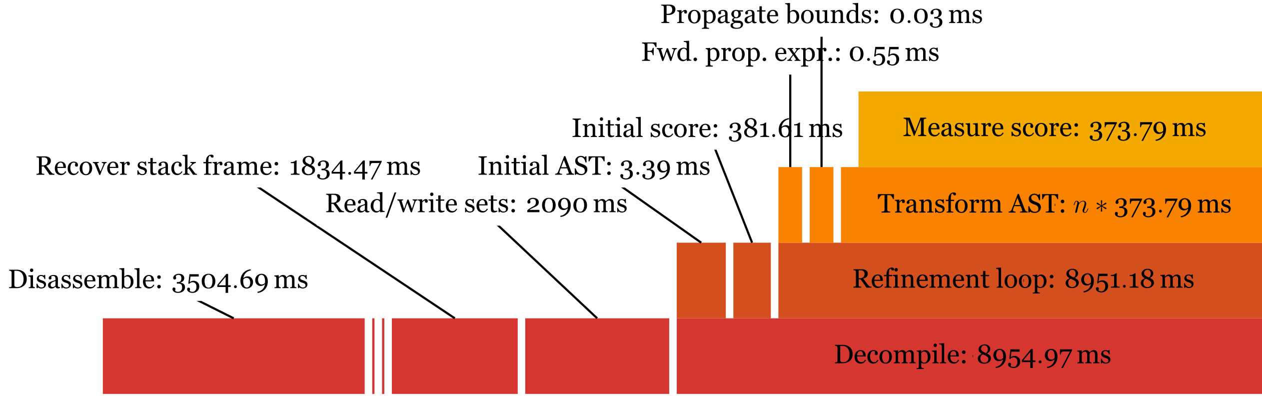


Figure 5.6.: Flame graph of the critical function in out decompiler. Bars higher up give a finer grained profiling of the bar below. All refinement loop iterations are folded.

The CPU profile makes one downside of our approach apparent. Time spent measuring the readability using an RNN is massive. The decompiler spends an average of 374 ms waiting for the measure server. This happens every iteration of the refinement loop. As comparison, selecting and applying an AST transformation takes 0.2 ms to 0.8 ms. In order to be viable for real-world use this needs to be faster. The obvious solution is to use a faster RNN implementation. For this thesis we used a CPU-only training. Using GPU can speed this up by up to one order of magnitude [9]. Also, due to the design of our RREIL dialect decompiling a function is embarrassingly parallel. Read/write sets of called functions and upwards exposed variables are encoded into the immediate language. This way, no synchronization between different functions needs to take place. This means that after disassembly and data flow analysis are complete, decompilation can be distributed across multiple CPU cores and multiple GPUs. Using Amdahl's Law [8] that states that the potential speed-up is one over the non-parallelizable potion task plus the quotient of the parallelizable potion and the available cores:

$$\text{speed} - \text{up} = \frac{1}{(1 - t_p) + \frac{t_p}{n}}.$$

Here t_p is the parallelizable potion, i.e. decompilation while $1 - t_p$ is the non-parallelizable potion, i.e. disassembly and data-flow analysis. The number of CPU cores is represented by n. Using the number from above we can derive,

$$\text{speed} - \text{up} = \frac{1}{0.453 + \frac{0.546}{n}}$$

To get the maximum possible speed-up we compute the limit for n approaching infinity:

$$\lim_{n\to\infty} \frac{1}{0.453 + \frac{0.546}{n}} = \frac{1}{0.453} \approx 2.2.$$

Limiting this theoretical speed-up of 2.2 to a mere 2 gives

$$\left\lceil \frac{2t_p}{1 - 2(1 - t_p)} \right\rceil = \lceil 11.61 \rceil = 12 \text{ CPU cores}.$$

Which is surprisingly reasonable, especially compared to a speed-up of 2.2 which needs more than 300 cores. All numbers above assume that the disassembly and data flow analysis cannot be improved. Which is unlikely considering both RetDec and Hexrays disassemble faster. It also ignores that the disassembly process can be parallelized at least partly.

All in all this shows that the performance problems of our approached can be fixed by parallelizing decompilation using commodity hardware.

5.4. Transformation Rules

Next we will take a deeper look into the distribution of applicable transformations as well as how often they improved readability. For this, we modified our decompiler to record for each iteration of the refinement loop what transformations can be applied as well as which of the selected transforms are found to improve readability. We then run the decompiler again on two applications from our test set `cp` and `ls`. The results can be seen in Table 5.7. Each row lists the data for one transformation, how often it was applicable and how often, after getting selected, it improved readability.

By far the most common transform is changing the number base from decimal to hexadecimal. This is not surprising as the default base for all integer literals is decimal. What is more interesting is that the reverse operation of switching the number base back to decimal is not only applied multiple times but was also found to improve readability. This means that the refinement loop first changed the number base from the default 10 to 16 and, at a later stage concluded that turning it back improved or at least did not decrease readability too much. The fact that the ration between the original and inverse transforms is $\approx 4.4 : 1$ shows that switching number bases is not useless in terms of readability.

The next outlier is the transformation combining a relational condition like $<$ and an equality condition to a single operation. This never applies in any of our test binaries. While we were able to construct RREIL code that fits the patterns, the AMD64 lifter does not produce these when run on GCC generated binaries. The relative popularity of the other transforms roughly proportional to its complexity, less complex transforms being more common.

When computing the success rate of a particular transform we have to take into account the possibility of it being tried in the first place is not 100%. The average number of available transforms per iteration is approximately 300. So, the probability of a transform being selected is $\frac{1}{300} \approx 0.3\%$. Using the decimal to hexadecimal transform as an example:

$$\#\text{Available} = 1\,831\,800.$$

Using the 0.3% probability for being selected we get

Transform	cp #Available	cp #Improved	ls #Available	ls #Improved	$P(S)$
$\mathtt{a}\,\&\,\mathtt{a} \rightarrow \mathtt{a}$	148 111	96	39 941	48	25%
$\mathtt{a} == \mathtt{0} \rightarrow !\mathtt{a}$	455 837	192	150 299	106	16%
$\mathtt{a} - \mathtt{x} \circ \mathtt{0} \rightarrow \mathtt{a} \circ \mathtt{x}$	12 000	3	4 632	1	8%
$\mathtt{a} == \mathtt{a} \Vert \mathtt{a} \circ \mathtt{0} \rightarrow \mathtt{a} \circ\ = \mathtt{0}$	0	0	0	0	N/A
$\mathtt{a}_{10} \rightarrow \mathtt{a}_{16}$	1 904 923	1 144	733 178	544	21%
$\mathtt{a}_{16} \rightarrow \mathtt{a}_{10}$	25 775	207	12 041	95	100%
$\mathtt{a} * \mathtt{x} \rightarrow \mathtt{a}/\mathtt{x}^{-1}$	42	0	0	0	0%
$\mathtt{a} << \mathtt{x} \rightarrow \mathtt{a} * 2^{\mathtt{x}}$	7 549	7	9 485	5	23%
$!(\mathtt{a} \diamond \mathtt{x}) \rightarrow \mathtt{a} \bar{\diamond} \mathtt{x}$	1 751	15	721	9	100%
$\mathtt{if}(\mathtt{true})\ X \rightarrow X$	1 072	2	323	1	71%
Swap if/else branch	31 733	128	12 510	58	100%
Convert if cascade to switch	0	0	0	0	N/A

Figure 5.7.: Popularity of AST transformation rules.

$$\#\text{Selected} = 1\,831\,800 * 0.003 \approx 6\,105$$

which results in a success probability of

$$P(S) = \frac{\#\text{Improved}}{\#\text{Selected}}$$
$$P(S) = \frac{1\,303}{6\,105}$$
$$P(S) = 21.3\%$$

The column named $P(S)$ in Table 5.7 lists the combined success probability for each transform in percent for both test binaries.

5.5. Decompilation Quality

Finally, after evaluating our decompiler by looking at various metrics we also sampled some generated C files to find typical failures or difficult to read expressions. We will compare them to the problems we found in RetDec and Hexrays to get a feel for the different strengths and weaknesses of the three approaches. Due to the bulk of the code we were not able to go through all C code we decompiled. For illustration, we picked the worst case for each class of failure.

5.5.1. Our Implementation

First, we looked at our implementation. The first common failure was lack of redundancy elimination in expressions and conditions. Redundant sub expressions result from the AST generation algorithm we employ as well as from our rather aggressive forward expression propagation. While we have transformation rules that remove them, they often fail to catch all redundancies. This is not surprising when looking

at the relative number of applicable transforms in table 5.7. As the probability of a transform being selected is equal for all, those that are more common are selected more often. Figure 5.8 shows a typical if-statement with redundant check for equality as well as a logical AND with itself. As we will see later, redundant or overly complex conditions are a common failure across decompilers.

```
if( RCX & RCX == 0 || RCX & RCX == 0 && DL & 1 == 0 ) {
  ...
}
```

Figure 5.8.: Redundant redundancy in an if-condition. A typical decompilation failure.

The second typical problem in decompiled code are overly complex conditions. These are related to redundancy but not entirely due to them. The root cause is again the AST generation algorithm that sometimes fails to split path conditions into simple expressions. While we employ techniques like exit node inlining to prevent path conditions from becoming too large, this does not always work. The result can be seen in figure 5.9. Due to the way our AST generation works this problem is unique to our implementation. To fix this problem we need to improve the AST generation. See section 6.1.3 for how this could look like.

One frequent problem not directly linked to the decompiler is our implementation's lack of understanding of register aliases. AMD64 has multiple names for the same (partial) register. The "A" register can be called `RAX`,`EAX`, `AX`, `AL` and `AH`. Each represent parts of the same bits in the register file e.g. `EAX` are the lower 32 bits of the 64 bit large `RAX` register. These aliases exist largely for backwards compatibility with Intel x86, both 32 bits 686 and 16 bits 386 processors. (R)REIL has no understanding of register aliases so the lifter tried every alias as a separate register. Figure 5.10 shows the translation for a simple AMD64 `mov` instruction.

The problem with this non-way of treating register aliases is that our decompiler that processes only RREIL instructions over approximates read/write sets and sometimes misinterprets data flow. When a function reads e.g. the `RAX` and the `EAX` register this will create two entries in the read-set of the function and calls to that function will be decompiled like the one shown in figure 5.11.

Note that the call gets `ESI` and `RSI` as arguments despite them being the same register. Another symptom of this lack of data flow information are stray Φ nodes. Their purpose is to encode the fact that data flow edges join at points where control flow joins, e.g. if a variable x is written in both branches of an if/else-statement both values are possible for all uses of x after the conditional. Φ nodes "tie" both uses together, yielding a combined value. These do not have any semantic at run time and thus are removed by forward expression propagation. Sometimes the dead code elimination removes some but not all aliases of a single register and prevents the Φ node from being removed. This causes Φ nodes from being emitted into the C code (c.f. figure 5.12). An easy fix for this would be to have a decompiler pass that merges all different aliases of the same register. Sadly, this breaks the nice property of out implementation that the decompiler does not need to know about details of the source machine. A less ad-hoc solution would be to allow for register aliases expressed directly in RREIL.

```
if( *(RDI) - 5 == 0 == 0 & *(RDI) - 5 == *(RDI) & CF |
    *(RDI) < *(RDI) - 5 == 0 && 1 & 54 == 0 &&
    4294967295 & 4294967295 < 0 || v588 - 0 == 0 &&
    v588 - 0 == 0 == 0 & v588 - 0 < 0 == v588 ^ 0 &
    0 ^ v588 - 0 ^ 18446744073709551615 && 4294967295 &
    4294967295 < 0 || *(RDI) - 5 == 0 == 0 & *(RDI) -
    5 == *(RDI) & CF | *(RDI) < *(RDI) - 5 == 0 && 1 &
    54 == 0 && AL & 8 == 0 && *(RDI + 4) & *(RDI + 4)
    < 0 && 4294967295 & 4294967295 < 0 || *(RDI) - 5
    == 0 == 0 & *(RDI) - 5 == *(RDI) & CF | *(RDI) <
    *(RDI) - 5 == 0 && 1 & 54 == 0 && AL & 8 == 0 &&
    *(RDI) & *(RDI) == 0 && v590 & v590 < 0 &&
    4294967295 & 4294967295 < 0 || *(RDI) - 5 == 0 == 0 &
    *(RDI) - 5 == *(RDI) & CF | *(RDI) < *(RDI) -
    5 == 0 && 1 & 54 == 0 && AL & 8 == 0 && *(RDI) &
    *(RDI) == 0 && v590 & v590 < 0 && *(v589) - 13 == 0 &&
    v589 - 116 == 0 && 4294967295 & 4294967295 < 0 ||
    *(RDI) - 5 == 0 == 0 & *(RDI) - 5 == *(RDI) & CF |
    *(RDI) < *(RDI) - 5 == 0 && 1 & 54 == 0 && AL & 8 ==
    0 && *(RDI) & *(RDI) == 0 && v590 & v590 < 0 &&
    *(v589) - 13 == 0 && v588 - 0 == 0 && 4294967295 &
    4294967295 < 0 || *(RDI) - 5 == 0 == 0 & *(RDI) -
    5 == *(RDI) & CF | *(RDI) < *(RDI) - 5 == 0 &&
    1 & 54 == 0 && AL & 8 == 0 && *(RDI) & *(RDI) == 0
    && v590 & v590 < 0 && *(v589) - 13 == 0 && v589
    - 116 == 0 && v588 - 0 == 0 && 4294967295 & 4294967295
    < 0 || *(RDI) - 5 == 0 == 0 & *(RDI) - 5 == *(RDI)
    & CF | *(RDI) < *(RDI) - 5 == 0 && 1 & 54 == 0 && AL
    & 8 == 0 && *(RDI + 4) & *(RDI + 4) < 0 && v588 - 0
    == 0 && 4294967295 & 4294967295 < 0 ) {
```

Figure 5.9.: Example of an overly complex `if`-condition.

```
move val:64, RDI:64
move BL:8, val:8
move BH:8, val:8/8
move BX:16, val:16
move EBX:32, val:32
move RBX:64, val:64
```

Figure 5.10.: RREIL code for `mov RBX, RDI`.

```
v582 = func_0x11a50(RBX, R14, R13D, RAX, 0, RSI,
                    0, 243, ESI, 114776, RDX,
                    R13, RCX);
```

Figure 5.11.: Example of an over approximated read/write set. Both `RSI` and `ESI` are listed despite them being the same register.

```
EBX:8 = PHI(EBX:6, EBX:7)
```

Figure 5.12.: Example of a stray Φ node.

To end on a positive note, our implementation correctly handles non-returning functions. As the analysis deciding whether a function returns is interprocedural [7] it is more time-consuming and cannot be easily parallelized. Figure 5.13 shows that the expended time is worth it. The `__stack_chk_fail@plt` function terminates the program after a modification of the stack canary [19] has been detected (the enclosing if-condition). The decompiler correctly infers that `__stack_chk_fail@plt` never returns and thus does not need a `return` statement.

```
if( aloc11837 ^ *(40) == 0 ) {
  return 0;
} else {
  __stack_chk_fail@plt()
}
```

Figure 5.13.: Example of a correctly handled non-returning function.

5.5.2. RetDec

Next we looked at the output of RetDec to see whether it has similar of different failures. The first problem we found is something our implementation struggles with, too: redundant or nonsensical expression. Figure shows an example of that. The if condition depends on the XOR of a value with itself being something other than zero. This is never the case and the positive branch will never be taken. As RetDec does not iteratively refine the AST but employs a set of rewrite rules it applies to exhaustion, this suggests that the decompiler does not handle this particular axiom. Also the fact that the code in the positive branch is obviously dead suggests a bug in RetDec or GCC.

```
int32_t v13 = *(int32_t *)20
            ^ *(int32_t *)20;
g3 = v13;
if (v13 != 0) {
  uint32_t v6 = (int32_t)a3 \% 32;
```

(a) Example of superfluous conditional statement.

```
if( *(RCX) - RBP == *(RCX) & CF
  | *(RCX) < *(RCX) - RBP ) {
```

(b) Example of an overly complex condition produced by RetDec.

Similar to our implementation RetDec is also fond of redundant checks in conditions. Figure 5.14b shows an example. In both parts for the logical disjunction the value of `*(RCX)` does not contribute to the outcome.

A thing RetDec does worse than our implementation are elimination of `goto` statements. While our AST generation code omits them completely, RetDec sometimes includes these. Worse, they also often happen to jump upwards, back to previously executed part of a function, something that can be modelled in an AST using a loop construct. Figure 5.15 came from that same assembly code as figure 5.13 only that RetDec does not handle the leaf call to `__stack_chk_fail@plt` correctly. GCC replaced calls to non-returning functions and functions that do not call other functions

[7] A function returns if it has an exit node and all called functions return.

```
// 0x10a32
g1 = result;
if ((*(int32_t *)20 ^ v12) != 0) {
    goto lab_0x121f8_4;
}
```

Figure 5.15.: Example of a stray goto-statement left by RetDec.

(leaf-calls) with jump instructions. Without special analysis steps, this causes goto statement like that.

```
if ( dword_24028 )
{
LABEL_141:
  dword_24020 = 4;
  goto LABEL_110;
}
```

(a) Example of a stray goto-statement left by Hexrays.

```
void sub_3DF0()
{
  ;
}
```

(b) Example of a complete failure to decompile by Hexrays.

5.5.3. Hexrays

Finally, we took a look at Hexrays. It suffers from the same problem of goto heavy ASTs as RetDec. Figure 5.16a shows a particular bad example where Hexrays generates code that has goto and labels inside an if condition. A quick search through all decompiled C code reveals that Hexrays emits around half the number of goto statements (24539) than RetDec does (51025).

Another rather surprising failure mode of Hexrays is that is sometimes completely fails to decompile a function and simply returns an empty function definition without return statement (c.f. figure 5.16b).

5.6. Identifier Selection

When implementing the language model directed identifier selection we found that using the last mention of the identifier in the source code as hint for the RNN greatly improved the quality of the output. So instead of querying the language model every time a new identifier is needed, we used a simple schema to generate one and used it when emitting source code. The source code is generated up to the last use of the identifier. This is sent to the RNN for identifier selection. The preliminary identifier is than renamed to the one predicted by the RNN. The identifier generated by the RNN are the same one would expect to see in real C code, but some are overly long consisting of multiple words. Figure5.17 list some of the generated identifiers.

priv	KERND
imx3040	index
kuidsize	data
sprivates	USB89173

Figure 5.17.: Sample of identifiers generated by the language model.

5.7. Summary

In this chapter we evaluated the iterative refinement algorithm against two existing decompilers Hexrays and RetDec. As test set, we used the `coreutils` binaries provided by Debian. We first looked at runtime performance: CPU and memory requirements. Our implementation used less memory than RetDec, but spend around two orders of magnitude more time decompiling the same test binaries. We measure the complete disassembly, static analysis and decompilation process. To understand better why our implementation is so much slower, we ran a detailed profiling of our program. This revealed that we spend half our time waiting for the neural network. Completeness of the disassembly is comparable to RetDec and Hexrays.

To evaluate decompilation quality we compared number of lines and final readability scores of all decompiled functions. Our implementation often generated the least amount of code.

Manual inspection of a random sample of decompiled functions showed that typical failures of our implementation were different from that of RetDec and Hexrays. We had the most problems with broken data flow, whereas RetDec and Hexrays tend to produce a lot of `goto` statements. Both our implementation and RetDec produce too complex `if`-conditions.

6. Conclusion

In this thesis we developed a decompiler for AMD64 machine code. After disassembly, we lift all instructions to RREIL code that is converted into static single assignment form. We use Value Set Analysis to infer data flow from registers into the stack and back. To summarize the effects of function calls we compute the set of registers read and written by a function using interprocedural data flow analysis. Then we create an initial abstract syntax tree from the control flow graph of each function using both structural analysis and condition based methods. We use a recurrent neural network trained on the Linux kernel's C source code to measure the readability of the generated AST. We apply semantic-preserving transformations on the AST that may improve its readablity. After each transformation, we use the neural network to generate a readability score. If the score improved we keep the transformation, otherwise we select another one. We iterate this until we stop making progress. Finally, we infer variable types using basic unification.

We evaluated this approach against a common commercial as well as an open-source decompiler. Our implementation is slower, largely due to the performance impact of repeatedly calling the RNN. It beats the open source decompiler in terms of memory usage. In terms of completeness our implementation is on pair with both the commercial and open-source tool. A manual review of decompiled functions shows that while the code reassembles C it still contains redundancies and overly complex expressions. This is partly due to the random selection process of the transformations. The number of transforms is so large that trying all of them is unfeasible, so we sometimes miss transforms that improve readability. The second problem is the RNNs sometimes too limited understanding of C. We opted for a "deep" network, i.e. one than has a high number of hidden layer. This limits the length of the sequence of previous characters it "remembers". This coupled with the fact that we feed C code directly, i.e. a low entropy representation, seems to cause wrong readability scores. This happens especially towards the end of decompiling a function when changes become more subtle. Tackling the performance problem may be possible by parellelizing decompilation across multiple machines.

Despite these problems we believe that the idea of iterative refinement of an AST using RNNs is feasible in practice. Maybe in a less extreme way as done for this thesis. A hybrid approach could start with some "obvious" transforms and only utilize the RNN for transforms that may or may not improve readability. Another way to speed up the decompilation without abandoning the neural network is to only use the RNN to evaluate the usefulness of transforms in general. Before an AST transformation is added to the decompiler it would be run on a test set, using the RNN to evaluate when and under which conditions it improves readability. This knowledge could then be cast into classic algorithms. For example the transform for turning multiplications into divisions (c.f. figure 4.14) only makes sense for certain integers. An offline run with that transform could note all integers for which the readability improved. Using stochastic methods the range of encodable integers can be split into those where the

transform applies and likely improves readability and those where is does not. This knowledge is easy to encode into standard algorithms. This could also be used to eliminate transforms that never or always improve readability. The RNN used for this offline transform evaluation could also be more complex as runtime performance is not important in this application.

6.1. Further Directions

In this final section we will present further extensions to our RNN based decompilation technique and to machine code decompilation in general. These either extend the work of this thesis or address shortcomings of the state-of-the-art in decompilation.

6.1.1. Algebraic Subtyping for Machine Code

Due to time constraints we implemented only extremely limited type inference in our decompiler. Still, the state-of-the-art type inference methods we talked about briefly in Section 2.5 can be improved. Recent advances in type systems may allow applying subtyping to machine code without inducing large implementation- or time complexity. In 2016, Dolan and Mycroft [23] present a novel way constructing type systems that support subtyping. Their core insight is that the lattice of types needs to be distributive. This means that $\wedge$ (meet operation) distributes over $\vee$ (join operation):

$$a \wedge (b \vee c) \equiv (a \wedge b) \vee (a \wedge c)$$

Second, we want type variables to be treated as uninterpreted invariants. This allows the type lattice to be extended without forcing us to proof all properties of our type inference algorithm again. Types with a subtyping relationship produce such a lattice and can be inferred using Dolan et al. *Biunification* algorithm. Binunification extends unification in such a way that is allows handling subtyping. The idea is to treat the left and right sides of an assignment different. The left side will have a positive type τ^+ and the right side a negative type τ^-. Same with function calls: inputs are positive, outputs negative. The polarity of a type is syntactic and thus the biunification algorithm can use this information when inferring types. We restrict all subtyping constraints those of the form

$$\tau^+ \leq \tau^-.$$

When all these requirements are met, biunification can infer principle types in quadratic time in the worst case. Biunification supports polymorphism and equirecursive types that are needed to infer more complex C data structures.

6.1.2. Abstract Syntax Tree Based Machine Learning

One avenue in improving the language model is to use an input encoding into the recurrent neural network other than plain text strings. Without changing the basic working principle of RNNs we need to produce a sequence of symbols for the model.

Instead of generating a character string we could send the AST nodes directly into the network. There exist multiple ways to turn a tree into a sequence of nodes: pre-, post- and level order. This sequence is unique for an AST if the children of a node are ordered. Figure 6.1 gives an example of the possible traversal orders.

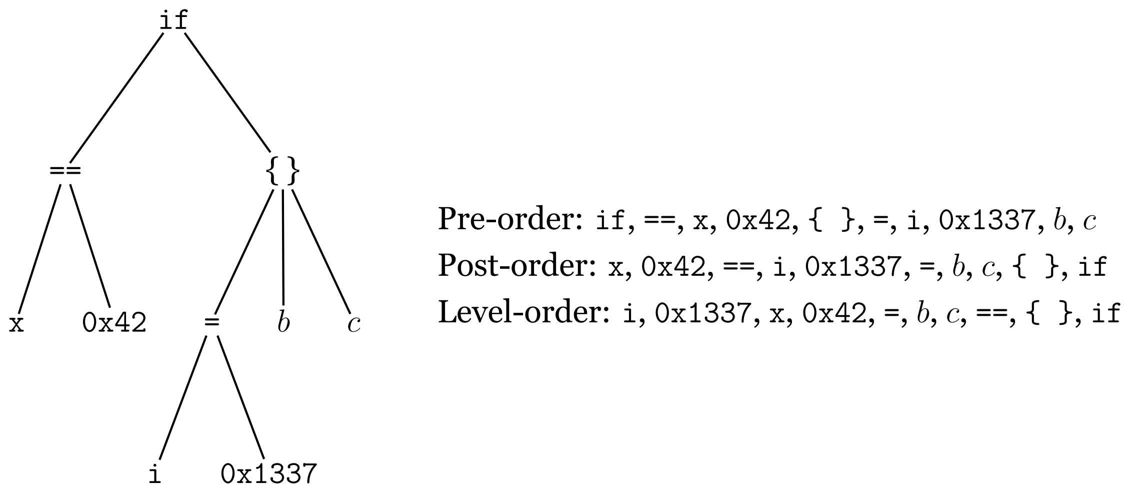


Figure 6.1.: Different AST node sequences resulting from different AST traversal orders.

While this looks similar to just using character sequences, they keep the network from learning some invariants about C. First, multi-character operators like `>=`, `->` and `&&` are represented as a single token. Second, white space that is insignificant (except for separating tokens) is absent from the input. This also could help with performance as white space makes up a large part of C code. Third, it removes parenthesis and braces from the source code and with them all details about correctly matching them, as well as precedence rules of operators. It also omits the special case that allows braces to be left out for single statement bodies. While the first advantage can also be archived by using the tokeniser output, the second and third cannot.

6.1.3. Faster Condition Based Refinement

While condition based refinement for control flow recovery is able to construct `goto`-less AST that is more compact than the purely pattern matched approaches used by other decompilers it is also slow. As mentioned in Section 2.4 this is due to the need to decide whether a logical formula subsumes another. To avoid integrating a SAT solver we used the disjunctive normal form as representation. While this degenerates the subsumption query to a set inclusion, computing the DNF has exponential time complexity. We sidestep this issue by limiting the maximal size of a DNF formula. This has the problem that we may end up with conditions that are opaque for the decompiler and cannot be simplified (Section 5.5). A more promising way could be exploiting the connection between the logic formula and the dominator tree of the control flow graph. The question of logic subsumption can be reformulated as query whether one node's edge dominates another node. If yes, the path condition to the dominator node subsumes the condition of the dominated. This query is linear in the size of the control flow graphs dominator tree.

For example, see Figure 6.2. When constructing an AST from this control flow

graph the algorithm will have to decide at one point whether the path condition of node 4 subsumes that of node 2:

$$\neg B \wedge A \wedge S \implies A \wedge S.$$

Instead, it can look at the dominance tree that tells it n_2 dom n_4. Same with nodes 1 and 4.

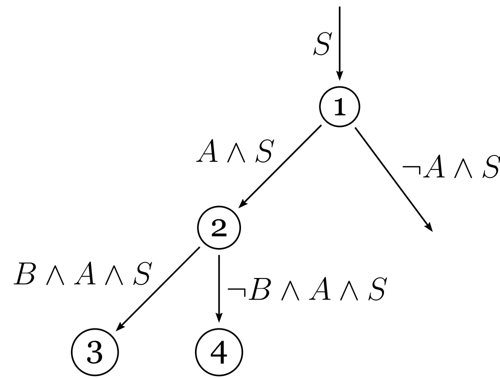


Figure 6.2.: Using dominators to decide path condition subsumption.

Computing the dominator tree is logarithmic [38] in the size of the graph. The second reason to operate on boolean formulas is to decide whether one is the negation of another. This is used to rewrite mutually disjoint `if` statements to `if/else`. Instead of doing this on the DNF where computing the negation has exponential time complexity, formulas can be represented as Binary Decision Diagrams [13]. These represent boolean functions as directed graph structures. While pathological cases exist, most functions have compact BDD. BDD support all elementary operations on booleans including negation. Again, worst-case behaviour is possible but most negated BDD are smaller than exponential. BDD expect all literals, i.e. variables in the formula to be ordered. The order can be arbitrary but must be fixed. Two equivalent boolean functions with equal literal order produce homomorphic BDD. Checking for this homomorphism can be done in linear time. It suffices to check whether vertex and edge sets are equal.

Bibliography

[1] *AMD64 Architecture Programmer's Manual*.

[2] Fcd — fcd. http://zneak.github.io/fcd/.

[3] Hex-Rays Decompiler: Overview.

[4] Snowman. http://derevenets.com/.

[5] *System V Application Binary Interface*.

[6] ANSI/ISO 9899-1990, 1990.

[7] Binnavi: BinNavi is a binary analysis IDE that allows to inspect, navigate, edit and annotate control flow graphs and call graphs of disassembled code, May 2018.

[8] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. page 483. ACM Press, 1967.

[9] Jeremy Appleyard, Tomas Kocisky, and Phil Blunsom. Optimizing Performance of Recurrent Neural Networks on GPUs. *arXiv:1604.01946 [cs]*, April 2016.

[10] Gogul Balakrishnan. *WYSINWYX: What You See Is Not What You Execute*. PhD Thesis, University of Wisconsin–Madison, 2007.

[11] M. Belyaev and V. Tsesko. LLVM Based Static Analysis Tool Using Type and Effect Systems. *Automatic Control and Computer Sciences*, 2012.

[12] Adam R Bryant. Understanding How Reverse Engineers Make Sense of Programs from Assembly Language Representations. page 266.

[13] Randal E Bryant. Graph-Based Algorithms for Boolean Function Manipulation. page 29.

[14] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. page 16.

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

[16] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*, 2011.

[17] Keith Cooper and Linda Torczon. *Engineering: A Compiler*. Morgan Kaufmann, Amsterdam ; Boston, 2 edition edition, February 2011.

[18] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[19] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. page 11.

[20] Balázs Csanád Csáji and Huub Ten Eikelder. *Approximation with Artificial Neural Networks*.

[21] Johannes Dahse and Thorsten Holz. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *In Symposium on Network and Distributed System Security (NDSS*, 2014.

[22] L Damas and R Milner. Principal Type-Schemes for Functional Programs. page 6.

[23] Stephen Dolan and Alan Mycroft. Polymorphism, Subtyping, and Type Inference in MLsub. *POPL*, 2016.

[24] Thomas Dullien and Sebastian Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis. page 7.

[25] Mike Van Emmerik and Trent Waddington. Boomerang: A Native Executable Decompiler.

[26] A. Fokin, E. Derevenetc, A. Chernov, and K. Troshina. SmartDec: Approaching C++ Decompilation. In *18th Working Conference on Reverse Engineering*, pages 347–356, October 2011.

[27] Brendan Gregg. The Flame Graph. *Commun. ACM*, 59(6):48–57, May 2016.

[28] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[29] Josef Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. page 74, 1991.

[30] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[31] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Pearson, Boston, 3 edition edition, July 2006.

[32] Linus Kaellberg. Circular Linear Progressions in SWEET.

[33] J. Kinder and H. Veith. Precise static analysis of untrusted driver binaries. In *Formal Methods in Computer Aided Design*, pages 43–50, October 2010.

[34] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, May 1983.

[35] J. Křoustek, P. Matula, and P. Zemek. *RetDec: An Open-Source Machine-Code Decompiler*. December 2017.

[36] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[37] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled Reverse Engineering of Types in Binary Programs. *NDSS*, 2011.

[38] Thomas Lengauer and Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, January 1979.

[39] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[40] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. page 12.

[41] Matthew Noonan, Alexey Loginov, and David Cok. Polymorphic Type Inference for Machine Code. *arXiv:1603.05495 [cs]*, March 2016.

[42] Andre Pawlowski, Herbert Bos, Elias Athanasopoulos, Thorsten Holz, Chris Ouwehand, Cristiano Giuffrida, Victor van der Veen, and Moritz Contag. MARX: Uncovering Class Hierarchies in C++ Programs. *NDSS*, 2017.

[43] Benjamin C. Pierce. *Types and Programming Languages*. Types and Programming Languages, Cambridge, Mass, January 2002.

[44] radare. Radare decompiler tool based on radeco-lib, April 2018.

[45] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. pages 745–762. IEEE, May 2015.

[46] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 Decompilation Using Semantics Preserving Structural Analysis and Iterative Control-Flow Structuring. *USENIX Security*, 2013.

[47] Alexander Sepp, Bogdan Mihaila, and Axel Simon. Precise Static Analysis of Binaries by Extracting Relational Information. pages 357–366. IEEE, October 2011.

[48] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The Art of War: Offensive Techniques in Binary Analysis. page 20.

[49] Carsten Sinz, Florian Merz, and Stephan Falke. *LLBMC: A Bounded Model Checker for LLVM's Intermediate Representation*. 2012.

[50] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study. *USENIX Security*, 2016.

[51] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. *NDSS*, 2015.

[52] Fabian Yamaguchi. *Pattern-Based Vulnerability Discovery*. PhD Thesis, Georg-August University School of Science, 2015.

Appendices

A. Complete Figures

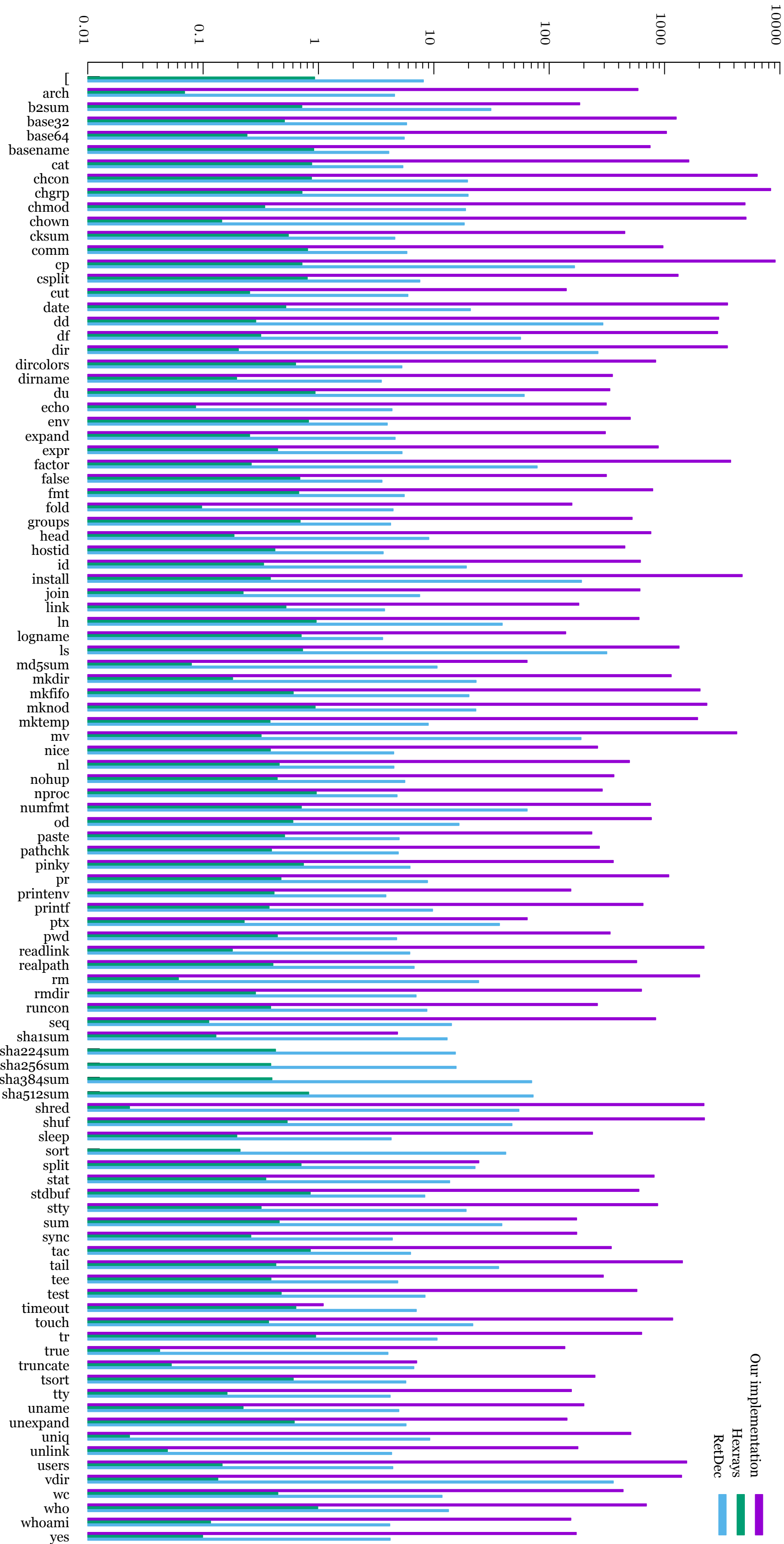


Figure A.1.: Time spent decompiling a subset of `coreutils` in seconds. Complete figure.

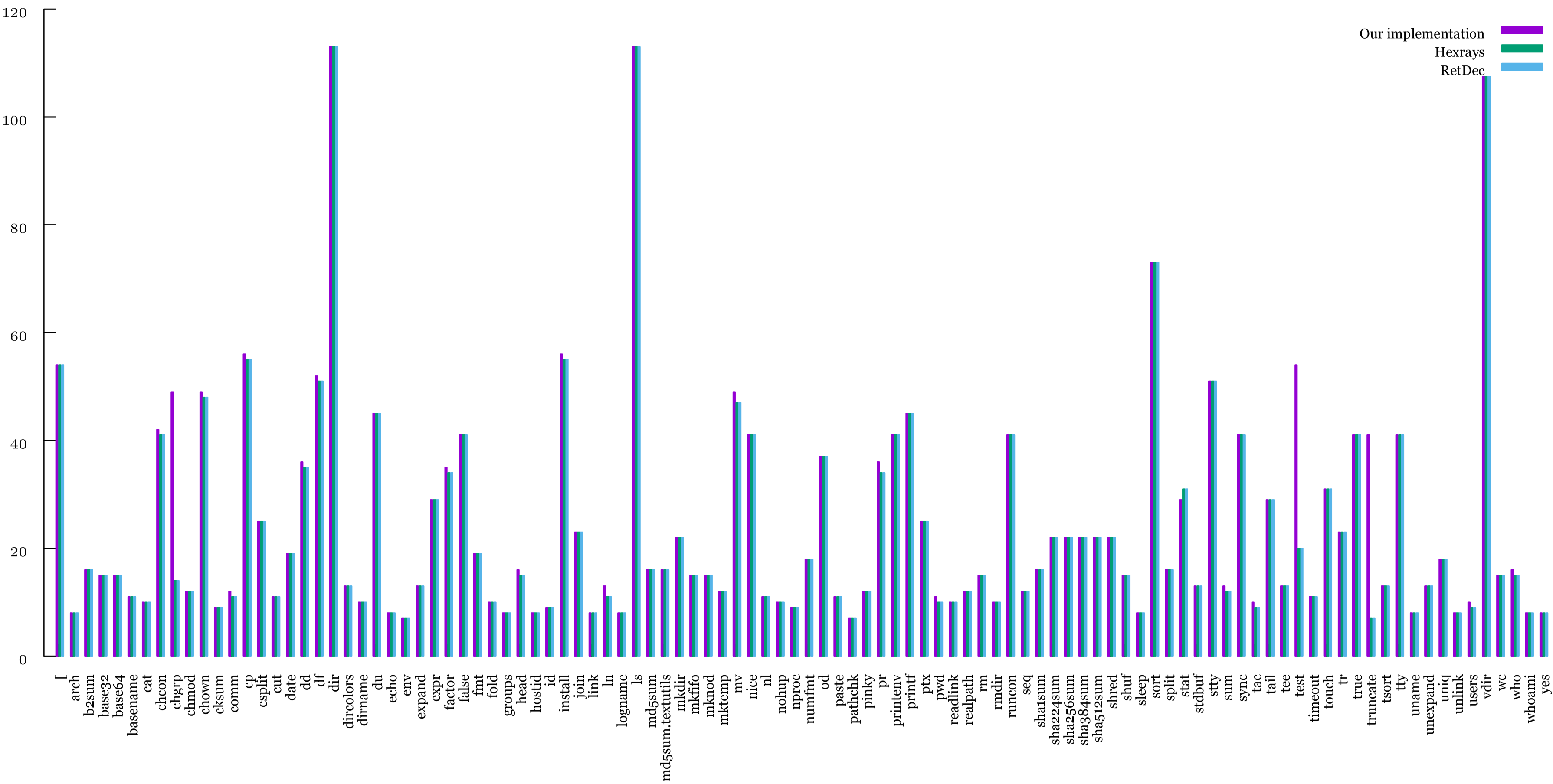


Figure A.2.: Number of functions not found by the decompilers. Complete figure.

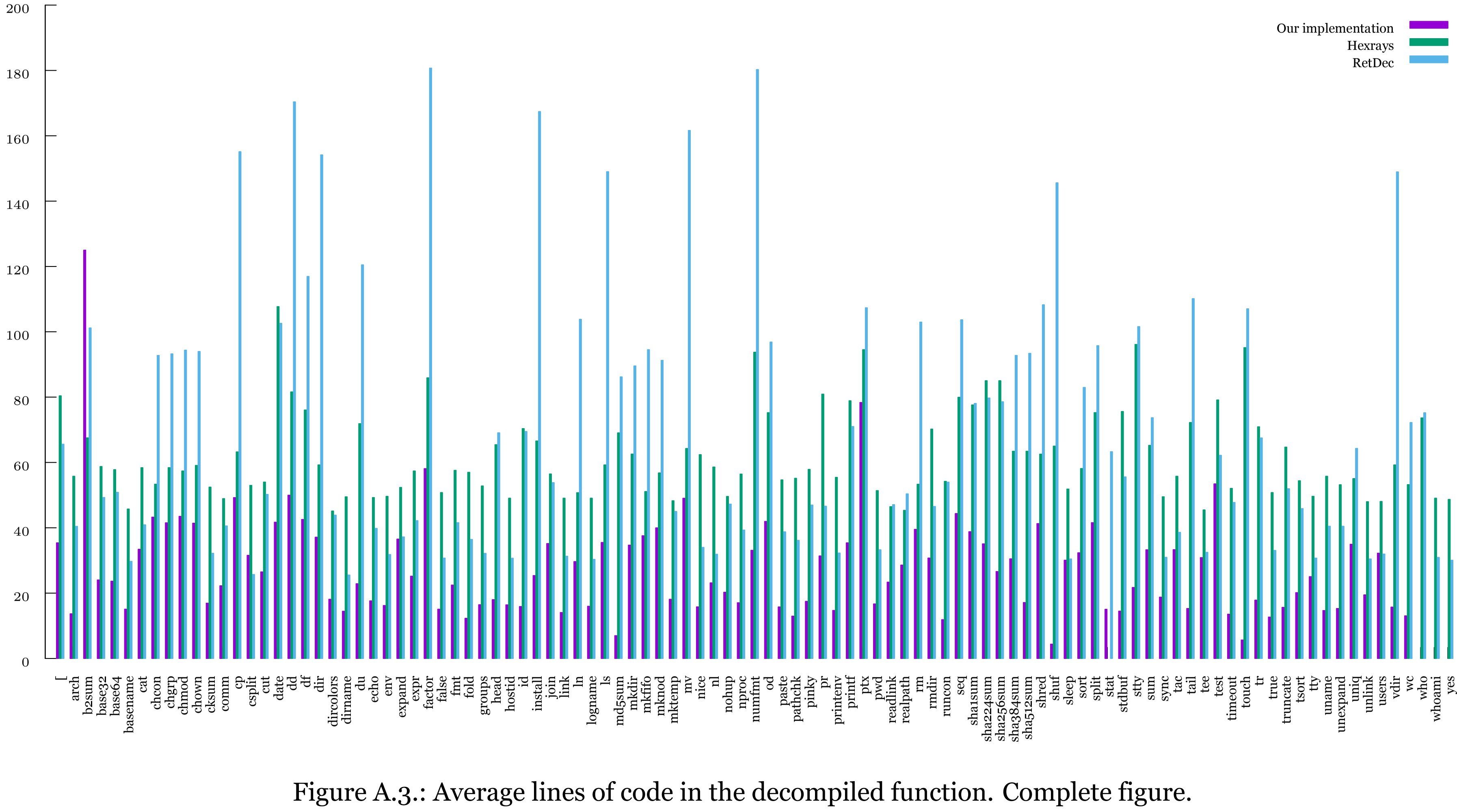


Figure A.3.: Average lines of code in the decompiled function. Complete figure.

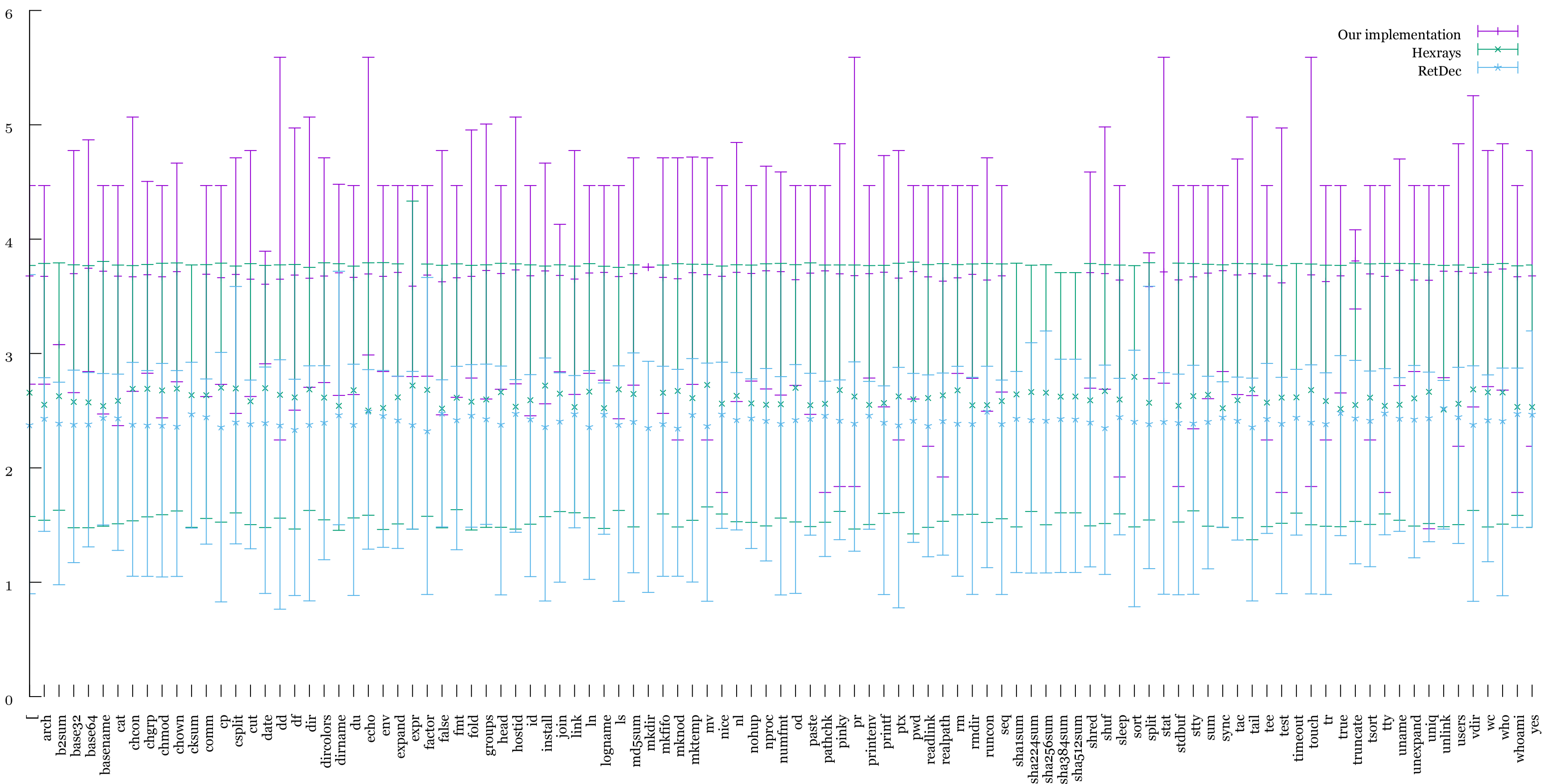


Figure A.4.: Final readability scores of all three implementations. Lower is better. Complete figure.