FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Ein Multi-Architektur Decompiler

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Thomas Wagner, BSc.
Matrikelnummer 001125782

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dr.techn. Edgar R. Weippl
Mitwirkung: Dr.techn. Georg Merzdovnik

Wien, 25. April 2019

Thomas Wagner	Edgar R. Weippl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Multi-architecture Binary Decompilation

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Thomas Wagner, BSc.
Registration Number 001125782

to the Faculty of Informatics

at the TU Wien

Advisor: Dr.techn. Edgar R. Weippl
Assistance: Dr.techn. Georg Merzdovnik

Vienna, 25th April, 2019

Thomas Wagner

Edgar R. Weippl

Erklärung zur Verfassung der Arbeit

Thomas Wagner, BSc.
August Greiml Weg 5

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 25. April 2019

Thomas Wagner

Danksagung

Zuallererst möchte ich mich bei meinem Betreuer für diese Arbeit, Georg Merzdovnik, bedanken. Mit seinen Vorlesungen und dem IT Capture the Flag Team der Technischen Universität Wien, wurde ich in das komplizierte und spannende Thema der IT-Sicherheit eingeführt. Viele angenehme Tage verbrachte ich mit ähnlich gesinnten Leuten während wir uns über die Funktionsweise von Programmen den Kopf zerbrochen haben. Danke für deine Hilfe und Betreuung der letzten Jahre.

Ein großes Danke geht ebenfalls an meine Eltern und Familie, welche mich mein bisheriges Leben unterstützt haben. Ohne ihrer Hilfe, ihrem Zuspruch und auch manchmal ihrer bitter nötigen Maßregelungen wäre mein bisheriger Weg ein steinigerer geworden.

Und zu guter Letzt noch ein spezielles Dankeschön an meinen Onkel, Wolfgang Ibl. Für seine Hilfe diese Arbeit zu korrigieren und mir vor allem bei meinem Schreibstil etwas nachzuhelfen. Auch die künstliche Deadline, welche er mir gesetzt hat, hat mir sehr geholfen mich auf das Abschließen dieser Arbeit zu konzentrieren, um den Prozess nicht unnötigerweise in die Länge zu ziehen.

Acknowledgements

First of all, I would like to thank my academic advisor for this thesis Georg Merzdovnik. With his lectures and the IT-Security Capture the Flag team of the Vienna University of Technology We_0wn_Y0u, I was introduced to the complicated and exciting topic of IT-Security and helped me develop in this area. I have spent a lot of enjoyable days with like-minded people stewing over the inner workings of programs which eventually led me to the topic of this thesis. Thank you for the supervision and help I got from you for the past few years.

I also have to thank my parents and family who supported me my whole life and helped me get me to where I am today. Without their help, encouragement and sometimes much-needed reprimands my journey up until this point would have been a much harder one.

And finally, a special thank-you goes out to my uncle Wolfgang Ibl who meticulously looked over this thesis and helped me improve on my writing style. His setting up of a soft deadline for finishing my studies helped me with gathering the necessary motivation in actually writing and completing the last steps that were needed.

Kurzfassung

Eine wichtige Aufgabe in der Abwehr von digitalen Bedrohungen ist das Analysieren und anschließende Rückentwickeln von Programmen. Es ist essenziell Schadsoftware zu verstehen, um ausgenutzte Schwachstellen in Systemen zu finden, um sich vor möglichen Angriffen zu schützen. Die Dekompilierung ist ein wichtiger Schritt in diesem Prozess, da sie den Code in einer strukturierten und für den Menschen lesbaren Form darstellt.

Unsere Arbeit befasst sich mit der Entwicklung eines Decompiler-Prototyps, welcher Genauigkeit und Korrektheit in den Vordergrund stellt. Die Korrektheit in dem Prozess ist von großer Bedeutung, da sonst entscheidende Teile der Gesamtlogik des Programms, die zum Verständnis seines Verhaltens notwendig sind, beim Dekompilieren verloren gehen könnten.

Unser Decompiler basiert auf einem ähnlichen Prinzip, welches in einer älteren Version von GCC verwendet wurde. Diese ursprüngliche Kompilierungstechnik wurde von uns für den Dekompilierungsprozess adaptiert. Sie basiert auf den folgenden Schritten. Zuerst transformieren wir die Binärdatei in eine Zwischendarstellung in “static single assignment”-Form(SSA-Form). Diese initiale Darstellung ist so detailliert wie möglich, da wir an diesem Punkt noch keine Optimierungen durchführen. Im nächsten Schritt wenden wir “peephole-optimization” und “dead code elimination” an, um die Codegröße zu reduzieren. Darüber hinaus werden speziell entwickelte Regeln verwendet, um bestimmte Muster zu finden. Diese werden durch logisch äquivalente Ausdrücke ersetzt, welche leichter zu verstehen sind als ihre nicht optimierten Gegenstücke. Dadurch erhöhen wir die Lesbarkeit des generierten Codes.

Zusätzlich haben wir uns mit der Rekonstruktion von Funktions-Signaturen befasst. Die Verwendung einer “usage analysis” und das anschließende Propagieren der Informationen über die Aufrufhierarchie hat zu guten Erfolgen bei kleineren Programmen geführt, allerdings nahm die Nützlichkeit rapide ab je größer die Programme werden.

Abstract

Reversing binary programs and extracting their meaning is a crucial task in defending against digital threats. It is important to understand malware and find exploitable vulnerabilities in systems in order to protect against possible attacks. Decompilation is an important step in this process because it presents the code in a structured and human-readable form.

In this thesis, we implemented a decompiler prototype with the specific aim to be as accurate as possible. Correctness is of utmost important because otherwise crucial parts of the programs overall logic which is necessary to understand it's behaviour could be lost in the decompilation process.

Our decompiler is based on a similar approach as was used by a compilation technique in old versions of GCC. It is based on the following steps. First, we transform the binary into a high-level intermediate representation in static single assignment (SSA) form. This representation is as descriptive as possible because at this stage we do not apply any optimizations. In the next step, we apply peephole optimization and dead code elimination to reduce the outputs code size. Additionally, specially crafted rules are used to find certain patterns and replace them with logically equivalent expressions that are easier to understand than their unoptimized counterpart. Thereby we increase the readability of the generated code.

Furthermore, we worked on the reconstruction of function call signatures. We based our approach on a usage analysis and propagated the gathered information over the call hierarchy. While this approach led to decent results for small programs in our evaluation it appears to lose usefulness for larger programs.

Contents

CHAPTER 1

Introduction

Malware and security flaws in software are an ever-present problem and with the continuous digitalization threats posed by them, it will surely increase with time. To combat these threats, it is important to be able to analyze programs and figure out how they work in order to combat malicious actors.

Malware can infect systems in different ways. The most prominent is the exploitation of one or more flaws in an existing system or user errors. The best defences against user errors is security awareness and a good anti-virus. Software flaws are different. They have to be found and removed.

The first way to find flaws is to analyze a vulnerable program. The problem with this approach is that most programs are commercial, which means that source code is not available to the public (Open Source projects excluded). As users, we can therefore not trivially figure out what a program actually does and how it functions in detail. This makes it difficult for a competitor to copy the functionality, but it also makes it difficult to find possible flaws for both benign and malicious actors.

The second way of finding flaws is to analyze malware directly. Most of the time this is even more difficult because malware is often heavily obfuscated to prevent exactly this kind of analyses[1]. Obfuscation for example with randomized/metamorphic binaries[2] also prevents Anti-Virus programs to use signatures[3] to detect them.

To understand a program without executing it we need to transform the binary back into a human-readable form. This is were decompilers come into play. These are programs, which take a binary and convert it into a readable representation of the behaviour. Mostly they convert the program into C or C++ code. This process is called decompilation.

Decompilation is a difficult task as machine code, especially on CISC-architectures is quite complicated and multifaceted. Instructions that look simple do a lot of things

"under the hood". Especially internal CPU-flags are often manipulated as a side effect to the normal results. A decompiler needs to detect and recognize all possible data dependencies and model them, which is not a trivial task. Additionally, the decompiler needs to discard unnecessary pieces of information and display the important data flow for a user to see. For obfuscated code, this is of critical importance because the obvious data-flows might not be relevant to the actual semantics of the machine code. Registers or flags not commonly used can carry information, which a superficial analyzer can miss. This is why a comprehensive and accurate modelling of behaviour is crucial. For visualizing the code simplicity is better. Most high-level languages do a lot of work implicitly that a programmer does not need to be aware of. Stack-management is one of such tasks: return addresses are written/used, registers are saved/restored and local variables are kept there. It is for example not necessary for a programmer to know how a stack is built and what information is stored to write a program. A compiler might even decide to introduce multiple stacks to improve security(e.g. preventing stack overflow from corrupting the return pointer). In any case, a decompiler should be capable of abstracting away those low-level concepts like the stack and presenting it in a simple way to illustrate the data flow. This is not an easy task because CPUs often have no real restriction on what can be done with the instruction set available.

Programs typically follow certain rules set out by conventions. For example, C and C++ programs mostly follow the same calling conventions, the default stack pointer is most of the time used for the stack and so on. Obfuscated binaries are not bound by such rules and are able to "invent" their own conventions and use them as they see fit. Strictly assuming certain conventions is therefore not always useful.

Malware is a big problem to decompile, as their developers do not want to have them reverse-engineered. Therefore it is important for them to make the purpose of the program as difficult to discern as possible. Searching for javascript malware as an example shows that trend quite clearly. Most of them are absolutely unreadable[4]. Having a decompiler to be open source and adaptable is therefore paramount in trying to figure out what a program really does. Let us give an example: Assume we have a malware that has a code section being 500 KB large. 500 KB is quite a lot and with a conservative estimation of 8 bytes per instruction, that means our program consists of over 60.000 instructions. Browsing them manually is time-consuming. Using a decompiler helps a lot, but if the code is obfuscated we might not be able to get much useful information. Let us take a certain obfuscation technique as an example: We replace every jump instruction with a "push targetaddress;return" combination and every call instruction by a "push returnaddress;push targetaddress;return" combination. This does not alter the semantics of the program at all, it only makes it slightly slower and bigger. Under those circumstances, any normal analysis for function bounds will no longer be correct as it will find a massive amount of functions. This reduces readability, but such a pattern can be recognized and an open and extendable decompiler can be made to detect and reverse these obfuscations.

The goal of our work was to implement a decompiler framework which focuses

on accuracy and extensibility. While using this approach here we will document our implementation and the quality of its results.

For accuracy, we tried to represent the dataflow exactly. That means instructions need to be decoded precisely. Any operation that is done and any value calculated and stored in any way needs to be represented in our decompiler. We can not guess what is happening or leave out side-effects that may be used in order to obfuscate control flow. We can not discard information that might be useful at the time or later during processing. Therefore we took a conservative approach in most of the transformations and optimizations used.

With extensibility, we mean two different attributes. Firstly new architectures must be definable in an easy way. To achieve this we created a unified interface for analysing binaries and disassembling functions. Also, we created a way to define instruction sets that allow the modelling of any processor-architecture. We also created a way to interact with our internal representation to arbitrarily change said interpretation. This can be used to implement new ways of analyses or transformations.

To help implement new stages in the decompiler we planned on implementing a scripting interface. This helps immensely in making the decompiler more extensible without recompilation but as of now, that was not added.

The main structure of our decompilation solution is to split up certain tasks and perform them completely independent. The first step is raising the binary program into our intermediate representation. The goal is to model the state of the processor as accurately as possible. Quality of results is not a major concern at this point just correctness. The second step is the optimization step. Here we use multiple techniques to improve iteratively on the previously unoptimized results. Especially our peephole optimization rules were effective in simplifying results. The last step is making the code presentable. It is concerned with displaying the results for users.

Because of the extensive amount of optimizations done by compilers nowadays it is not trivial to reconstruct complicated control and data flows, especially over multiple functions. Our conservative approach does not allow us to make assumptions about function arguments and return values, which in turn makes it nearly impossible to remove unnecessary arguments. Our analysis worked well for simple functions. With rising complexity, we observed a significant drop in discarded values. Especially return values could often not be optimized away when function calls were used inside of loops.

The main focus of our work was in the IR-generation and optimization stage. This meant improving the intermediate representation to be as concise and at the same time as expressive as possible without losing information. As a result improvements to the pseudo-code generation were neglected because without a proper intermediate representation the quality of the pseudo-code generated will suffer either way.

While most research on decompilation focuses on reconstructing data-types and control flow we have determined the biggest issue to be the reconstruction of just function

signatures. While types and control flow can improve the readability of the resulting pseudo code by a lot, function calls with a wrong signature can falsify the result that is displayed. Current solutions mostly rely on heuristics to guess the correct signature, which is not in line with our goal for accuracy. We used an inter-procedural analysis to propagate arguments which are not used through the call hierarchy. This approach worked well for simple problems, but rising call depth meant less potential for optimizations and therefore more arguments which could not be optimized away.

This document contains six parts. First, we talk about the technologies that our work is based upon. This includes disassembler and relevant compilation techniques, which we use. We will then take a look at reverse engineering technologies used in decompilers and analysis tools that currently exist in Chapter 2 and Chapter 3. Chapter 4 goes into a detailed description of our implementation. We will talk about the data structures and algorithms used to create the pseudo code. Chapter 5 shows the results of our decompiler from a few examples. Next, we will evaluate our results in comparison to other decompilers that we could use. Then we will reflect on our implementation, the results that we achieved and possible next steps to improve the decompiler. Finally, we will look at possible works that we might do in the future.

CHAPTER 2

Background

There is already a wide range of techniques that exist to reverse engineer binary programs. In the beginning in it is essential to extract the semantics of the program by analysing the byte-code and extracting the instructions. These instructions are then transformed into a representation ideal for further processing. There are some techniques usually seen in compilers which are also useful in reverse engineering.

2.1 Disassembling

The first step of any static analysis that looks at the functionality of a binary program is disassembling. Conversely to assembling, disassembling takes machine code which is in binary form and extracts the instructions into either a readable form or a representation which can be used for further processing. These instructions are the basic building blocks, that computer programs are built upon. Each instruction tells the processor how to transform its current state to the next.

2.1.1 Textual Instruction Representation

There are two major syntaxes of how to display said instructions:

- AT&T syntax: In this syntax first we write the mnemonic on an instruction then the source-operands and finally the target-operands. Registers are prefixed with a per cent symbol and constant values with a dollar sign. Complex operations such as memory offset calculations are denoted with brackets like this: *segmentation-reg:displacement(base,index,scale)*.
- Intel syntax: The Intel syntax also starts with the mnemonic but source-operands and target-operands are the other way around. The first operand is the destination

and the second is the source. Neither registers nor values are prefixed with a character and complex memory offset calculations are done within a square bracket written out like this: $segmentationreg : [base + index * scale + displacement]$.

When displaying assembly we will mostly use Intel syntax as it does not have the prefix-rules for arguments and the memory offset calculations are clearer as the operation is written like an actual mathematical expression.

2.1.2 Techniques

It is also important to distinguish how disassemblers find sections to disassemble. There are three major techniques, which are used in programs[5].

Linear Sweep

This technique starts at the first byte(or smallest addressable memory block) of an executable memory section and disassembles until it processed the whole executable memory. Functions are typically not detected or handled in any way. This catches most instructions in a binary but has a big downside. Data interlaced with code is not detected and often leads to wrong instructions, when disassembled. Also on systems, which can have instructions of variable length, it is possible to generate polymorphic instructions that a CPU can process, but a linear sweep algorithm has problems with.

Recursive Descending

This kind of disassembler starts at one or more points, where the disassembler can be sure, that a function starts at. This can be for example an entry point, defined in the metadata of the binary. From this point on the disassembler follows paths through jumps, branches and calls. While disassembling it interprets the instructions in a limited way and can, therefore, follow the control flow. If a new function is called the entry point of the function can be used to continue disassembling at that address. This is a powerful technique, as obfuscation techniques like polymorphic instructions can now be circumvented, that a linear sweep disassembler can simply not deal with[6]. The problem with recursive descending disassemblers appears when indirect jumps occur. Indirect jumps are often used for virtual function calls and switch-statements where the case-values are part of a sequence of numbers or close to a sequence. For virtual functions, it is difficult for a purely recursive descending disassembler to figure out all possible targets. Additional analysis of the metadata or the data flow needs to be performed before targets can be inferred. Indirect jumps from switch statements are a bit easier, as the jump-targets are commonly in a table at a certain static address, which is easier to deduce because the offset is most of the time hardcoded in the program[7].

Hybrid Approaches

It is reasonable to combine the advantages of both techniques to optimize function detection[8]. Starting at the beginning of an executable section or a previously known entry point a program may use a recursive descending disassembler to find as many functions as possible. In case no more functions are found it can then find sections that were not yet disassembled and assume such holes are undiscovered functions. The beginning of the not yet analysed block can then be used to start another recursive descending pass. This can be done until all executable memory has been completely converted into assembly code.

2.2 Compilation Techniques

There are some representations and transformations used mostly in compilers, which help in constructing analysers or decompilers. We use a small amount of those ideas in our own implementation. In the following section, we go over these techniques and describe their general use-cases.

2.2.1 Instruction Representation and Intermediate Representation

As the name suggests an intermediate representation(IR) is a way to represent a program during one or multiple steps while performing different operations on the code. The concept is primarily used in the field of compilers that convert a programming language into their intermediate representation in order to perform optimizations, transformations and finally instruction selection. Specialized IRs can offer many advantages over an abstract syntax tree, which is an IR to represent program code directly. IRs are mostly designed to be as generic as possible while supporting any operation that might be needed. This means optimizations that are universal like dead code elimination(DCE) can be performed in the IR without in-depth knowledge about the programming language or the target architecture. Obviously, optimizations that are specific to the programming language or the instruction set architecture might not be as easy to perform on a generic IR itself. They should then be performed with the appropriate representation.

The generic approach of most IRs means that multiple languages can be converted into the IR which can make the compiler retargetable for the source language. Also, machine code selection can be changed to support different instruction set architectures more easily. This allows programmers to write compilers for a large number of targets and source languages without rewriting the compiler for each possible combination. The amount of work needed shrinks from O(n*m) to O(n+m) where n is the number of programming languages and m is the number of instruction sets. This idea of a “universal computer oriented language”(UNCOL) dates back to the “Communications of the ACM Volume 1”[9] in 1958 but was never seriously implemented. Compilers over the years tried different approaches to create general intermediate representations, for example, GCC with its register transfer lists(RTLs) and others, but none were remarkably successful.

RTLs, for example, supported many Algol-inspired programming languages(for example C and C++) but writing frontends for any existing language was quite difficult. In the years 2003 and 2004 the intermediate representations LLVM and GIMPLE respectively were released. They are the intermediate representations of the compilers Clang and GCC respectively and are in a single static assignment(SSA) form which we will go into detail later. Both seem to lead into the direction that UNCOL was pointing to, but the developers made clear[10], that their IRs are not meant to be absolutely universal.

Intermediate representations are not only used in compilers. Most analysis-tools first transform a program or a piece of code into a specialized intermediate representation to perform their analyses.

An instruction representation is a specialized form of an intermediate representation. The goal is to represent what an instruction actually does. There exist a lot of instruction representations for example REIL[11], ESIL[12], BIL[13][14] etc. The goal of such a representation is to describe in a computer readable form what effect an instruction has on the state of a machine. It is important that the description is precise so that it may be used to interpret a program as accurate as possible.

2.2.2 Static Single Assignment form

A static single assignment form(SSA-form)[15][16] is a type of intermediate representation that has special properties:

- *Only one definition*: Different to other intermediate representations, in SSA-form a write to a variable can occur only once. That means a reference to a variable points to exactly one definition. In normal programming languages, variables are called that way because over their lifetime different values can be assigned to them. Technically that means that the term variable in an SSA-form is not correct. We will henceforth refer to them as definitions.

  This property makes a lot of analysis techniques easier and some become trivial to implement because the usage of a variable has only ever one exact definition. It can neither be overwritten nor deleted.

  When transforming a programming language into an SSA-form, variables are typically numbered and each assignment is converted into its own definition[17].

- $\phi - expressions$: In SSA-form, phi(ϕ)-expressions are special constructs. If a basic block has multiple incoming edges (for example the head node in a loop or the block after a branch), multiple values need to be merged together. For every incoming control flow edge, a parameter is needed for the phi-expression, that takes this value and merges them based on where the control flow came from and create a new definition. All phi-expressions for each block are executed simultaneously. There is no ordering for phi-expressions so in the case of phi-expression references another they refer to the phi-expression from a previous execution.

Advantages

An IR in SSA-form has important advantages, which other intermediate representations might not share:

- It is easy to reason about values because there can only ever be one definition for every value. Two uses of the same definition always hold the same value. For variables, on the other hand, it is possible that values change during their lifetime.
- It is easy to generate such information as reaching definitions or liveness for dataflow analysis.
- Optimized code might decide to reuse registers for different usages to minimize register pressure[18]. When generating a static single assignment form all assignments to a register are handled independently. This breaks dependency chains between the two register-uses and register reuse is mitigated.

Because of the features of an IR in SSA-form, certain optimizations are easier to implement. These include:

- Constant Propagation
- Full/Partial Redundancy Elimination
- Dead Code Elimination
- Various Loop Optimizations

But are not limited to these passes. Even instruction selection can be performed directly on the IR in SSA-form[19].

Let us take a look at a prominent set of compilers that nowadays are using a static single assignment form.

GCCs SSA form

Since the beginning GCC used a straightforward technique for optimizing code[20]. The idea was to create a large number of expressions and not to care about optimization first. Then peephole optimization is used to compress the expressions back together into efficient code. This technique worked quite well but had the problem of often generating sub-optimal solutions. Also, the IR that was used(RTLs) was not suitable for a number of optimization techniques. In 2003 an IR in SSA-form(GIMPLE) was proposed[21] so that new optimization passes could be implemented in the compiler and existing passes could be reimplemented much easier there.

In Figure 2.1 you can see a flow chart of all new stages and which optimizations are done on each stage. The RTL-backend and instruction selection mechanism still remain unchanged.

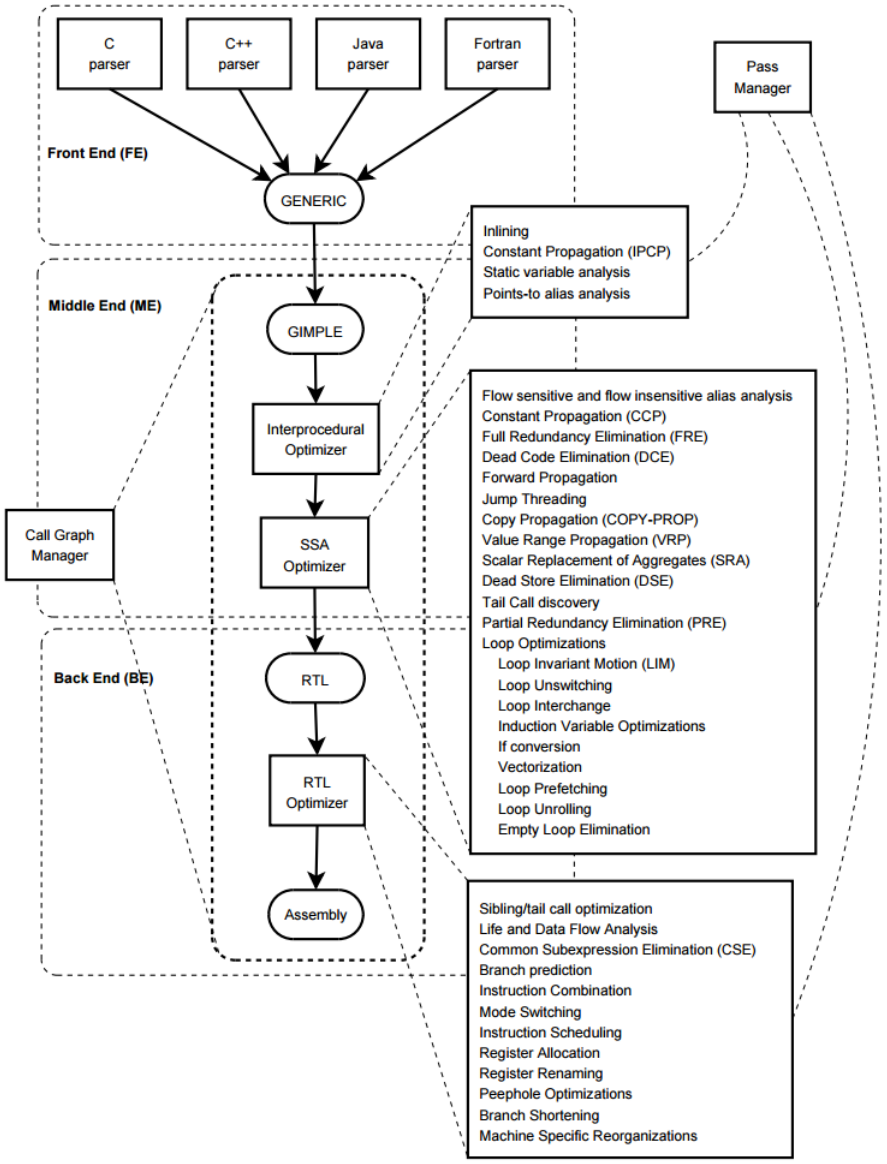

Figure 2.1: GCC Overview[22]

Clang/LLVM

LLVM(Low Level Virtual Machine) is the intermediate representation used in the compiler framework Clang. It is based upon a fully typed IR in SSA-form. LLVM was first developed for GCC[23]. It was designed to replace the GCC-IR at the time, which

did not allow for extensive inter-procedural optimizations and LLVM make certain optimizations[22] easier to implement. It was not included into the GNU Compiler Collection but was turned into its own full-fledged C/C++/Objective-C compiler with the creation of Clang. Other languages like Haskell, Swift and Rust later created their own Front-ends for LLVM in order to implement their own compilers on top of LLVM.

LLVM is an open source framework and can be used for all sorts of things. This means that many projects use this intermediate representation for different goals. As we will mention later on in Section 3.4.2 RetDec, for example, uses LLVM as their IR successfully in their decompilation process.

2.2.3 Memory-SSA

Representing memory-operations is always difficult because memory is just a generic concept of storing data. For compilers tracking variables can become difficult if pointer arithmetic is possible. Figuring out if two pointers address the same memory location(aliasing) is an undecidable problem. Lots of techniques exist to solve the issue[24][25][26].

One such approach uses an SSA-form to represent memory access[27][28]. This has all the advantages of the SSA-form while being able to sparsely represent variables in said memory.

2.2.4 Peephole Optimization

Peephole optimization is an optimization stage that can be used to convert certain patterns of expressions to more specific or better representations with the same meaning[29][30].

The name peephole optimizer comes from the fact that only a few expressions are viewed at a time. It looks at the code through a small hole(a “peephole”). The algorithm tries to match a small batch of expressions with its patterns. If a pattern is matched, then the matching expression is replaced according to the rule. This optimization technique is common in compilers. Especially in the earlier versions of GCC peephole optimization was crucial, because lots of expressions were generated from the source code, which then needs to be compressed as tightly as possible before machine code generation. Peephole optimizers do not need a complicated algorithm to traverse the expressions. They can simply loop through all expressions and try to apply the patterns each time. Every time an expression has been replaced, all rules have to be applied again to guarantee, that the maximum amount of rules were applied.

CHAPTER 3

State of the Art

Compiled programs lose a lot of metadata during the compilation process, like most type-information and control flow information. There are abstract instruction set architectures for which this is not the case such as WebAssembly where the control flow is kept intact. Such architectures are typically not used to run directly on hardware and are either interpreted or first compiled by a just in time compiler before being executed on actual hardware. For binary programs, which are directly executed on a CPU, that information is generally not available. To reconstruct higher level code it is therefore essential to extract such information out of the available byte-code. There has been much research in recovering structured information from binary programs.

3.1 Variable Detection

Variables are important in programs. They are typically either stored in a global section, the heap or the stack. Detecting these variables and labelling them is helpful as it improves the readability if memory is addressed by a name instead of a pointer. Pointers are not easy to read as they are mostly either big static numbers or a value/argument plus an offset. Replacing this representation with a name improves clarity. This reconstruction is not trivial because binary executables do not have the structure that programming languages have. Data can be laid out in memory in many different ways and locations with no inherent structure behind them. While data can be completely unstructured most of the time there is a certain structure behind data, which can be reconstructed.

Developed by G. Balakrishnan and T. Reps[31] Value Set analysis, for example, aims to detect variables on the stack, the heap and any global section. They track pointer and integer values at the same time to find so-called abstract locations which correspond to variables. Additionally, they over-approximate the values that a register or a memory location might hold at certain points in a program. This is represented by a value-set.

The value-set is expressed by a start, an end and a stride. Having this information, the algorithm can infer for example array sizes and jump tables.

3.2 Type Reconstruction

Type Reconstruction deals with the problem of reconstructing higher-level types from a binary[32]. Type information is typically added to a binary as debugging information. It can be used by decompilers to reconstruct the original code but it is uncommon to decompile a binary with debugging information as programs are normally stripped of such superfluous data before being released. This is especially true for programs where the source code is not publicly available such as commercial programs or malware. Therefore in type reconstruction, it can be assumed that such additional information is unavailable and results need to be generated purely from the code itself.

Types are important as they structure the program in a way of giving context to what are otherwise non-distinct bit-fields. Adding context results in much better readability and makes the code understandable.

A. Mycroft in 1999[33] already uses a type reconstruction algorithm based on the type inference algorithm used in ML designed first by Hindley[34] and then Milner[35]. The algorithm creates type-constraints for every instruction in a program. Then unification is used to resolve these constraints. In case of conflicts casts or union types are inserted.

In Aggregate Structure Identification(ASI)[36] composite types are created lazily. The usage of the program defines the types created. With these structures of aggregated types can be reconstructed.

Combining value-set analysis and aggregate structure identification allows even better analysis of the usage of memory[37][38][39]. ASI is doing data structure reconstruction while VSA analyses only the values held by the registers or memory locations, with a basic variable detection mechanism. Using ASI initially for structure detection and then running VSA on the results increases the quality of the input for the value-set analysis. With these results, ASI can be performed again having now much more information about possible values available. The results can be iteratively refined. Problems appear in loops where there is an affine-relationship between values. For example, loop-counters and pointers may be iterated separately in a loop. Normally, this means a finite value-set cannot be found by VSA. To alleviate this problem an affine-relationship analysis is introduced between ASI and VSA. This gives the value-set analysis enough information to find correct bounds for the value-sets. There are also best-effort implementations where an unsound pointer analysis during VSA is used to speed up the process to make it more scalable[40]. The downside of this approach is losing out on the number of variables found.

In comparison to this, Mycrofts approach has the distinct problem of not being able to discover arrays well. Only static offsets play into the creation of new types. This

new approach looks at the possible values and can, therefore, analyse arrays much more efficiently.

A different approach is used by REWARDS(Reverse Engineering Work for Automatic Revelation of Data Structures)[41] where dynamic execution is used to record how data is actually used. It works similar to the previously described algorithm but it dynamically checks the program. Additionally, memory locations are tagged which makes it possible to include type-information into a memory-dump. Initial type information is generated using so-called type sinks, which are functions whose type signature is already known. Good type sinks are standard library functions. These calls are used to infer the types that are used in the program.

TIE[42] uses static analysis to recover types from a binary. The code is analyzed using the Binary Analysis platform(BAP)[14] and raised into BIL which is the corresponding intermediate language. Before type reconstruction, a VSA algorithm is used to reconstruct variables for functions. The results are used as a base set of terms of unknown type. Then constraints are applied on that set e.g. a signed division instruction introduced the constraints that both arguments are signed types. These constraints are then resolved and types are generated. TIE is more conservative than other techniques but still achieves high accuracy.

3.2.1 The Type Flattening Problem

General problems of type reconstruction are the loss of hierarchical data structures. Nested structures cannot be detected and are often flattened out. This problem is not solvable without additional metadata or human input. As a basic example, the C-types for a linked-list entry can be seen in Listing 3.1.

```
struct highscore_entry{
    char name[20];
    unsigned int points;
};

struct list_entry{
    struct list_entry* next;
    struct highscore_entry value:
};
```

Listing 3.1: C-Example of Types which will be flattened by a compiler

The *highscore_entry* structure will not be reconstructed correctly because a decompiler can not know the meaning of the struct and it will at best reconstruct a data structure such as in Listing 3.2.

```
struct list_entry{
    struct list_entry* ptr;
    char string[20];
    unsigned int number;
};
```

Listing 3.2: C-Example of the flattened Type of Listing 3.1

The data structure is flattened and the hierarchy is lost. Without additional metadata, the decompiler cannot recognize the original structure as shown in the example. The only way the sub-type might be reconstructed if a pointer is being taken to the sub-struct and the correct type is being inferred for the pointer like in Listing 3.3.

```
void do_stuff(struct highscore_entry* highscore);

void a_function(void)
{
    ...
    list_entry entry;
    ...
    do_stuff(&entry.value);
    ...
}
```

Listing 3.3: Usage of the sub-structure of the types defined in Listing 3.1

Typically compilers store data structures flat in memory, which makes it difficult to recover structures. The memory model of most languages that compile straight to a binary(like C and C++) composite types by directly including sub-structs in their parent-structs. Without additional information like debugging information, this composition remains indistinguishable from the flat version. There are algorithms that use the *baseaddress + offset* pattern[43] to reconstruct types and sometimes succeed in the reconstruction of composite types.

Another algorithm[44] uses dynamic analysis which may lead to suboptimal code coverage and incomplete results, because of code-paths not being taken into account.

3.2.2 Using Heuristics and Idioms

A common approach to solving type flattening and reconstruction is to use heuristics based on common instruction idioms. Compilers usually generate certain instruction-patterns for certain types. Heuristics are then used to guess the correct types. These heuristics are successful in common cases, but they may return wrong results. A good example

is function-arguments. Around a function call, you usually see two things. Arguments are put into the correct places according to the calling convention(either pushed onto the stack or moved into a certain register) and caller-saved registers are pushed onto the stack. Let us look at an example in Listing 3.4 for the x86-64 architecture with the AMD64 calling convention.

```
call 0x500
sub rdx, r11
sub r9, r15
add rax, rsi
mov rsi, rax
mov rdi, 0x200
call 0x80
```

Listing 3.4: Disassembly of two chained function calls

The AMD64 calling convention specifies the first six integer arguments as being the registers *rdi, rsi, rdx, rcx, r8, r9* in this order. A simple heuristic would be to look at the instructions prior to the call instruction, iterate backwards and assume all instructions being arguments. To not pollute the result the iteration stops if a calculation appears that is not relevant to the possible arguments. In our example, the first argument will be determined as 0x200 by this heuristic and the second as whatever the value of *rax + rsi* is. The instruction *sub r9, r15* does not refer to an argument so the heuristic finishes. The problem with such a simple approach is that *rdx* may be relevant again. An analyser not aware of that will produce false function signatures as if a compiler orders the instructions in an unusual manner.

It is possible to mitigate such mistakes by improving the heuristics with a comprehensive dataflow analysis. This would make it able to follow the dataflow of all possible registers and then decide based on the heuristic. This gives better results as non-relevant instructions can be ignored.

3.2.3 Variable Unification/Memory Reuse

Another type of problems are overlapping local variables. This pattern is generated when local variables are used in different control code paths of the same function and they exist in different non-overlapping scopes. Local variables typically are stored on the stack. If the scope of a part of a function does not overlap with another scope then local variables of the first may exist inside the same stack-section as variables from the second scope. This can confuse a type reconstruction algorithm which will either assume both being the same variable with occasionally casts or introduce a union type to represent the usages. Again this problem is not completely solvable without additional information. Variables in two different scopes are fully independent of each other and don't exist outside their

...
int i
struct custom_struct var1
int new_variable
struct random_struct var2
...

Table 3.1: A naive stack frame for the code in Listing 3.5

...
int i, new_variable
struct custom_struct var1, struct random_struct var2
...

Table 3.2: A better stack frame for the code in Listing 3.5

respective scope. Compilers typically try to optimize memory consumption and that includes the stack memory. Recycling memory which is not needed any more and where the lifetimes do not overlap is a typical optimization done.

```
for(int i = ...){
    struct custom_struct var1 = ...
    ...
}
...
{
    int new_variable = n;
    struct random_struct var2 = ...
    ...
}
```

Listing 3.5: Control flow blocks with different variables in non-overlapping local scopes

Given the example in Listing 3.5 a simple stack-frame can look like in Table 3.1. This is not space efficient as *i* and *new_variable* are never alive at the same time as *var1* and *var2*. A compressed version of the stack frame will therefore look like in Table 3.2.

The stack frame in Table 3.2 is much smaller than the one in Table 3.1 and reuses space, but makes it subsequently more difficult for decompilers with no knowledge about local scopes to reconstruct structures. This information can be available via debugging information, but type reconstruction should not rely on such, because debugging information typically is not available and if it is the type information for the stack is commonly found there, which makes the whole process unnecessary as the information is readily available.

A type reconstruction algorithm can in our example not distinguish between *i* and *new_variable* as they are stored in the same memory location and they are actually of the same type. It will erroneously report only one variable of type *int*, which is unsatisfying but without an alternative. Struct *var1* and *var2* are special as they will most likely not be recognized in their original form. Probably an amalgamation of types will be found at different offsets and will be represented as some kind of union-type. This is in consequence of the fact that a stack is, in essence, nothing more than a struct that holds all local variables. Any local struct will then not be recognized, similar to the problem with type flattening. The structs are embedded into the stack and cannot be reconstructed reliably as already discussed earlier.

3.3 Control Flow Reconstruction

When compiling into machine code most information about structures from higher level programming languages gets lost. Concepts such as functions do not exist in most architectures. There are helpful instructions in some instruction sets that help implement these concepts like *call*, *return*, *branch* or *jump* which perform common operations used to implement such subroutines, but programs are not restricted in any way in how they use them. The same can be said about the control flow of a function. Programming languages usually have a predefined control flow that is driven by certain control flow structures like branches and loops. In some languages like C and C++ programmers have access to the goto-expression, where the control flow jumps to another arbitrarily placed label in the same function. This breaks the structured control flow and usually is highly discouraged as it often produces unwanted side effects. Additionally, goto structures are hard to read as the control flow can jump arbitrarily. A loop, for example, is easy to understand. There is one incoming control flow edge and one outgoing edge. The inner block is executed a certain number of times dependent on the condition. Even the additional concepts of break and continue are easy to understand. While the first exits the loop, the latter continues with the next iteration prematurely.

```
int i = 0;
while(i < 10) {
    printf("Iteration␣%d\n", i);
}
```

Listing 3.6: A Simple Loop

The control flow structure of Listing 3.6 is understandable because it follows normal reading patterns. The control flow goes from top to bottom. Let us look at a more complicated example using c++ with gotos:

```
if(version){
    if(version == 1){
        obj = new Object1();
        goto L1;
    }
    if(version == 2){
        obj = new Object2();
        L1:
        global_obj = obj;
        goto L2;
    }
    if(version == 3){
        obj = new Object3();
        goto L1;
    }
}
L2:
return obj;
```

Listing 3.7: Branches with gotos

The example in Listing 3.7 is hard to understand. At first glance, it looks like some objects are created and they are stored in some global variable. but the exact relationship between the objects and what happens with them is not obvious. The example in Listing 3.8 is a more cleaned up version of exactly the same Code.

```
if(version){
    if(version == 1){
        obj = new Object1();
        global_obj = obj;
        return obj;
    }
    if(version == 2){
        obj = new Object2();
        global_obj = obj;
        return obj;
    }
    if(version == 3){
        obj = new Object3();
        global_obj = obj;
        return obj;
    }
}
return nullptr;
```

Listing 3.8: Same code as Listing 3.7 without gotos

We did not intentionally create the example in Listing 3.7 to show a strange control flow but we found such programs in real-life. The reason for this is quite simple: Compilers do a lot of work under the hood. One optimization is to reduce the number of return instructions. Usually, in optimized programs, there is only one return per function. This explains the single return at the end. The gotos exist because the compiler detects duplicated code when setting the different objects to the global variable and creates only a single code path for them. The *goto L2;* statement is also needed because otherwise, the control flow might lead back into the branch with version-3 which would lead to an infinite loop.

This example was created with an older Microsoft visual studio compiler(MSVC++ version 8.0). And displays clearly the problem with reconstructing control flow. Gotos are bad because they destroy readability. Therefore reducing them to an absolute minimum is preferable.

There are techniques that use pattern matching to reconstruct higher level control flow structures[45]. The patterns are predefined and do not change. If no patterns match then gotos are inserted. Different sophisticated algorithms exist that try to refine the control flow better. The Phoenix decompiler uses an iterative approach to insert gotos if no pattern is found[46] and tries again similar to Chaitins register allocation algorithm[18]. This effectively reduces the amount of gotos compared to a more naive approach because the control flow edge makes the control flow graph simpler and patterns can be found more easily.

Goto-free control flow structure recovery[47] was developed because the pattern matching algorithms were still creating unnecessary gotos. Instead of using patterns, the semantics of the pattern is used to recover structures. A loop, for example, is just a cycle in the control flow graph. Using such simple rules, it is possible to eliminate almost all unnecessary gotos.

3.4 Existing Tools

There already exist a lot of tools for reverse engineering binaries. The two types which we will take a look at are disassemblers and decompilers.

3.4.1 Disassemblers

Disassemblers simply take a binary and disassemble code sections based on the metadata found in the binary. The disassembled instructions are then displayed to the user directly without transformations. Some disassemblers organize the disassembly into control flow blocks to visualize the control flow.

Objdump is a part of the GNU binary utilities and displays information of an object file. Executed with the option -d all sections, which are marked as executable, are disassembled by objdump. Instructions are detected via a linear sweep algorithm.

Radare2 (or r2 for short) is an open source reverse engineering tool. It can analyse a wide range on binaries and architectures. It uses a recursive descending parser which takes information of metadata, debugging information and other sources to try to disassemble as many functions as possible. A user can interact with the tool via a command line interface. The instructions of a function and the control flow can be displayed in a linear form with additional comments such as cross-references and jump-targets/jump-destinations. It also offers a graphical visualization, where a function is split into basic blocks, connected via labelled edges. Radare2 has a lot of functionalities and is a powerful tool for debugging or reversing a binary.

Binary Ninja is a GUI application which has a clean interface and focuses on reverse engineering. It has a disassembler which displays the result in a basic block graph.

Similar to Binary Ninja, IDA is a GUI application to disassemble programs. It supports a wide range of instruction sets and file formats and also acts as the base application for the Hex-Reys decompiler.

Capstone is an open source disassembling framework that supports a lot of different architectures. Being a library, it is used in a lot of different software solutions. Radare2 and Snowman, for example, use it. Capstone supports Arm, Arm64, Ethereum Virtual Machine, M68K, M680X, Mips, PowerPC, Sparc, SystemZ, TMS320C64X, XCore, x86 and x86_64.

3.4.2 Decompilers

Because of the complexity of decompilation, there are not many decompilers available to the public.

Hex-Rays[48][49] is a commercial decompiler for the x86, x86_64, ARM, ARM64 and PowerPC architectures. It is sold as part of IDA Professional[50]. The price range of an initial licence goes from 2600 USD to 4000 USD. Because of its commercial nature, there is no detailed information available about it. They use a proprietary intermediate representation to represent code. The first step translates the instructions into that IR, which is akin to microcode. The reason given is to abstract processor specific concepts by a simple language. The IR code generation is handcrafted for each processor type.

The generated IR does look like RISC-code. The difference is that optimizations can make expressions more complex by combining multiple together. For example a conditional jump might contain a complex condition such as *if (argv[1][1] == 'a')*. Such a condition is represented by the IR as:

jz [ds.2{4}:([ds.2{4}:(ebx.4{8}+#4.4){7}].4{6}+#1.4){5}].1{3}, #0x61.1, 7

The whole process is similar to register transfer lists(RTLs) in GCC. First, a lot of expressions are generated which are then combined together to form more complex patterns. Unlike GCC which generates instructions, the Hex-Rays decompiler creates pseudo-code expressions.

Hex-Rays also uses a wide variety of rules to optimize the expressions. The rules mostly work on def-use-chains, which are kept for each expression.

Also remarkable is stack handling. One part of the stack is mapped to registers, and the other is aliasable. To determine if two pointers are aliases of each other (which means that they point to the same location) is, in general, an undecidable problem. To alleviate this, some sections of the stack are to be considered unaliasable. This means if two pointers are using two different dynamic values and one point into the unaliasable section then they do not alias. Let us visualize this using an example for the *x86*-architecture. We have two memory writes to locations *esp + 4* and *eax + 4*, where both *esp* and *eax* are not changed from the function-entry to their use. If *esp + 4* is not an aliasable location then the two memory locations are not aliases of each other even if it might be theoretically possible for them to refer to the same location. The optimizer will always treat them as different.

In comparison to Hex-Rays there exist multiple open source decompilation solutions. Snowman[51] is one of those. It supports the x86, x86_64 and ARM architectures. It has created its own IR to represent the program. The generated code is C/C++. The code is freely available on Github and there are plugins for IDA, radare2 and x64dbg.

The Retargetable Decompiler(RetDec)[52] is an open source decompiler developed by Avast which is based on LLVM. It supports a certain range of 32-bit architectures such

as x86, ARM, MIPS, PIC32 and PowerPC. RetDec detects and uses debug information to among other things reconstruct type information(including C++ types and vtables) and uses instruction idioms to reconstruct higher level code.

There are other decompilers such as:

dcc A decompiler which transforms *i386* binaries for DOS to C programs[53][54]. It uses its own intermediate representation. Control and data flow analysis is used to improve on the code it produces.

Boomerang This is a modular and retargetable decompiler. It supports x86 Sparc and PowerPC[55][56]. The target-language is C. It was last updated in 2012.

REC decompiler The REC decompiler supports the architectures x86, x86-64, MIPS, m68k, Sparc, and PowerPC[57]. It is closed source but the decompiler itself is available for free to download. The last update to the project was done in 2011.

Ghidra In March 2019 the NSA has released a part of the source code of their in-house developed decompiler called Ghidra[58]. It is claimed to be on par or even better than other current decompilers.

Proprietary decompilers

Most other decompilers are as far as we could discern fully proprietary, no longer in active development or generally not available to the public. One of them is the Phoenix decompiler which is referenced only in academic papers[46][47] but without an available implementation.

Specialized decompilers

There are specialized decompilers for a wide range of languages that are specially designed for those languages. Most of them take a special input format which represents the program for a specialized virtual machine like Java, C# or compiled Python. To achieve more platform independence, binaries of such languages are not compiled for a specific processor architecture, but into a higher level intermediate representation which retains much more of information about control flow and program structure.

For example compiled Python: Code is simply converted into a binary representation. Short stack based snippets are generated for every line in the code. With this information, the original source code can be reconstructed.

Java and C# are a bit different as they are compiled into byte code and then JIT-compiled into actual native code. The byte code retains a lot of structure and information to reconstruct the source code similarly to the original.

There are special kinds of decompilers which work on binary executables but mostly consider metadata. One such example is the Haskell decompiler *hsdecomp*[59] which

looks at metadata generated by the Glasgow Haskell Compiler additionally to the native code itself.

All of those specialized decompilers convert a binary representation into a human-readable code but they are fundamentally different from a general purpose decompiler. A program such as Hex-Rays, retdec or snowman works directly with native code having no overarching structuring rules. A Java decompiler, on the other hand, uses the information that is inherent to the Java byte code for reconstruction and is specialized for this task. This means, that these programs solve different problems as they have completely different assumptions that they are able to make about their input.

CHAPTER 4

Implementation

Our implementation is written in C++ and encompasses disassembling, IR-generation, multiple optimizations passes, and finally pseudo code generation. Here we will present our implementation and explain the techniques we used and implemented in detail

4.1 Objects

To describe the binaries we have to somehow depict the information in a data structure. Most objects we use are addressed by an id, which is used to access them. This allows for fast access and efficient storage of the information.

We organized the data into the following objects:

Architecture Each instruction set architecture has to have a definition in our decompiler. An architecture is defined by at least the following list of data:

- Bit-base: Specifies how many bits are in one byte.
- Byte-base: Specifies the maximum number of bytes in one basic-word as used by all instructions.
- Instruction-pointer-size: Specifies how many bytes the instruction-pointer has.
- Registers: A list of all registers.
- Stacks: A list of stacks.
- Memories: A list of memory-spaces.
- Builtins: A list of built-in functions.
- Instrdefs: A list of instructions and how they are resolved into the IR.

Stack This represents a built-in stack which is handled by the architecture. It defines how the stack grows, what the stack-pointer is, if the stack is in the memory or if it is simulated in registers(e.g. x87 floating point stack). At the moment we only use this for simplifying decoding pop and push expressions into our SSA-IR.

Memory-Space Represents a region in system memory that can be addressed by a program. It has a maximum size and a word-size. The word-size defines the number of bytes a single memory location contains. This is needed because there are architectures where each memory addressable location can contain multiple bytes.

Built-in Some specific behaviour is hard to represent in a generic way. We use Built-ins to represent such functionality. Cryptographic instructions, for example, should be represented by a built-in because otherwise, everything about the instruction would necessarily need to be modelled by us. Using a single built-in for such cases is preferable because it simplifies the generated code. Each built-in has a return type and bit length defined, so they can be transformed into the IR and be typed correctly there.

Register This is our representation of registers. They are crucial components in most CPU-architectures.

Instruction Definition They are composed of a mnemonic to identify the instruction and a list of possible IR-translations. How their IR-translations are used to generate the IR is explained in Section 4.3.

Binary A binary is an executable which maps data from a file into memory. It is used to translate memory-addresses from the offset in memory into the original file. We also store additional information which is necessary for reading data and disassemble code, such as endianness and the CPU-architecture. Additionally, it is used to store global information about memory such as symbols and sections.

Symbol These define a certain location in memory that is of interest, this includes global variables, functions, and GOT-table entries.

Section A section is an area of memory that has certain properties. Sections can be marked as readable, writeable and executable.

We will go more into the details for registers and memory-spaces because they are not trivial constructs and they are vital for the implementation.

4.1.1 Register

Registers are a crucial part of every architecture. Some architectures(e.g. x86, x86_64) have overlapping registers, which makes modelling even more difficult. The *rax* register in x86_64 is a good example. It has 64 bits. The register *eax* is 32 bit in size and occupies the lower half of *rax*. The register *ax* has 16 bits and occupies the lower bits of *rax* and

eax. Finally there are *al* and *ah*, which are both 8 bit and *al* occupies the lower 8 bits of *ax* and *ah* the higher 8 bits of *ax*. If *al*, *ah* or *ax* are written to, only the bits occupied by that register are actually written, but if *eax* is written to, then the upper 32 bits of *rax* is cleared(filled with 0s). A visual representation of the *rax* register can be seen in Figure 4.1.

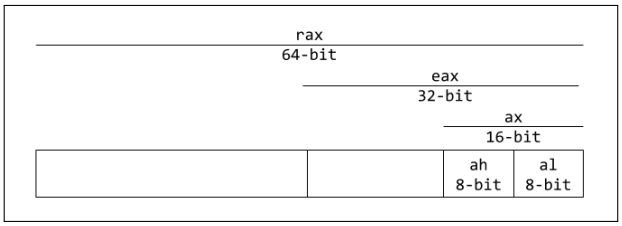

Figure 4.1: Example of the segmentation of the *rax* register in x86_64

Because of those complicated behaviours, each register in our definitions has the following attributes:

- Size: Size of the register in bits.
- Offset: Offset of the register from its up-most parent.
- Parent Register: The upmost parent register.
- Direct Parent Register: The direct parent register.
- Clear on Write: Whether to clear the rest of the parent register when this register is written to.

Notice that we assume that registers can only have one parent. That means registers may only be totally contained in other registers. Partly overlapping registers are not allowed as can be seen in Figure 4.2.

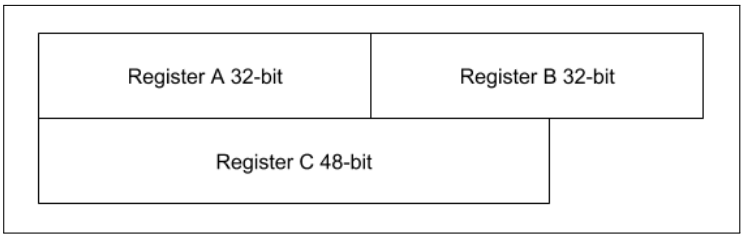

Figure 4.2: Overlapping Registers which are not allowed

4.1.2 Memory-space

Computers can be based on different architectures, which may have different memory models. A well-known memory model is the Von Neumann architecture, which has only one memory-space where both data and code is located. Another model is the Harvard architecture having two distinct memory-spaces. One for code and one exclusively for data. Access to these memory-spaces may be different. For example, the AVR architecture (for 8-bit micro-controllers from the company Microchip) has two distinct memory-spaces: RAM is for changeable data only and flash memory is purely used for the program (and global data which typically is copied into RAM during startup). Each has its own address space. A location in program memory most likely holds a different value than the location at the same address in the data memory. In the AVR architecture, the data memory is addressable one byte(8 bit) at a time. On the other hand, program memory is addressable two bytes(16 bits) at a time. Some binary analysis tools ignore those differences. A good example is objdump for AVR. It does not take this difference into account and addresses program memory one byte at a time and has to therefore also adjust the pointers. This can cause confusion, because addresses may not correspond to the actual instruction-pointer in a running program. For AVR it makes sense to change the minimum addressable word in program memory because there exists a special instruction (*elpm*) which uses addresses as if program memory is organized into 8-bit words. Changing the minimum addressable word-size makes modelling this instruction much easier, but for other architectures, this may not be the case.

A memory-space has a name, a word-size (for addressing as discussed above) and a list of Data-segments. The list of Data-segments is needed because it is not always the case that one continuous block of data is mapped into memory as defined by the program binary. It enables us to model fragmented sections of memory.

4.2 Virtual machine

We defined a virtual machine for our intermediate representation(IR). We translate all instructions for a given architecture into that IR. Therefore, the virtual machine should be flexible and extensible to the point in which every instruction in as many architectures as possible can be represented in the virtual machine.

The virtual machine works with typed bit-arrays of a specific size. Each operation has a type, which defines its semantics. For example, an addition with *float*-types works different than an addition with *unsigned integer*-types. Underneath all of that, the values are still bit-arrays, but the type defines how they are interpreted and what operation is performed.

The default type *unsigned integer* means that a bit-array can be directly interpreted as a binary number. With this interpretation, we can use optimizations such as constant propagation out of the box. We also support signed integers and floating point numbers. Optimizations relying on the bit layout of a type are not possible. For example, a bit-shift

operation of a one-complement signed integer results in a different value than the same operation on a two complement signed integer. Therefore we have to be careful with optimizing expressions that are bit-layout independent such as bit-shifts, bit access, bitwise and/or/xor/not, etc.

4.2.1 Operations

Operations assume, that signed and floating point representations are consistently used throughout the instruction set: A signed add operation shall always do the same. One-complements in one version of the instruction and two-complement in the other are not intended. If an architecture has multiple different representations of signed integers, built-ins are needed to represent one of them. In the future, it might be possible to add types to the architecture definition. This would allow us to define custom types, which would allow for more than the current three types.

There is a large number of operators defined in the virtual machine divided into four general types:

- Control Flow: These operations don't return a value, but influence the control flow of the current function. They either wrap expressions into different constructs like branches or loops, jump to different instructions or call other functions.
- Arithmetic: These operations have predefined functionality, which is performed on the input values. The type of operation defines the return value and also the type on which the operation is performed.
- Memory: These operations manipulate a memory-space. Pop and Push operations are special cases because stacks can be memory or register backed. Memory backed stacks write and read to memory and use a register as stack-pointer. Register backed stacks (e.g. x87-floating point stack) use designated registers to function.
- Misc: Miscellaneous operators that add necessary functionality like assign and cast.

4.2.2 Flags

Flags are set implicitly when an arithmetic operation is performed and are available until the next arithmetic operation occurs. There are three flags: carry, overflow and underflow. By default, flags are unsigned integers with a length of 1. The carry-flag can also be annotated with a size n. In that case, it is interpreted as the carry from the bit at position n. If no size is given, it is implicitly the carry from the most significant bit.

Carry The carry flag is set if an arithmetic operation needs one or more bits than are available to perform an operation.

Control Flow Expressions	
trap	Traps the execution and stops the program
syscall	A system call using the first argument as the id.
call	A function call using the first argument is the target of the call.
jump	Jump to a different Location. The arguments appear in pairs, except for the last, which is the default jump-target. The first is the condition, the second is the jump-target. The jump-target of the first pair whose condition is not 0 is chosen by the jump or the default jump-target if no condition matches.
return	Exits a function. The first argument is the target of the return. The rest is the current state of the machine.

Table 4.1: Control Flow Expressions

Arithmetic Expressions	
greater	1 if the first argument is bigger than the second or 0 otherwise
greater equals	1 if the first argument is bigger than or equals to the second or 0 otherwise
less	1 if the first argument is lower than the second or 0 otherwise
less equal	1 if the first argument is lower than or equals to the second or 0 otherwise
equal	1 if both arguments are the same or 0 otherwise
addition	Adds 2 or more values together.
subtraction	Subtracts the remaining values from the first.
multiplication	Multiplies all values together.
division	Divides the first Argument by the remaining arguments.
modulus	Divides the first Argument by the remaining arguments and returns the remainder.
and	A logical and operation.
or	A logical or operation.
not	If the argument is 0, return 1 otherwise return 0.
binary and	A bitwise and operation.
binary or	A bitwise or operation.
binary xor	A bitwise xor operation.
binary not	A bitwise not operation.
shift right	A bit-shift to the right.
shift left	A bit-shift to the left.
rotate right	A bit-rotate to the right.
rotate left	A bit-rotate to the left.

Table 4.2: Arithmetic Expressions

Memory Expressions	
load	Loads values from memory. The first argument is a memory-space, the second the address and the third is the amount of bits to load.
store	Stores values to memory. The first argument is a memory-space, the second the address and the third is the value.
push	Pushes a value onto a stack. The first argument is a stack, the second is the value.
pop	Pops a value from a stack. The first argument is a stack, the second is the amount of values to pop.

Table 4.3: Memory Expressions

Misc Expressions	
assign	Assigns a value to another. The first argument is either a register, a temporary or an argument. The second is the value to be assigned.
append	Appends multiple values together.
nop	No-Op
cast	Casts a value from one type to another.

Table 4.4: Misc Expressions

- **Addition** The flag is set if the operation would result in a value to be bigger than the maximum possible value.
- **Subtraction** The flag is set if the operation would result in a value to be smaller than the minimum possible value.
- **Multiplication** The flag is set if the operation would result in a value to be bigger than the maximum possible value.
- **Bit shift** This flag is set if the last bit shifted out was a 1.

Overflow The overflow flag is set if the sign of the result differs from what it normally should be if the calculation was done with infinite bits or a push is performed on a stack, which is already full.

Underflow The underflow flag is set if a pop is performed on a stack, that is empty.

4.3 IR-Translation

We created a simple custom language, which we parse and use to create our SSA-form. This makes the whole SSA-IR generation process dynamic and allows fast changes and easy extensibility as new instructions only need to be disassembled and defined in our IR-translation language from which they are automatically transformed into the intermediate representation. The translation language is based upon the virtual machine we defined previously.

rjmp	#jmp(#arg[1])
breq	#cjmp(#arg[1],$zf,#arg[2])
mul	#seq(=(#t[1],#mul(#arg[1],#arg[2])),=($r0,#t[1][0,8]),=($r1,#t[1][8,8]), =($zf,==(#t[1],0)),=($cf,<[s](#t[1],0)))
movw	=(#arg[1],#arg[2]) =(#arg[1],#app(#arg[2],#arg[3])) #seq(=(#arg[1],#arg[3][0,8]),=(#arg[2],#arg[3][8,8])) #seq(=(#arg[1],#arg[3]),=(#arg[2],#arg[4]))

Table 4.5: Example of IR-translations

A few examples of instruction definitions can be found in Table 4.5.

As seen in the examples the IR is defined in prefix notation. Arguments are grouped with brackets. Most operators can have a different amount of arguments. Each symbol is prefixed with either # or $. There are special operators which are shorthand for simple operations. A prefix of # means that the symbol is predefined by the IR and a prefix of $ means, that the symbol is defined by the architecture.

4.3.1 IR-defined symbols

There are four IR-defined types of symbols. Those are arguments, temporaries, flags and operators.

Mapping of VM-operations to IR-translation

To use operations directly in the IR-translation-language the operator can be addressed via the symbols or shorthands from Table 4.6. There are additional special operators in our IR-translation-language making more complex expressions possible

seq Sequentially executes expressions.

if/? If the first expression does not equal 0 execute the second expression otherwise execute the third expression.

rep As long as the first expression does not equal 0 execute the second expression repeatedly

rec[mnemonic] Call the instruction defined by the *mnemonic* with the arguments of the operator

undef Undefines all arguments. The arguments can only be registers.

val Memory operands from an argument are interpreted as a value.

size/bsize The number of bytes/bits of the expression.

Operation	**IR-Symbol**	**Shorthand**
trap	trap	
syscall	syscall	
call	call	
jump	jmp	
return	ret	
greater	g	>
greater equal	ge	>=
less	l	<
less equal	le	<=
equals	eq	==
addition	add	+
subtraction	sub	-
multiplication	mul	*
division	div	
modulus	mod	
and	and	
or	or	
not	not	
binary and	band	
binary or	bor	
binary xor	bxor	
binary not	bnot	
shift right	shr	
shift left	shl	
rotate right	ror	
rotate left	rol	
load	ld	
store	st	
push	push	
pop	pop	
assign	assign	=
append	app	
nop	nop	
carry-flag	c	
overflow-flag	o	
underflow-flag	u	

Table 4.6: IR-defined symbols and shorthands for the IR-translation language

Arguments

The IR is designed, to be parsed with a list of arguments from actual instructions. They are indexed starting with 1. These arguments can be registers, values(signed, unsigned or float) and memory operations(segment, baseaddress + index * scale + displacement). They are set by the disassembler and allow us to share instruction-definitions for multiple version of an instruction. As an example, an add operation in x86 can be used with registers, memory locations and immediate values. Generating a parser for every 14 types of add instructions in x86 is tedious and error-prone.

We can define a simple add instructions without side effects as "=(#arg[1],+(#arg[1],#arg[2]))".

Memory locations are special arguments, which when read are turned into memory reads and if written to turn into memory writes. They contain a base address, a scale, a size and a displacement. These values are used in the following way to calculate the address: *base + size*scale + disp*. Each of those values can be an integer or a register.

When used inside of a value operator(e.g. "#val(#arg[1])") no memory read or write is generated. Then the address is returned as a value. This can be used for example in the lea instruction in the x86-architecture.

Temporaries

As the name suggests, temporaries only exist in the scope of the current IR. Writing to a temporary sets its value and type. When a read occurs, the currently set value and type are used. Temporaries are stored in a simple array and they can be indexed with a number starting with 1.

4.3.2 Architecture-defined symbols

There are four types of symbols that can be defined by the architecture: registers, memory-spaces, stacks and builtins.

- Registers: Registers are expressions in the IR and can be read and written by an assign expression.
- Memory-spaces: A memory-space identifier may only be used for loading or storing expressions as the first argument.
- Stacks: Stack identifiers may only be used for push or pop expressions as the first argument
- Builtins: Builtins are similar to operators. They can have a certain amount of arguments and perform an operation, that extends the instruction set with additional functionality.

4.3.3 Expression modifiers

If an operation only uses a part of an expression, then bit subscripts can be used to access only certain bits. Subscripts have one or two attributes: an offset and an optional size. The offset starts with the value 0 contrary to arguments and temporaries which start with 1. The reason why we number bits differently is a bit of an arbitrary decision: We wanted to differentiate the usage of ids, which always start at 1 with bit-array subscripts.

Bit subscripts are represented by square brackets after the expression. In the square brackets, one or two values split by a colon can be added. As an example the expression *$rax[0,8]* uses the first 8 bits of the register *rax*. Bit subscripts must not be used as the first argument of the assign operator. This means that only writing the whole register is allowed. The offset and the size in an expression can be expressions again but must be constant when evaluated them as described in Section 4.3.4.

Additionally, to the bit subscripts, a type can be added to any expression. This interprets the result of the expression as the given type. We can change the previous example "$rax[0,8]" and interpret it as a float like this: "$rax[f,0,8]", which is not a cast, but a reinterpretation of the bit-array. Casts are done using the *cast* operator (e.g. *cast[s]($rax[u]).* There are three types that can be used u, s, and f which correspond to an unsigned integer signed integer and floating point number.

As can be seen by the cast example in the previous snippet, operators can also have modifiers. There are two additional pieces of information, that can be added to an operator. The type and the size. The size is most of the time not necessary because it gets inferred automatically from the arguments even if all of them are temporaries.

4.3.4 Constant Expressions

When disassembling we get the arguments of the instructions from the disassembler. When generating the SSA-form, a constant expression is an expression, which can be simplified to a specific value at the time of translating the instruction into the IR. Values are trivially constants. Additions, subtractions, multiplications, divisions and modulo operators are constant if the operation has the type *unsigned integer* and all arguments are constants. The *size* and *bsize* operators are always constant, as the bit-length of every expression must be constant at evaluation time as well.

4.3.5 Internal representation

To reduce execution time, we parse the IR only once and build a data structure, that represents the IR-string. This structure can then be efficiently traversed during disassembling for each disassembled instruction.

4.4 IR in SSA-form

The IR, we are primarily using is in SSA-form. It is generated from the IR-translation language during disassembling. We will quickly go over the basic components the intermediate representation is built upon.

4.4.1 Basic Blocks

Functions are rarely a list of expressions, all getting executed sequentially. Branches and loops complicate the control flow. Basic blocks are special constructs, which act as wrappers for a list of expressions to be executed sequentially. Basic blocks themselves are laid out in a directed cyclic graph, where each node is a basic block and each edge is a possible control flow.

4.4.2 Locations

To not lose information about what registers are read or written to, at certain points we need to add additional metadata to expressions and arguments. We call these locations. Locations either refer to a register or a memory-space. Once the SSA-form is built the location are mostly additional information for expressions and have mostly no real impact, with some notable exceptions. Optimizations typically do not use them or can in certain situations remove some entirely. Locations are only important for the semantics of the input, output, call, jump and return expressions and their arguments respectively.

4.4.3 Value/Memory/Void-expressions

Each expression that represents an actual value is a value-expression. The result of said expressions are bit-arrays with a fixed size.

Another type of expression is the memory-expression. The result refers to a memory-space and encapsulates it wholesomely. A store-expression, for example, is always a memory-expression, because it changes the memory-space and creates a new definition which can be used later on. Memory-expressions must always have a location set to a specific memory-space.

Every expression, which does not represent a value or a memory-space is a void-expression. These include control flow expressions such as jump-, return- and call-expressions. They have no return value and should never be referred to by an argument.

4.4.4 Expression-types

The expressions are mostly based on the operations defined by the virtual machine. Here is a short description of all types of expressions in detail:

Label During IR-generation we might be in need to find the start of a specific instruction. Especially when trying to disassemble a new basic block all labels are searched if

the instruction was already disassembled. In that case, we split basic blocks before the occurrence of the label-expression. The label-expressions are inserted for every instruction and specify the point where the IR-expressions for a specific instruction starts. Because every IR-expression holds the address of the instruction it was generated from, we do not need additional information for the label-expression. It must not have arguments. Additionally, these expressions are void-expressions but they are still considered by the dead code elimination. This means they always get eliminated early on.

Undef An expression that specifies undefined results. Any operation with an argument referring to an undef-expression, results in another undef-expression. It must not have arguments.

Nop No-operation. Does nothing. It must not have arguments.

Op Operates on values. They are further divided into add, sub, mul, div, mod, and, or, not, eq, ne, lower, lower-eq, greater, greater-eq, band, bor, bxor, bnot, shr, shl, ror, and rol. It must have at least one value-expression as an argument. The actual amount depends on the type of operation.

Flag We defined three implicit flags to extract additional information from an operation. A flag-expression always refers to an operation-expression. The flags are the same as for our IR. They are carry, overflow and underflow. Carry is special as the size of the argument refers to the index where an overflow occurs.

Append Takes a non zero amount of arguments and combines them into a single bit-array. The size of the expression is the sum of the sizes of all arguments. The first argument will be placed at the lowest bits of the result and the following arguments are appended consecutively. Therefore, the last argument occupies the last bits of the result. The expression can have an arbitrary number of arguments, all referring to a value or a value-expression.

Cast As already mentioned, a cast is special as it takes the value of one argument and casts it to another value. The type of the argument is taken as the source-type and the type of the cast-expression is taken as the target-type. A cast expression may only have one argument referring to a value or a value-expression.

Call Calls in the SSA are special because the control flow actually leaves the function. We assume, that the control flow continues with the expression following the call when the called function has finished. In said function calculations may be done, that read/write registers or store/load data to/from memory. That means for every register that is read and for every memory that is used in a load we need a *use*-argument in the call-expression. Additionally, values calculated in the function might be written to a register or stored in memory. In this case, appropriate defs need to be added for every such register/memory. Since an expression is the same

as an SSA-definition we create special *output*-expressions, which refer to the single registers or memory-spaces via locations.

Built-in An operation we cannot define with the current set of IR-expressions. Similar to call-expressions, each argument needs to be labelled with a register or a memory-space and it also needs output-expressions to mark all changes but it does not use the first argument as a jump-target. A builtin may be defined with a set of registers/memory-spaces that it might use which reduces the number of arguments and output-expressions generated.

Input Functions are not executed in a vacuum. The processor starts in a certain state, which is defined by the calling function or the program loader for the entry point. This means, for every register or memory-space we need one starting definition. The first basic block is always the entry of the function and contains only input-definition. To match inputs with registers or memory-spaces, a location is added to each expression. Two inputs with the same location can in theory appear, but it does not make sense, as the input defines the starting state, which would make them interchangeable.

Output Output-expressions are additions to call-expressions. When control flows returns from a call, we need to create a definition for every register which was written and for every memory-space that was used in a store. An output-expression is created for every register and memory-space and is referred to by the location. The first argument of an output expression is always the corresponding call-expression. An output-expression can also have an optional second argument, which refers to the corresponding register/memory-space argument in the call-expression.

Return A return uses its first argument as the target to exit a function and return the control flow to another function. The rest of the arguments are one for each register and memory-space being changed inside of the function.

Syscall Similar to a call-expression but performing a system call. The first argument is the id of the syscall and the other arguments define the current state of the machine via arguments that are labelled with register or memory locations.

Trap This expression traps the execution of the processor ending the control flow immediately.

Phi Phi nodes are crucial in the SSA-form. They represent the non-gated phi-function. For illustration let us imagine a simple program with two branches that join back together before ending. In the first branch, a register is set to 1 and in the second branch, it is set to 2. The two branches then lead back together into a common basic block, which needs to use the value of the register. Modelling this without Phi nodes this is impossible because if we use the definition from the first branch it might not be defined at the actual use. If the control flows taken during execution is the second branch the use was never generated. To solve this problem Phi nodes

are used to merge values from different incoming control flow edges. Phi nodes are special expressions, that take one definition from every preceding basic block and merge them together into a new definition, which can henceforth be used.

In the case of our example, a Phi node is inserted at the beginning of the basic block where we want to use the register. For both branches, a use is added to the Phi node and the definition of the Phi node is now used as described above. It is important to know that Phi-nodes do no actual work. That means they only merge values together into a new definition. This is their only purpose.

They have exactly twice as many arguments as the basic block has incoming edges. The first of every pair references the basic block the control flow comes from for this edge and the second references the actual use.

Assign An assign is a simple copy of a value to a new definition. They are all removed by dead code elimination.

Split A split expression needs exactly one argument. Besides the size, it has an offset too. It splits the bit-array of the expression referenced by the argument. Both offset and size are used in the split.

Branch Has to be at the end of a basic block and changes the control flow of the function. The first argument is always the default branch-target. The rest of the arguments are pairs of condition and target. The condition is either a value or a value-expression. If the value equals 0 then it evaluates to false, every other value evaluates to true. The conditions are evaluated sequentially and the first branch-target whose condition is true is used. If no condition is true then the default target is chosen. This expression can effectively simulate gotos, if-else branches and switch-case constructs.

Store Stores a value into a memory-space. It must have three arguments: the first refers to a memory-space, the second to an address and the third is the value to be stored. The result of the expression refers to the same memory-space as the first argument, but this time with the value stored.

Load Loads a value from a memory-space. It must have two arguments: The first refers to a memory-space and the second to an address. The result of the expression is the loaded value from the address.

Except when talking about the static single assignment form in general, we will refer to an SSA-definition as an expression and an SSA-use as an argument.

4.4.5 Def-Use-Chains

For dataflow analysis, def-use-chains are important as they are data structures, which track where variables are written to and read from. Normal variables can be written

multiple times so def-use chains can be quite complex especially when aliasing is possible. In the static single assignment form, def-use chains are simple. The lack of multiple writes to variables means that every def-use-chain has only one definition at the very beginning followed by a variable amount of uses.

4.4.6 Memory in the SSA-IR

Using memory in SSA-form is more complicated. The state of the memory-space is represented in the SSA-form as a definition. At the beginning of a function, the memory-space gets initialized with an input-expression like any other register. Load-expressions take a memory-space definition as the first argument. They do not create a new definition as memory is not changed. Store-expressions are different, they create new definitions referencing the new state of the memory-space. At the end of a function, the return-expression references the memory-space, too.

With this representation of memory-spaces in the SSA-form, we can treat memory like other values. This makes it easier but results in less-fine-grained options for analysis. Let us demonstrate this with an example. We assume having two store-expressions that both write to the same memory location and the second store-expression depends on the first. Since both store-expressions write to the same memory location we do not need the first one for the final state. If there is no load-expression using the definition of the first store then it is completely useless for calculation of the final result and will be removed. In most cases this is reasonable but in some situations, it is not applicable. Let us assume, a program maps shared memory. Writing to the same memory location multiple times in a row can be intended functionality because another program is reading from the memory location. The same is applicable for reading from the same location.

Another example: some architectures, like the GameBoy, use memory writes to control hardware. For this hardware, a certain memory region is reserved for controlling rendering, sound, DMA(Direct Memory Access), etc.. Removing memory writes from the program, therefore, would likely lead to misbehaviour. Removing loads is different as loads usually don't produce side-effects and if the loaded value is never used, it can be removed. Reading from a memory location immediately after writing is the most complicated. The value may change between writing and reading, but in most cases, it will stay the same. We assume that the value stays the same, but generally, we do not remove stores because it may produce side-effects. In Section 4.7.2 we will discuss how we can remove loads by determining the value that is to be loaded for special patterns.

Simplifying def-use-chains for memory is still a worthwhile task as it could offer us optimization possibilities. Loading directly after a store is not a common case. The most occurrences of reading the same value that was written are when the compiler spills multiple variables onto the stack or pushes callee-saved registers onto the stack. Between the store and the load, we can mostly expect multiple different stores which change the state of the memory-space. To alleviate this problem we could replace the argument of a store that refers to the state of the memory by the previous if the two states are

functionally the same, which means that a store completely overlaps the previous store. This ensures that there is no discernible effect on the state itself. Nevertheless, it can be problematic in reducing the number of stores from two to one as mentioned previously. If the store changes a state we do not or can not model then this surely influences the behaviour of the program. This must not happen therefore simplifying def-use chains by removing stores is generally impossible without making assumptions about memory behaviour.

4.5 Function-Disassembler

The goal of our disassembler is to have a general algorithm that acts as a framework for disassembling a whole function at a time. The algorithm is recursively descending which means for every call we resolve, we queue the address as next to be disassembled. As already mentioned in the description of the SSA-form, functions are not simply a sequence of instructions, all getting executed sequentially. Conditional and unconditional jumps influence the behaviour of functions, creating a directional graph of control flows. Similar to the SSA-form we generate basic blocks which hold disassembled instructions and are connected by directed edges. We will not go into more detail because this structure exists mostly for manually checking and comparison and has little to do with the decompilation process.

Additionally, an array of arguments is attached to each disassembled instruction, which is used to generate the IR.

Evidently, the algorithm can not disassemble every instruction-set but it is extensible. For every instruction set we just need to write a function, that disassembles a given memory block, generates instructions and the argument array and submits it to the IR-generation algorithm. The algorithm then generates the basic blocks and edges between them. We implemented such a disassembler for x86 based on capstone and one for AVR which we wrote ourselves.

We do not disassemble all executable sections, only functions we can find starting with the entry point. This is quite inefficient, but for an initial implementation, it has proven to work well enough. Especially for AVR, where dynamic dispatch is quite rare, we found all functions without problems.

As already mentioned, conditional and unconditional jumps and similar instructions are important for our algorithm. In the beginning, we added a type to each instruction and jump-destinations which are set by the disassembler itself. This worked quite well, but we decided to integrate the IR-generation into disassembling so that we can use the information gathered during the IR-translation to control function disassembling. This allowed us to simplify disassembling by a lot and reduce its responsibilities by the jump-target resolution.

4.5.1 IR-Generation during disassembling

Submitting an instruction to the algorithm instantly calls the IR-generator. If the IR-generator creates a new return- or branch-expression we know, that a basic block has just finished and we stop disassembling for this path. For conditional and unconditional jumps, the target addresses need to be resolved if possible and the targets are added to a queue to be disassembled next. Additionally, for conditional jumps the address of the next instruction needs to be added from that queue. Return instructions terminate disassembling along the path completely.

Calls work similarly to branch-expressions, as they add the target if possible to the functions to disassemble. The difference is, that we assume the function returning after execution. This means that it immediately continues with the next instruction. For this reason, we do not need to create a new basic block.

Using the SSA-generation to control the disassembling means it depends on how instructions are defined and do not need special information from the disassembler.

4.6 SSA-Generation

Every snippet of an Instruction-IR consist of three parts:

- Number of Argument: how many arguments the IR needs to parse
- Condition-IR: a condition that should be parsed with the arguments of an instruction. The expression should be constant at evaluation time.
- Descriptive-IR: The IR, which is used to transform the instruction into the SSA-form

When generating the IR for every disassembled instruction the number of arguments has to match and the condition-IR has to be evaluable to anything else than 0. Then the descriptive IR is used to generate the SSA-form. Table 4.7, contains a small list of instructions from AVR and their condition-IR and descriptive-IR.

4.6.1 Register writes

Writing to registers is special, as registers can consist of other registers that partly overlap the parent register. Reading from a register that is part of a parent register can be done by generating an argument where the bit-offset and size are set. Writes, on the other hand, are different as they only write a part of the register. In the case of x86 16- and 8-bit registers do not clear the parent registers. 32-bit registers fill the remaining bits of the register with 0. This is the reason why every register has an additional flag that specifies if the parent register gets cleared on a write. If a child register is written and the flag is set, all bits of the parent register, not covered by the child register is cleared to 0. This is implemented by an additional append-expression or an extend-expression in

Instruction	Args	Condition	IR-definition
rjmp	1	-	#jmp(#arg[1])
breq	2	-	#cjmp(#arg[1],$zf,#arg[2])
mul	2	-	#seq(=(#t[1],#mul(#arg[1],#arg[2])), =($r0,#t[1][0,8]),=($r1,#t[1][8,8]), =($zf,==(#t[1],0)),=($cf,<[s](#t[1],0)))
movw	2	==(#bsize(#arg[1]), #bsize(#arg[2]))	=(#arg[1],#arg[2])
	3	==(#bsize(#arg[1]),16)	=(#arg[1],#app(#arg[2],#arg[3]))
	3	==(#bsize(#arg[3]),16)	#seq(=(#arg[1],#arg[3][0,8]), ,=(#arg[2],#arg[3][8,8]))
	4	-	#seq(=(#arg[1],#arg[3]),=(#arg[2],#arg[4]))

Table 4.7: Examples of Instruction-IRs

case only the lowest bits are used by the child-register. If the flag is not set, the original content of the parent register is appended on both sides of the new value by an additional append-expression.

4.6.2 Basic Block Resolving

After the creation of SSA-expressions we need to bind the basic blocks together. We iterate all branch-expressions and replace absolute addresses with the block-ids. During SSA-generation we can not immediately add the block-ids to the branch-expressions. The blocks may not have been generated yet, because the addresses can still be in the queue to be disassembled.

If the jump-target cannot be resolved we have to treat it as a jump to an unknown location. From that point, no further assessment of the statements is possible, so we add all registers and memory-spaces to the branch-expression and end the control flow at that point.

4.6.3 Phi-expression generation

As already mentioned, in the description of the SSA-form phi-nodes are important to combine expressions after two or more basic blocks have been joined together. During the SSA-generation phase, we create the expressions, but if registers or memory-spaces are used, we can not reliably determine the id of a certain expression because not all paths are necessarily already created. Retrieval of such ids is performed as a separate pass. It is split into four phases:

In the first phase, we iterate over all arguments of every expression and remove the id if a location is set. Afterwards, we remove all phi-expressions. This allows us to rerun this pass multiple times. It may look unwise to generate phi-expressions more than once,

but there is an important reason for it: Branch-expressions can still be unresolved at this point. IR-generation follows all jump-addresses backwards and investigates the reason for the jump, which sometimes is unsuccessful especially if dynamic dispatch is used in the program. In some cases investigation will not be successful, this includes standard compiler constructs like jump-tables generated from switch-case expressions. Without actually binding the arguments to the corresponding expressions we have to rely on the analysis of the instructions themselves which is much more complicated than using our already generated IR. To use the IR for this analysis we first need to complete it by generating the phi-expressions and after resolving additional jumps we have to rerun phi-expression generation. This can be repeated as long as new jump targets are resolved.

The next phase is to iterate over all basic blocks and analyse the containing expressions in order of their appearance. For every expression, we gather the registers and memory-spaces and the expression-id of the last definition in a write-set. Additionally, we iterate through the arguments and for every use of a register or memory-space we add it to a read-set in case the register/memory-space is not already in the write-set. If so, we update the argument with the id from the write-set and do not add it to the read-set.

The third phase iterates through all basic blocks again and for every entry in the read-set, we check if for every incoming basic block there is an entry in its write-set. If all previous basic blocks have an entry in their write-set, we generate a phi-expression for the register/memory state. For every incoming basic block having no entry in the write-set, we follow the control flow backwards until we find either a basic block with a corresponding entry in the write-set or another block with two or more incoming edges. If an entry is found we use the id from the write-set for this path. If no entry is found, we got a new *backward branch* (2 paths converge so it is technically not a branch but a convergence) and try to create a new phi node from there recursively. The actual complexity of this step arises out of loops. If we create a phi-node inside a loop, then following the *backward branches* leads back to the basic block we want to create the phi-expression in initially. Because of the *backward branch* the algorithm tries to create a new phi-expression again in the initial block and recurses indefinitely. To alleviate that problem the first thing we do is to insert the new phi-expression and add its id to the write-set of the current basic block if an entry does not already exist. This means if our algorithm returns to a basic block already visited, it will use the id of the generated phi-expression.

For the fourth phase, we just repeat the second phase and all arguments that refer to registers or memory-spaces by location should be resolvable to actual ids.

4.7 Optimization Passes

To improve on the intermediate representation we implemented multiple optimization passes which transform the IR to simplify it without changing its meaning. We present the three passes which had the most impact on the quality of the intermediate representation.

4.7.1 Dead Code Elimination

After the IR in SSA-form is generated and all arguments refer to the correct expressions we discovered, that most expressions are actually not used at all. What we needed, was additional information about each expression where its result is used. We have chosen to add a list containing the ids of the expressions using the definition. We called it the reference-list.

References are easily calculated by iterating over all expressions and their arguments and adding the id of the current expression to the reference-list of every expression that was referenced.

Dead code elimination(DCE) uses the information of the reference-list to efficiently remove all expressions not influencing the final result. Every expression, whose reference-list is empty is not used in any future computation and can safely be removed. The control flow expressions jump and return are expressions, which dead code elimination must never remove. because they implicitly change the state of the program. Additionally, some builtins have side-effects which we can not model. These builtins have to be flagged accordingly and must also not be removed. Any other expression even call-expressions can be removed by dead code elimination.

4.7.2 Register/Memory Section Usage analysis

A call-expression models the passing of the control flow to a specific function of the program, a function of an external library or some other unknown location. For unknown locations, we have to take a conservative approach assuming that every register and memory-space will be read or written to. This means that many arguments are added to the call-expressions and a lot of output-expressions are generated. In this step, we try to find out which registers and memory-spaces are actually used and written to in order to reduce the number of arguments we need in call-expressions and to simplify output-expressions to assign-expressions.

A function always starts with a basic block of input-expressions and ends either with a jump to an arbitrary location or a return. We ignore unknown jump-targets as they are not important for now. Every argument except the first of a return-expression defines the resulting value of a register or memory-space. If the argument references the input-expression of the same register/memory-space, the register/memory-space was not changed. Additionally, if the input is only referenced by return-expressions then the register/memory-space was never read. For the first case, we can replace every output-expression that references a call to this function with a simple assign. For the second case, we can simply remove the corresponding argument from the call-expression for this function.

Because this analysis is rather basic, we added additional functionality: In most architectures, the return address of a function is stored on the stack before or during the call instruction. This address is popped from the stack before returning from a function.

That means between the passing of the control flow and the return, the stack-pointer changes at least by the size of the instruction pointer. For our simple analysis, this means that the stack-pointer will never be resolved if a call-expression appears in a function. To alleviate this problem we added an additional way a register/memory-space might be affected by a function besides reading and writing. This additional state is called an arithmetic change and only refers to registers. Every time a register is changed by a fixed and constant value, the arithmetic change is set.

We calculate the arithmetic change by following the references of the register-arguments of the return-expressions. But we can only follow assign-expressions, additions and subtractions with constant values and append-expressions. The length of the expressions may never change along the path except for append-expressions.

- Assign: We take the first and only argument and follow it.
- Addition or subtraction with constant value: If only one argument is non-constant then we follow this argument and change the resulting arithmetic change according to the operation.
- Append: Append-expressions are special because they use multiple values and "glue" them together. We follow all paths and they all need to resolve to the input-expressions with the correct bit-offsets so that the append correctly puts them back together.

In case a value was written to and the arithmetic change is set, we can now replace the corresponding output-expressions that refer to calls that invoke this specific function with add or subtract expressions. This worked extremely well for the stack-pointer and we could in many cases propagate the change of stack-pointers over many function calls.

4.7.3 Callee saved register optimization

The problem of stores followed by loads at first looks quite trivial to solve with an alias-analysis. An alias analysis tries to isolate areas where reads and writes do, do not or may overlap. With this information, optimizations can reduce the number of reads and writes. As already mentioned, removing a store-load pattern is especially interesting for callee-saved registers and local variables that are stored on the stack. One thing both of them have in common is that they store values on the stack for later retrieval. Another characteristic of such pattern is that both the store and the load reside in the same function. On first glance, it looks like we do not need any inter-procedural analysis, but in the general case, this is not true. We use a technique very similar to what is used in *SecondWrite*[40] to determine arguments that can be optimized away.

Callee saved registers mostly follow specific patterns: An input-expression followed by a store at a certain address calculated by the stack-pointer plus/minus a constant

offset. At the end of the function, there is a load-expression from the same address followed by the return.

It is important to know that the input-expression and the argument of the return that refers to the load have the same register-location and no other store accesses the memory region that the value of the callee-saved register is at. Callee saved registers are special in that they are not changed within the function so we need to get to the input expression unchanged. Additionally, we check if another function was called with the pointer of the callee-saved register as an argument. In that case, we have to assume that the value has been changed in that called function and we can not eliminate the parameter and return value. For checking if the pointer was passed to another function we compare all arguments to the pointer.

The current analysis makes important assumptions about stack behaviour: Aliasing cannot happen for the current stack frame. That means if we compare the addresses of two store-expressions and one refers to the stack-pointer plus/minus an offset then the other address has to refer to the same stack-pointer plus/minus the same offset. If we do not make this assumption our algorithm is absolutely paralyzed in case any store-expressions appears that does not write explicitly onto the stack. Also, bugs in the implementation can completely throw off the whole analysis.

To illustrate this problem let us assume a function with callee-saved registers. This function calls a second function which has an arbitrary length stack overflow bug. The analysis can not just assume that a function does not overwrite the stack frame of calling functions. At least not callee-/caller-saved registers. So it has to assume that this possibility exists. In our case, it does exist. Therefore the analysis now reports that the location where the callee-saved register is located could possibly be overwritten. Therefore we can not optimize the register away. This is not good. Functions do not typically overwrite callee-saved registers on different stack frames. This is an explicit assumption that we made in this analysis. It can theoretically happen that we run into callee-saved registers that get overwritten in called functions. It might even be possible that an obfuscator deliberately uses such patterns. That is possible and then this optimization would give an erroneous result. This is an assumption that we have to make otherwise analysis on the stack is nearly impossible.

Another problem are arguments that are arithmetically changed during a function call. A good example is the stack pointer. On most architectures, a return value is pushed onto the stack before jumping to the new function and pushed from the stack before the function returns. This means that the stack-pointer is changed by the number of bytes that the instruction-pointer has during such a function. Normally the pointer at the end of the function is always the same no matter what control flow path is taken. For recursive function-calls we can not calculate all paths as at least one path contains the recursive call which we can not resolve as we are currently in the process of resolving it. Therefore we are dependant on the solution to the problem if we want to solve the problem. We solved the issue by assuming the result if it is available. Basically what we do is we try

to follow the value of a register from the return-expression to an input-expression. In case we can reach the input-expression then we save it in the result. If we reach the input-expression so that the path does not equal to the previous solution(a different arithmetic change to the register for example) or we can not reach the input-expression then the algorithm can not find a correct solution. If a recursive call is detected then we look at the solution that we have already found. If a solution was found we assume that it is correct and act like the call-expression changes the value accordingly. This works well in case a result was already computed in one path. If we reach a recursive-call in the first path we that we are looking at, we can not have found a solution yet and can not resolve the value of the register over the call. Another, later path might still resolve the value correctly so this naive assumption is not enough to solve all cases even though they might be solvable. To still find all solutions we made a small change based on a fact about our intermediate representation. The paths data travels on through the function are merged at phi-expressions. That means if we look at the data flow in reverse like we are doing in our analysis, phi-expressions are the only points where paths are split into multiple paths. Therefore if we reach a phi-expression we try all branches. If all succeed we have found a solution. If the first did not succeed we try the rest of the branches and then we try all branches again. This makes sure that if at least one branch finds a solution in the first iteration then recursive calls in other branches can be resolved at worst in the second iteration. If no solution can be found after the second iteration then we can not find a solution even if we iterating one more time. It is possible to construct a function where a recursive call might appear in every branch. Our algorithm will not find a solution for such a case, because there is no real solution. We might be able to guess a value and find a solution. That would work but a function that calls itself in every branch indefinitely recurses and will never return. Therefore it is impossible to resolve a value because the return-expression will never be reached.

Handling Recursive Functions

We use the same algorithm to also simplify not used arguments for simple recursive functions. The problem that occurs in those cases is that registers that are passed and not used are passed straight into the recursive call. Because a call means we leave the control flow of the current function the recursive call expression needs all registers and creates output expressions for every register. That means these registers are seen as normal arguments to the recursive call. Also, the registers cannot be removed from the return as they are possibly written to in the recursive call.

To solve this problem we added an additional condition into our pattern: If the returned value from a register refers to an output expression we look at the associated call expression. If the call refers to the current function that we are in and the output-expression refers to the same register-location then we assume that the recursive call expression does not modify the value of the register. This assumption can be made because if it holds then our algorithm can follow the value back to the corresponding input-expression then it the function does not modify the value and the recursive call does

not modify the value as well. This would mean we can optimize away the return-argument. If the assumption does not hold then our algorithm has to find a path where the value gets changed and our algorithm will find no solution. Therefore the return-argument will not be optimized away.

The described code only removes return-arguments of non-changed registers for recursive functions. The input-expressions do not get optimized away because they are still in use at the recursive call-expression. To handle this case we look at all usages of the input expression and in case it is used in a recursive call-expression with the same register then the value is just passed through. If there is at least one use outside of the call-expression then the expression is also used as part of the functionality of the function itself and we can not remove the value. In case an input-expression in a recursive function is deemed to not being used then we replace the input-expression with an undef-expression. This removes the register from the argument list and it can be effectively removed from the call-expression with the optimization mentioned in Section 4.7.2.

This improvement does only work with simple recursions as only one function is looked at a time. In case we have a 2-function recursion where function A calls function B and function B calls function A this analysis is not enough. We can improve on the algorithm to include multi-function recursions by the handling of the call-expressions recursively. Basically when a call-expression to a different function that we have not visited appears in the def-use chains then we run the same analysis on the register for the called function. This might be possible but at this point, we have not implemented such an inter-procedural analysis.

Backwards Register/Memory Section Usage analysis

Using this usage analysis we can simplify a lot of call-expressions. Still, a lot of registers are tracked by the return-expressions in the functions. To simplify those further we implemented an inter-procedural usage analysis. For every function, we found we searched for all call-expressions and counted for each function for each register/memory-space if the output-expression remains or if the dead code elimination removed it. If it was not removed then we add the register/memory-space to a used-set. After all, functions are analysed we now know for each function what results are relevant to a calling function. With this information, we can now remove every argument from the return-expression referring to a register/memory-space which does not exist in the used-set of the function. This further allows the dead code elimination to remove calculations that are not relevant to the final result of the program. While this reduces the number of values in the return statement if we miss out on functions to analyse we might also miss out on possible return values and we do not want that. We would actually remove return parameters that might be used somewhere. We, therefore, did not remove the parameters but rather hid them in the final pseudocode generator. This means we lose a bit of optimization potential but we retain correctness.

Especially for binaries using dynamic dispatch, we might miss connecting calling sites with the actual functions. This might lead to exactly what we described earlier where

return values might not be recognized correctly. We found that embedded AVR-firmware written in C rarely have dynamic dispatch. We specifically looked at a number of AVR-binaries which had no dynamic dispatch at all. This allowed us to use the backwards usage analysis which eliminated a lot of return values from functions.

4.7.4 Peephole Optimizer

Our main optimization pass is the peephole optimizer. We implemented a matching algorithm. The algorithm matches basic structures of expressions. If a match occurs it lets a more sophisticated function(the executor) take over. The executor can then do more sophisticated matching rules and can implement the desired transformations to the IR.

The matcher has a list of rule-sets. Each rule-set has a number of rules which each describe exactly one expression. The ids of the matched expressions are gathered in a list so that the executor can access them again. Our rule-matcher iterates over each expression of the SSA-form and tries to match each rule in order. If all rules match then the executor is called. A rule consists of two parts:

- The first part fetches an expression. It uses an expression-index and an argument-index.

  The first is the expression-index. It fetches an already matched expression starting with the index 1. If the expression-index equals 0 then the first matched expression is fetched. If not enough expressions were matched then the rule is broken and the rule-set will never find a match.

  The second is the argument-index. It also starts with 1 and indexes the arguments from the previously fetched expression. If the expression does not have enough arguments then the rule fails. If the argument exists but it does not reference another expression then the rule also fails. If the argument-index equals 0 then all arguments are fetched one after the other and a match is generated for each one. If an argument refers to an already matched expression this argument is skipped. If the argument does not equal to 0 then the expression to the corresponding argument is fetched.

- The second part checks if the type of the fetched expression matches. The type can be any SSA-expression including an operation or a flag. The type may also be the invalid-type then any expression is matched.

Each rule is matched in the order they were added to the rule-set and for each matched exception its id is added to a list which is passed to the executor so it may do a more fine-grained selection. Additionally, the executor must return true in case it changed anything in the IR or false if everything stayed the same. The first rule to each rule-set must have the expression-index and argument-index set to 0. This selects the base-expression in the iteration.

Our rule-matcher iterates through all expressions. We iterate the expressions in no particular order. If at least one rule-set matches and the executor returns true then we rerun all rule-sets for this expression. This might seem excessive but sometimes an executor can change the current expression which means different rule-sets might suddenly match again. Also if one or more rule-sets match over the course of the whole iteration then we iterate over the expressions again. The reasoning behind this approach is that executors can change any expressions in the IR. They might also change the expressions that were already iterated over. To get the most amount of matches we have to, therefore, iterate over all expressions as long as rule-sets still match.

Optimization Rules

We implemented a short set of rules of which we will present a few that showed a big impact on the overall result. We have implemented a large amount of other small rules that each improved the result but we will not list all of them. Just the most important once.

Assign-reducer The SSA-generation creates a large number of assign-expressions. For every argument that references the assign-expression the executor tries to replace it with the argument of the assign.

Append-reducer If an argument of an append refers to another append then we can simplify them. Let us assume we have two append-expressions. The first one represents the write to the *al*-register in x86 and the second represents the write to the *ah*-register. The first append, therefore, has two arguments. The first has a length of 8-bit and represents the new value for the *al*-register. The second argument has a length of 56-bits and it represents the current value of the *rax*-register which is the parent-register of both *al* and *ah* with an offset of 8.

The second append-expression has three arguments. The first argument references the first append-expression but only the first 8 bits. The second represents the new value of the *ah*-register. The remaining 48 bits of the *rax*-register is represented by the third argument and references the previous append-expression with the offset 16.

As can be seen, by this example writing to different parts of a register create append-expressions. If different parts of a register are written to consecutively we can replace the arguments referencing the previous append with the arguments of the previous expression.

In our example, we can replace the first and third arguments of the second append-expression. The first gets replaced by the new value of *al* and the second gets replaced by the original value of the *rax*-register. Notice that the references to the first append get removed completely and in case this is the only place where it is used then the dead code eliminator can remove the first append-expression completely.

This rule is useful for registers such as the *rflag*-register in x86 because it holds nearly all flags of the CPU. This means most operations set some of the bits in said

register separately. This creates a large number of append-expressions. With this rule, those appends get simplified by a lot.

Multi-Register Operations In an architecture like AVR, the CPU only supports arithmetic operation with 8-bit registers. If we want to compile a simple C-program that adds two int variables we need more than just simple 8-bit adds because ints are defined to be at least 16-bit. It is quite normal for architectures to support carry-flags. They indicated if the value overflows the bits in an operation. Additionally, additions that also add the mentioned carry-flag can also be seen on many architectures. A 16-bit add operation can, therefore, be simulated with one 8-bit addition and one 8-bit addition with carry. Having two such instructions means that we generate at least three expressions in our SSA-form: two additions and one carry-flag-expression. To combine the expressions into one we implemented rules for additions and subtractions.

Jump Condition Simplification In a lot of architectures, flags are used to control branching operations. For example, a subtract operation that does not store the result is often used to calculate conditions such as lower than, equals and bigger than. This instruction is in many architectures like x86 called the *cmp*-instruction. Implementing conditions with the *cmp*-instruction and the zero-, overflow- and signed-flag are quite simple. Here are a few examples:

Signed Compares Signed compares are often implemented by comparing the zero-, signed- and overflow-flag.

- **Smaller** The first argument of the subtract operation is smaller than the first if the signed and overflow flags are not the same.
- **Smaller Equals** The first argument of the subtract operation is smaller or equals than the first if the signed and overflow flags are not the same or the zero flag is set.
- **Greater** The first argument of the subtract operation is greater than the first if the signed and overflow flags are the same and the zero flag is not set.
- **Greater Equals** The first argument of the subtract operation is greater or equals than the first if the signed and overflow flags are the same.

Unsigned Compares Signed compares are often implemented by comparing the zero- and carry-flag.

- **Smaller** The first argument of the subtract operation is smaller than the first if the carry flag is set.
- **Smaller Equals** The first argument of the subtract operation is smaller or equals than the first if the carry- or the zero-flag is set.
- **Greater** The first argument of the subtract operation is greater than the first if the carry- and the zero-flags are not set.

Greater Equals The first argument of the subtract operation is greater or equals than the first if the carry flag is not set.

We implemented peephole-rules for some of these patterns to simplify the operations and make them more readable in the final result.

4.8 Transformation to Pseudocode

Our last stage for our decompiler is the transformation into actual pseudocode. We decided to not target C or C++ as the final language as it is just too restrictive. While using those programming languages would have some advantage it restricts us too much in what we can actually represent. Advantages of C would be that there exist a lot of analysis programs that work with C as input and can be used to find problems in the resulting code. Another advantage is that C is familiar to most programmers. The major problem, on the other hand, is that C assumes that functions return only one value. To preserve correctness we have to assume that all registers are possible return values which are tricky to display in C. Also memory-spaces do not exist in C. Special constructs would be needed to display memory read or writes for all memory-spaces especially if there can be more than one. Therefore we defined our own pseudo code which has many similarities to C.

4.8.1 Structure reconstruction

We based our control flow structure reconstruction on the goto-less[47] algorithm.

As described previously the control flow in our IR is defined by a directional graph of basic blocks. That means if we want to reconstruct proper control flow constructs like loops or branches we only need to work with the basic blocks.

Simplifying the control flow graph

The control flow graph(CFG) up until now could look any way because no transformation did take its structure into account. The only analysis that really works with the control flow graph is the phi-expression generation but it only cares about *backward branches* and does not need a special format for the CFG. For the pseudo-code generator we changed the CFG so that it contorts to the following rule:

Every basic block must have either exactly one input edge or exactly one output edge.

With this rule blocks can either merge multiple paths together(merge-blocks), can branch into multiple paths(branch-blocks), have no previous blocks(start-block), have no next block(end-blocks) or just pass the control flow along(sequence-blocks). There can only be one start-block and it is the entry point into the function.

To transform the control flow graph according to our rule we simply look through all basic blocks and in case one does not fit our rule we add an empty block in front of the nonconforming block. All input edges get pointed to the new block which gets an unconditional branch to the original block. Even the end-block might get split if it has two or more incoming edges.

Building up a hierarchical representation

Our structure reconstruction algorithm is based upon the observation, that every control flow can be modelled with just three different constructs:

- If a basic block has two or more outgoing control flow edges, then it is a branch, which translates to an if/else or switch/case construct. The head node of the branch is the branching basic block.
- If a basic block has two or more incoming edges and following the edges backwards leads to the original block, then we have found the head node of a loop. All blocks, that lead back to the starting block are part of the loop as well.
- If a basic block is neither a branch or a loop, then it is represented as a sequence. It can be combined with other sequences if the first sequence has only one outgoing edge, which leads to the second sequence and the edge is the seconds' sequence only incoming edge.

Each construct consists of one head block and multiple other contained blocks. Each block that exists in addition to the head block must be dominated by the head block. Additionally, we add the outgoing edges and how often they appear. We define the main-exit as the outgoing edge that is used the most.

To shortly describe what is commonly referred to as domination in a directed graph: A node X in a directional graph is dominated by a node Y if every path leading to node X has to go through node Y. The dominator frontier of a node Y is the set of nodes where each node has a predecessor that is dominated by Y but is not strictly dominated by Y itself. We defined an additional term: the merging dominator frontier. The merging dominator frontier of node Y is the set of nodes that either is the earliest node X that is dominated by each node on the path from node Y to X or if no such node X exists is simply the normal dominator frontier.

Hierarchy of structures

Branches and loops can be nested arbitrarily in programs. We may find triple nested loops with seven nested branches in the innermost loop. To correctly find those constructs we rely on a hierarchy. Every construct has next to the head-block a set of different blocks that it owns. The head-block is the entry point into the control flow structure.

A branch owns all blocks until the control flow merges back into one singular block or another block appears that is not dominated by it. A loop is the same. A sequence is slightly different. A sequence has a head-block but every other owned block has to be laid out in a sequence. Every block except the head-block has to have only one incoming- and outgoing-edge. Also, a sequence can not have child structures.

Branch and loop structures can have children. If they own one or more blocks that are not the head block then each of those blocks has to be part of a child structure. This forces each block to be the head node of either a branch or loop or be part of a sequence.

Finding a Loop Body

Loops are special because the control flow reaches back to the head-block. That means that the head has to have at least two incoming edges which make finding possible candidates easy. We use this fact when searching for loops. We use two stages for the loop-finder:

- First, we recursively follow the control flow of the head-block candidate and mark every block. If the block is already marked or is not part of the parent structure, we stop.
- Then we recursively follow the control flow backwards from the head-block and add every block that has been marked during the first scan to a set. We do not start at the head-block directly, but the predecessors. If a block is already in our set or is not part of the parent structure, we stop the recursion.

This simple algorithm returns a set of blocks which are part of the loop body. If we want to detect nested loops we have to be careful. Once we have found a loop and find a nested loop candidate we have a problem because our recursive marking algorithm follows all paths and will also follow the parent loop head backwards. This trivially marks the whole parent loop, which we do not want. Therefore we always stop marking when we reach the head-block of the parent loop.

Extending our Constructs with the Merging Dominator Frontier

Because control flow structures in a program can be nested, for every control flow structure we need to somehow detect its whole scope. At this point, our branch only owns the head block and a loop only holds the blocks where it is possible to reach back to the head-node.

For example, let us assume that a loop has two exits from its main body. The first exit is the regular exit, while the second leads to an intermediate block, which then jumps to the regular exit. This block is not part of the main body of the loop. Including only the main body results in missing out on pulling the alternate exit into the loop,

which means we have to use at least two gotos when generating the pseudo code because there is no other way to display arbitrary loops with two or more exits. To solve this, we use our definition of the merging dominator frontier. To be precise for every control flow structure we look at the exits. If an exit-block is not part of the merging dominator frontier then we pull it into the original control flow struct and add it to the owned blocks.

We illustrate this with a few simple cases:

- If the control flow struct has only a single exit-node then this exit is part of the merging dominator frontier and we stop. This means we have found a regular exit.
- If it has multiple exit-nodes then we look at the incoming edges of the exit-nodes.
  - If an exit node has an incoming edge that is not part of the control flow struct then it is possibly a part of the merging dominator frontier and we can not merge the block into our structure.
  - If all incoming edges of an exit node lead into the control flow struct then it is merged into the structure because it is not part of the merging dominator frontier.

This has to be done repeatedly until there is no more exit node to merge.

4.8.2 Pseudo code generation

The final step of our decompiler-prototype is the pseudocode generator. As for the target language, we decided not to use C or C++ because we quickly realised that we had too much information to display and using C/C++ will not be appropriate. While C has a lot of advantages as it is a widely known language and is still heavily used. Also, there are many source-level tools that analyse C-code directly. Representing concepts such as virtual method tables in C, on the other hand, might be quite confusing. With C++ such concepts can be represented better and the fact that C++ can be seen as a more feature-rich C means there are no big trade-offs in using C++. As already mentioned we have a lot of information that we can not optimize away easily which makes both C and C++ not the best candidate for the final language. We finally decided to use custom imperative language which is heavily influenced by C. As of now the control flow reconstruction is the only transformation we used to make the pseudo code better readable. The remaining part is a basic representation of the IR.

For every expression that is used more than once, represents a variable or is a load/store, we create a pseudocode-expression. If an IR-expression is only used once it gets resolved as part of another expression.

Unification of Variables

Phi-expressions are treated specially. We use them to create variables. For every Phi-expressions we create a data structure that represents a variable. Each such structure holds a set of expression-ids. At the start, only the id of the phi-expression is in the set. Then we add all expressions to that set that are referred by that phi-expression. If one of the ids is already a member of another set, we unify both variable structures by combining both sets into one.

If an expression is emitted and its id is in a variable-set, in addition to the write to a temporary we also add the write to the variable.

Store/Loads

For stores and loads, we always generate an expression in our pseudocode. They always have a form like:

- Store: dmem[u8 , (tmp1 + 0x0)] = 0x54
- Load: u8 tmp35 = (dmem[u8 , (tmp23 + 0xd)])

In this example “dmem” refers to the memory-space written to or read from. “u8” refers to the type used in the example which is an unsigned integer with the length of 8 bits. The second parameter in the square brackets is the address that is used in the load/store.

CHAPTER 5

Results

Our first implementation aims to be rather conservative in its approach. There are some optimizations based on assumptions that in special cases may produce erroneous results but we limited them to a minimum. One of such optimizations are the callee saved register detection where we assume a certain pattern means we can optimize away the load expressions. We try to be as accurate as possible without losing any information about the data flow in the program, therefore the results are not especially impressive the larger a program becomes. The reason for this is that large programs often contain many functions. Every new call-depth increases the risk to miss optimization possibilities. That means a function that calls no other function is optimized best and the closer we get to the entry point when traversing the function graph (the directional graph depicting function calls which other function) the optimization quality decreases.

At the moment, we focused on decompiling AVR programs. They are a good first step because they are limited in scope and self-contained. Focusing on a less complicated architecture first allows us to rethink the basic concepts without having to implement a vast amount of optimizations and analyses in order to get a half-decent result. As already described in the previous chapter, we implemented an AVR and a basic x86-frontend that generates our intermediate representation. In the IR, we then optimize to make the generated code as easy to read as possible without losing out on crucial information which is needed to understand the original behaviour of the program. We have implemented a base set of optimizations which brought massive improvements to the IR. We did as of this point no survey over large amounts of binaries to look at generated code patterns rather we focused on a small number of binaries and analysed them for common patterns. Finally, we implemented a relatively simple pseudo-code generator that would output readable code with proper control flow structures.

Our decompiler currently can handle x86-64 ELF-binaries. For AVR we implemented an ihex parser which is a simple file format to store smaller firmware images. For now, our

frontend assumes all ihex binaries being AVR. The AVR-frontend was easy to implement and took us about a week while refining the code to support multiple architectures and file formats. The base x86-frontend took a while longer mostly because of the sheer ludicrous amount of instructions which exist in x86 and because it was the first instruction set, we integrated and implemented the instruction representation at the same time.

As mentioned previously, the discovery rate of found functions in ELF-binaries was not particularly large, because we purely relied on the recursive descending parser from the entry point onwards. Nevertheless, as long as no dynamic jumps occur we can find all used functions. Dynamic jumps obviously drop our discovery-rate by the number of dynamic jumps/calls being made. For example, a C++ program which extensively uses inheritance with virtual functions results in a poor discovery rate for now. Additionally, we can not optimize away arguments for functions because the dynamic nature of jumps does not allow us to use the function signature of the called function for optimization.

5.1 Comparison with other Decompilers

As we have focused mostly on the AVR-architecture it is quite difficult to compare our results with other available decompilers like snowman or RetDec. These decompilers support mostly mainstream architectures like x86 and ARM, so a direct comparison will not be completely accurate. To facilitate comparisons, we also take a look at the results for our x86 back-end, but it is not as optimized as the AVR version. We use the x86 parsers of snowman (version 1.3) and RetDec (version 3.2) to generate decompiled code and use this as our base cases and compare them to first the AVR back-end and then the x86 back-end.

Our test cases consist of a limited number of small programs that were decompiled with our solution, snowman and RetDec. We focused mostly on small programs to get a manageable output. We chose two simple functions of which we will compare the results. The first test-case we used is an iterative, the second is a recursive implementation of the well known Fibonacci sequence. We already pointed out the limits and problems of the current implementation at the appropriate places in this document. They have to be taken into account for the tests.

5.2 Fibonacci Iterative

The iterative Fibonacci example in Listing 5.1 consists of a simple function that calculates the i^{th} value in the Fibonacci-sequence. The calculation is done iteratively in a loop.

```
uint32_t fibonacci(uint32_t val) {
uint32_t first = 0, second = 1;
if(val == 0)
    return 0;
if(val == 1)
    return 1;
for(uint32_t i = 1; i < val; i++){
    uint32_t temp = first + second;
    first = second;
    second = temp;
}
return second;
}
```

Listing 5.1: C-Source of iterative Fibonacci

5.2.1 AVR

We compiled the code with the AVR-version of GCC. We used *-O1* as optimization. The generated assembly can be found in Listing 5.2. We also compiled with -O3, but the result was not much different, because the function is quite trivial to compile.

```
0x00000028  push r8
0x00000028  push r8
0x0000002a  push r9
0x0000002c  push r10
0x0000002e  push r11
0x00000030  push r12
0x00000032  push r13
0x00000034  push r14
0x00000036  push r15
0x00000038  push r16
0x0000003a  push r17
0x0000003c  mov r8, r22
0x0000003e  mov r9, r23
0x00000040  mov r10, r24
0x00000042  mov r11, r25
0x00000044  or r22, r23
0x00000046  or r22, r24
0x00000048  or r22, r25
0x0000004a  breq 0xac
0x0000004c  ldi r24, 0x02
0x0000004e  cp r8, r24
0x00000050  cpc r9, r1
0x00000052  cpc r10, r1
0x00000054  cpc r11, r1
0x00000056  brcs 0xb6
0x00000058  ldi r20, 0x01
0x0000005a  ldi r21, 0x00
0x0000005c  ldi r22, 0x00
0x0000005e  ldi r23, 0x00
0x00000060  ldi r24, 0x01
0x00000062  ldi r25, 0x00
0x00000064  ldi r26, 0x00
0x00000066  ldi r27, 0x00
0x00000068  mov r12, r1
0x0000006a  mov r13, r1
0x0000006c  mov r14, r1
0x0000006e  mov r15, r1
0x00000070  mov r19, r15
0x00000072  mov r18, r14
0x00000074  mov r17, r13
0x00000076  mov r16, r12
0x00000078  add r16, r24
0x0000007a  adc r17, r25
0x0000007c  adc r18, r26
0x0000007e  adc r19, r27
0x00000080  subi r20, 0xff
0x00000082  sbci r21, 0xff
0x00000084  sbci r22, 0xff
0x00000086  sbci r23, 0xff
0x00000088  mov r12, r24
0x0000008a  mov r13, r25
0x0000008c  mov r14, r26
0x0000008e  mov r15, r27
0x00000090  mov r27, r19
0x00000092  mov r26, r18
0x00000094  mov r25, r17
0x00000096  mov r24, r16
0x00000098  cp r8, r20
0x0000009a  cpc r9, r21
0x0000009c  cpc r10, r22
0x0000009e  cpc r11, r23
0x000000a0  brne 0x70
0x000000a2  mov r22, r16
0x000000a4  mov r23, r17
0x000000a6  mov r24, r18
0x000000a8  mov r25, r19
0x000000aa  rjmp 0xbe
0x000000ac  ldi r22, 0x00
0x000000ae  ldi r23, 0x00
0x000000b0  ldi r24, 0x00
0x000000b2  ldi r25, 0x00
0x000000b4  rjmp 0xbe
0x000000b6  ldi r22, 0x01
0x000000b8  ldi r23, 0x00
0x000000ba  ldi r24, 0x00
0x000000bc  ldi r25, 0x00
0x000000be  pop r17
0x000000c0  pop r16
0x000000c2  pop r15
0x000000c4  pop r14
0x000000c6  pop r13
0x000000c8  pop r12
0x000000ca  pop r11
0x000000cc  pop r10
0x000000ce  pop r9
0x000000d0  pop r8
0x000000d2  ret
```

Listing 5.2: AVR assembly of the iterative fibonacci implementation

holodec

Looking at the generated code for the iterative implementation of the Fibonacci-algorithm the first obvious problem is the great number of arguments that remain for the Fibonacci-function (*func_0x28*). Most of these registers are callee saved registers. We learn this from the fact that they are just written to memory but only ever read at the end of the function shortly before the return-instruction and the read value is never used. Also, we can detect that there is no obvious read from the stack during a function call, so we can optimize away the read. We did not remove the writes because the memory which is written to might be special memory with side effects and we do not want to remove such writes. For the stack, this is generally not the case, but in the spirit of not assuming any platform or compiler related conventions, we can not make this assumption. What we could do is to optionally hide those writes.

After all the caller saved registers are stored there is a branch which combines

four input registers together. The C-Code uses *uint32_t* type so we basically force the compiler to use 32-bit values. If we chose the normal *int* C-datatype it would most likely use 16-bit, instead of 32-bit values. This is allowed in C as an *int*-type is defined as containing at least 16 bit. The four registers are *r22-r25*. According to the calling convention that is used by avr-gcc these registers are used to pass an argument which is 32 bit wide. The arguments are bit-shifted into the correct position and then combined together by a bitwise or. We used the commonly understood C-syntax for combining the values instead of a custom syntax/function.

The resulting value is compared to the value 0 which corresponds to the first branch of our initial function. In the true-branch five values are set. These values correspond again to registers *r22-r25*. As before, according to the avr-gcc calling convention these registers correspond to a 32-bit return value. The last variable that is set is temporary. That means, it may or may not be used. They directly correspond to expressions in our IR, while all variables starting with *var* correspond to phi-nodes either via their id in the IR or a register prefixed by "_". At the end of the control flow block, you can see a list of variable writes. These variables correspond to all registers that were written to during the function and are used later in the return statement.

We can see the same pattern in the next code-block. The result is 1 if the argument is less than 2. These are typical Fibonacci early exits. Up to here, everything is quite understandable, but maybe a bit too verbose.

The else-block of the inner branch is much more interesting. We can see three 32-bit variables being set. Two of them are set to 1 and one is set to 0. *var2* is the loop variable and the rest are the variables first and second. If we follow the loop variable we can see that it is written to in the loop body. The variable is decremented by 0xFFFFFFFF. This constant number is the same as -1 in twos-complement notation so, in fact, the operation is an increment by 1. The reason for this syntax is optimizations made by the compiler because subtracting all four register-parts by the same value(0xFF) has a smaller code-size than adding one register by 0x01 and three by 0x0.

We can see a lot more calculations done with the loop variable. These calculations are here for completeness because they are later used in the return. After the loop variable was written it is compared to the argument and if equals the loop is finished. The first and second variables are mapped to *var5* and *var6* and additionally stored in the temporaries *tmp52* and *tmp82*. Those correspond to the phi-expressions inside the loop. Stripping all unnecessary expressions generated for the return statement and the unused labels the loop would look like in Listing 5.3.

```
Input (arg15 <- r22, arg16 <- r23, arg17 <- r24, arg18 <- r25 ){
        u32 var2, var6, tmp87 = (0x1)
        u32 var5, tmp4 = (0x0)
        u32 tmp29 = var2
        u32 tmp52 = var5
        u32 tmp82 = var6
        loop {
            u32 var6, tmp47 = (tmp52 + tmp82)
            u32 var2, tmp38 = (tmp29 - (0xffffffff))
            u32 tmp29 = var2
            u32 tmp52 = var5
            u32 tmp82 = var6
            if((((arg15 | arg16 << 8 | arg17 << 16 | arg18 << 24)) != tmp38)) {
                continue;
            }
            else {
                break;
            }
        }
}
```

Listing 5.3: Cleaned up AVR-result of holodec-decompiler

This result is actually quite good, but there is room for improvement. Function type reconstruction would be able to group registers together into single arguments and would reduce the number of return arguments. This could further improve quality but was not in the scope of our work.

The rest of the generated output is quite straightforward. A list of register writes for the final return at the end. Notice that the return is actually reading the return-address from the stack and then jumps there, so we model the inner workings of a return instruction correctly. The plus one in the memory pointer argument may look odd for some readers, but the stack always points to the next free byte. A push on most architectures post-decrements the stack-pointer and a pop pre-increments it. Therefore, the return address is stored at the stack pointer plus 1.

Listing 5.4: Iterative Fibonacci decompiled by holodec(AVR)

```
Input (arg1 <- r8, arg2 <- r9, arg3 <- r10, arg4 <- r11, arg5 <- r12, arg6 <- r13, arg7 <- r14,
        arg8 <- r15, arg9 <- r16, arg10 <- r17, var_r18, arg11 <- r18, var_r19, arg12 <- r19,
        var_r20, arg13 <- r20, var_r21, arg14 <- r21, arg15 <- r22, arg16 <- r23,
        arg17 <- r24, arg18 <- r25, var_x, arg19 <- x, arg20 <- sp, ){
    Label L2:
    (dmem[u8 , arg20] = arg1)
    (dmem[u8 , (arg20 - (0x1))] = arg2)
    (dmem[u8 , (arg20 - (0x2))] = arg3)
    (dmem[u8 , (arg20 - (0x3))] = arg4)
    (dmem[u8 , (arg20 - (0x4))] = arg5)
    (dmem[u8 , (arg20 - (0x5))] = arg6)
    (dmem[u8 , (arg20 - (0x6))] = arg7)
    (dmem[u8 , (arg20 - (0x7))] = arg8)
    (dmem[u8 , (arg20 - (0x8))] = arg9)
    (dmem[u8 , (arg20 - (0x9))] = arg10)
    if(((((arg15 | arg16) | arg17) | arg18) == (0x0))) {
        Label L7:
        u8 var_r25, var_r24, var_r22, var_r23, tmp93 = (0x0)
        u16 tmp97 = var_x
        u8 tmp99 = var_r25
        u8 tmp103 = var_r24
        u8 tmp107 = var_r23
        u8 tmp105 = var_r22
        u8 tmp95 = var_r21
        u8 tmp34 = var_r20
```

```
        u8 tmp32 = var_r19
        u8 tmp28 = var_r18
    }
    else {
        Label L3:
        if((((arg15 | arg16 << 8 | arg17 << 16 | arg18 << 24)) < (0x2))) {
            Label L8:
            u8 var_r22, tmp33 = (0x1)
            u8 var_r25, var_r24, var_r23, tmp30 = (0x0)
            u16 tmp97 = var_x
            u8 tmp99 = var_r25
            u8 tmp103 = var_r24
            u8 tmp107 = var_r23
            u8 tmp105 = var_r22
            u8 tmp95 = var_r21
            u8 tmp34 = var_r20
            u8 tmp32 = var_r19
            u8 tmp28 = var_r18
        }
        else {
            Label L4:
            u32 var2, var6, tmp87 = (0x1)
            u32 var5, tmp4 = (0x0)
            u32 tmp29 = var2
            u32 tmp52 = var5
            u32 tmp82 = var6
            loop {
                Label L10:
                Label L5:
                u32 var6, tmp47 = (tmp52 + tmp82)
                u8 var_r19, var_r25, tmp44 = ((u8 var6 >> 24))
                u16 var_x, tmp35 = ((u16 var6 >> 16))
                u8 var_r18, var_r24, tmp53 = ((u8 var6 >> 16))
                u8 var_r22, tmp58 = (u8 var6)
                u8 var_r23, tmp57 = ((u8 var6 >> 8))
                u8 var_r20, tmp84 = ((u8 tmp29) - (0xff))
                u8 var_r21, tmp42 = ((u8 ((u16 tmp29) - (0xffff)) >> 8))
                u32 var2, tmp38 = (tmp29 - (0xffffffff))
                u32 tmp29 = var2
                u32 tmp52 = var5
                u32 tmp82 = var6
                if((((arg15 | arg16 << 8 | arg17 << 16 | arg18 << 24)) != tmp38)) {
                    continue;
                }
                else {
                    break;
                }
            }
            Label L6:
            u16 tmp97 = var_x
            u8 tmp99 = var_r25
            u8 tmp103 = var_r24
            u8 tmp107 = var_r23
            u8 tmp105 = var_r22
            u8 tmp95 = var_r21
            u8 tmp34 = var_r20
            u8 tmp32 = var_r19
            u8 tmp28 = var_r18
        }
    }
    Label L11:
    Label L9:
    u24 tmp83 = (dmem[u24 , (arg20 + (0x1))])
    Return (tmp83, r1: (0x0), r18: tmp28, r19: tmp32, r20: tmp34, r21: tmp95, r22: tmp105,
        r23: tmp107, r24: tmp103, r25: tmp99, x: tmp97, sp: (arg20 + (0x3)), )
    goto L9
}
```

5.2.2 x86

Generated assembly of the fibonacci-function can be found in Listing 5.5. Compared to the AVR-assembly that we generate it is six times smaller. We compiled the program with the *-O1*. We also tried to compile the x86-version with *-O3*, but the result was not much different, as the function is quite trivial to compile.

```
0x0000053d push esi
0x0000053e push ebx
0x0000053f mov ebx, dword [esp+0xc]
0x00000543 mov eax, ebx
0x00000545 test ebx, ebx
0x00000547 je 0x56b
0x00000549 cmp ebx, 1
0x0000054c jbe 0x56e
0x0000054e mov ecx, 1
0x00000553 mov edx, 1
0x00000558 mov esi, 0
0x0000055d lea eax, [esi + edx]
0x00000560 add ecx, 1
0x00000563 mov esi, edx
0x00000565 mov edx, eax
0x00000567 cmp ebx, ecx
0x00000569 jne 0x55d
0x0000056b pop ebx
0x0000056c pop esi
0x0000056d ret
0x0000056e mov eax, 1
0x00000573 jmp 0x56b
```

Listing 5.5: x86 assembly of the iterative fibonacci implementation

holodec

Our decompiled version of the *x86* binary is slightly smaller than the AVR-version. The main reason is that fewer registers are used to store values. At the start of the function, we again see the callee-saved registers being stored on the stack. A few calculations are unusual, like *tmp36+tmp35*(0x1)+(0x0)*. This is an optimization result where calculations are replaced by *lea* instructions. The rest of the code is more or less identical to the AVR-version. A major difference is the lack of a body of the first branch. The compiler assigns the input value to *rax*. That means if the parameter equals 0 then the jump leads directly to the end-block, which returns from the function. At the moment we represent this as an empty block instead of swapping the condition and leave out the else-block.

```
Input (arg1 <- ebx, var_ecx, arg2 <- ecx, var_edx, arg3 <- edx, arg4 <- esi, arg5 <- esp, ){
    Label L2:
    (mem[u32 , (arg5 - (0x3))] = arg4)
    (mem[u32 , (arg5 - (0x7))] = arg1)
    u32 var_eax, tmp46 = (mem[u32 , ((arg5 - (0x8))+(0x0)*(0x1)+(0xc))])
    u32 tmp35 = var_edx
    u32 tmp34 = var_ecx
    u32 tmp32 = var_eax
    if(((tmp46 & tmp46) == (0x0))) {
    }
    else {
        Label L3:
        s32 tmp40 = (Cast[-> s32 ]((0x1)))
        if(((tmp46 < tmp40) || (tmp46 == tmp40))) {
            Label L7:
            u32 var_eax, tmp21 = (0x1)
            u32 tmp35 = var_edx
            u32 tmp34 = var_ecx
            u32 tmp32 = var_eax
        }
        else {
            Label L4:
            u32 var_edx, var_ecx, tmp36 = (0x1)
            u32 var_esi, tmp33 = (0x0)
            u32 tmp26 = var_ecx
            u32 tmp25 = var_edx
            u32 tmp24 = var_esi
            loop {
                Label L8:
                Label L5:
                u32 var_edx, var_eax, tmp29 = (tmp24+tmp25*(0x1)+(0x0))
                u32 var_ecx, tmp28 = (tmp26 + (0x1))
                u32 tmp26 = var_ecx
                u32 tmp25 = var_edx
                u32 tmp24 = var_esi
                u32 tmp35 = var_edx
                u32 tmp34 = var_ecx
                u32 tmp32 = var_eax
                if((tmp46 != (Cast[-> s32 ](tmp28)))) {
                    continue;
                }
                else {
                    break;
                }
            }
        }
    }
    Label L9:
    Label L6:
    u32 tmp1 = (arg5 + (0x8))
    u64 tmp16 = (mem[u64 , tmp1])
    Return (tmp16, eax: tmp32, ecx: tmp34, edx: tmp35, esp: tmp1, )
}
```

Listing 5.6: Iterative Fibonacci decompiled by holodec(x86)

Snowman

We decompile the same program with *snowman*. It is immediately apparent that the result is a lot smaller than our result. Looking at the variable names the influence of the internal IR in SSA-form becomes obvious. While in our interpretation we mostly discarded the information about registers as soon as the IR is built, snowman retains that information and seems to use the typical renaming approach[60] for handling registers/variables. In general, the result of snowman is smaller and easier to understand, in major parts due to a better pseudocode generator, which was not our focus.

```
uint32_t fibonacci(uint32_t a1) {
    uint32_t ebx2;
    uint32_t eax3;
    uint32_t ecx4;
    uint32_t edx5;
    uint32_t esi6;

    ebx2 = a1;
    eax3 = ebx2;
    if (ebx2) {
        if (ebx2 <= 1) {
            eax3 = 1;
        } else {
            ecx4 = 1;
            edx5 = 1;
            esi6 = 0;
            do {
                eax3 = esi6 + edx5;
                ++ecx4;
                esi6 = edx5;
                edx5 = eax3;
            } while (ebx2 != ecx4);
        }
    }
    return eax3;
}
```

Listing 5.7: Iterative Fibonacci decompiled by Snowman

RetDec

RetDec produces a slightly larger result than snowman. Interesting is, that RetDec consolidates the branches into a switch-case construct. This tells us that the RetDec decompiler does more during control-flow reconstruction in comparison to snowman or our own solution.

Unfavourable in the control flow is the goto statement, which ends the while loop. It should be replaced by a *break;*, but RetDec seems to not detect it. The reason for such a flaw can be that the exit is inside of an if-block, confusing the RetDec-decompiler. In our own implementation, we traverse the control-flow-structures upwards to find a loop. If the jump-target is the head of the loop we generate a continue-statement if it is the main-exit we generate a break. Interestingly enough in the comment in the previous line RetDec notifies that it breaks the loop. It seems not to know that the exit is, in fact, the main-exit as to generate a break.

Another curious case is the assignments of the variables *v1* and *g2*. These are callee-saved registers. It seems as if *RetDec* applies a calling convention (*cdecl* in this case), and just treats registers which are not considered by this convention as global variables. While this generates legal C-code it is an inelegant solution and more importantly: it confuses the reader, especially when seeing something like in Listing 5.8. Which we have found in the implementation of the printf function.

```
*(int32_t *)(g4 - 4) = v3;
*(int32_t *)(g4 - 8) = v2 + 194;
*(int32_t *)(g4 - 12) = 1;
function_3e0();
```

Listing 5.8: Call of printf generated by RetDec

```
int32_t fibonacci(int32_t a1) {
    int32_t v1 = g2;
    switch (a1) {
        default: {
            int32_t v2 = 1; // 0x5603
            int32_t v3 = 0; // 0x5631
            int32_t v4 = 1; // 0x5652
            // branch -> 0x55d
            int32_t result;
            while (true) {
                // 0x55d
                result = v3 + v4;
                int32_t v5 = v2 + 1; // bp+560
                if (v5 == a1) {
                    // break (via goto) -> 0x56b
                    goto lab_0x56b;
                }
                v2 = v5;
                v3 = v4;
                v4 = result;
                // continue -> 0x55d
            }
            lab_0x56b:
            // 0x56b
            // branch -> 0x56b
            // 0x56b
            g2 = v1;
            return result;
        }
        case 0: {
            // 0x56b
            g2 = v1;
            return 0;
        }
        case 1: {
            // 0x56e
            // branch -> 0x56b
            // 0x56b
            g2 = v1;
            return 1;
        }
    }
}
```

Listing 5.9: Iterative Fibonacci decompiled by RetDec

5.3 Fibonacci Recursive

The second program we compared is a recursive implementation to calculate the Fibonacci-numbers (Listing 5.10).

```
uint32_t fibonacci(uint32_t val) {
    if(val == 0)
        return 0;
    if(val == 1)
        return 1;
    return fibonacci(val - 1) + fibonacci(val - 2);
}
```

Listing 5.10: C-Source of recursive Fibonacci

5.3.1 AVR

We compiled the code with the AVR-version of GCC. We used *-O1* as optimization. The generated assembly can be found in Listing 5.11.

```
0x00000028 push r8
0x0000002a push r9
0x0000002c push r10
0x0000002e push r11
0x00000030 push r12
0x00000032 push r13
0x00000034 push r14
0x00000036 push r15
0x00000038 cp r22, r1
0x0000003a cpc r23, r1
0x0000003c cpc r24, r1
0x0000003e cpc r25, r1
0x00000040 breq 0x82
0x00000042 cpi r22, 0x01
0x00000044 cpc r23, r1
0x00000046 cpc r24, r1
0x00000048 cpc r25, r1
0x0000004a breq 0x8c
0x0000004c mov r12, r22
0x0000004e mov r13, r23
0x00000050 mov r14, r24
0x00000052 mov r15, r25
0x00000054 subi r22, 0x01
0x00000056 sbc r23, r1
0x00000058 sbc r24, r1
0x0000005a sbc r25, r1
0x0000005c rcall 0x28
0x0000005e mov r8, r22
0x00000060 mov r9, r23
0x00000062 mov r10, r24
0x00000064 mov r11, r25
0x00000066 mov r25, r15
0x00000068 mov r24, r14
0x0000006a mov r23, r13
0x0000006c mov r22, r12
0x0000006e subi r22, 0x02
0x00000070 sbc r23, r1
0x00000072 sbc r24, r1
0x00000074 sbc r25, r1
0x00000076 rcall 0x28
0x00000078 add r22, r8
0x0000007a adc r23, r9
0x0000007c adc r24, r10
0x0000007e adc r25, r11
0x00000080 rjmp 0x94
0x00000082 ldi r22, 0x00
0x00000084 ldi r23, 0x00
0x00000086 ldi r24, 0x00
0x00000088 ldi r25, 0x00
0x0000008a rjmp 0x94
0x0000008c ldi r22, 0x01
0x0000008e ldi r23, 0x00
0x00000090 ldi r24, 0x00
0x00000092 ldi r25, 0x00
0x00000094 pop r15
0x00000096 pop r14
0x00000098 pop r13
0x0000009a pop r12
0x0000009c pop r11
0x0000009e pop r10
0x000000a0 pop r9
0x000000a2 pop r8
0x000000a4 ret
```

Listing 5.11: AVR assembly of the recursive fibonacci implementation

holodec

The reconstructed result for the recursive implementation (Listing 5.12) is similar to the incremental version shown previously. The important parts are the recursive calls. We can see that all callee- and caller-saved registers are actually optimized away so that they do not appear as return values. The only expressions that could be improved are the multiple subtracts, generated before each recursive call. Without consolidating the four registers into a unified argument this is barely solvable. The optimizations exist in the decompiler to reduce the expressions, but the information that all four calculations

actually calculate different parts of the same value is missing. To solve that we would need proper type reconstruction which we, as already mentioned, have not implemented yet.

Listing 5.12: Recursive Fibonacci decompiled by holodec(AVR)

```
Input (arg1 <- r1, arg2 <- r8, arg3 <- r9, arg4 <- r10, arg5 <- r11, arg6 <- r12, arg7 <- r13,
       arg8 <- r14, arg9 <- r15, arg10 <- r22, arg11 <- r23, arg12 <- r24, arg13 <- r25,
       arg14 <- sp){
    Label L2:
    (dmem[u8 , arg14] = arg2)
    (dmem[u8 , (arg14 - (0x1))] = arg3)
    (dmem[u8 , (arg14 - (0x2))] = arg4)
    (dmem[u8 , (arg14 - (0x3))] = arg5)
    (dmem[u8 , (arg14 - (0x4))] = arg6)
    (dmem[u8 , (arg14 - (0x5))] = arg7)
    (dmem[u8 , (arg14 - (0x6))] = arg8)
    (dmem[u8 , (arg14 - (0x7))] = arg9)
    if((((arg10 | arg11 << 8 | arg12 << 16 | arg13 << 24)) ==
            ((arg1 | arg1 << 8 | arg1 << 16 | arg1 << 24)))) {
        Label L5:
        u8 var_r25, var_r24, var_r23, var_r22, tmp97 = (0x0)
        u0 tmp37 = var_r0
        u8 tmp63 = var_r25
        u8 tmp67 = var_r24
        u8 tmp69 = var_r23
        u8 tmp73 = var_r22
        u0 tmp34 = var_r1
    }
    else {
        Label L3:
        if((((arg10 | arg11 << 8 | arg12 << 16 | arg13 << 24)) ==
                (((0x1) | arg1 << 8 | arg1 << 16 | arg1 << 24)))) {
            Label L6:
            u8 var_r22, tmp94 = (0x1)
            u8 var_r25, var_r24, var_r23, tmp93 = (0x0)
            u0 tmp37 = var_r0
            u8 tmp63 = var_r25
            u8 tmp67 = var_r24
            u8 tmp69 = var_r23
            u8 tmp73 = var_r22
            u0 tmp34 = var_r1
        }
        else {
            Label L4:
            u8 tmp108 = (arg10 - (0x1))
            u16 tmp79 = ((arg10 | arg11 << 8))
            u8 tmp75 = ((u8 (tmp79 - (((0x1) | arg1 << 8))) >> 8))
            u24 tmp74 = ((arg10 | arg11 << 8 | arg12 << 16))
            u8 tmp70 = ((u8 (tmp74 - (((0x1) | arg1 << 8 | arg1 << 16))) >> 16))
            u32 tmp68 = ((arg10 | arg11 << 8 | arg12 << 16 | arg13 << 24))
            u8 tmp62 = ((u8 (tmp68 - (((0x1) | arg1 << 8 | arg1 << 16 | arg1 << 24))) >> 24))
            u16 tmp60 = (arg14 - (0xb))
            u16 tmp58 = (arg14 - (0xa))
            (dmem[u24 , tmp58] = (0x5e))
            u8 tmp87 <- r22, u8 tmp100 <- r23, u8 tmp101 <- r24, u8 tmp102 <- r25, Call (0x28)
                (r1 <- arg1, r8 <- arg2, r9 <- arg3, r10 <- arg4, r11 <- arg5, r12 <- arg10,
                r13 <- arg11, r14 <- arg12, r15 <- arg13, r22 <- tmp108, r23 <- tmp75,
                r24 <- tmp70, r25 <- tmp62, sp <- tmp60, )
            u8 tmp43 = (arg10 - (0x2))
            u8 tmp53 = ((u8 (tmp79 - (((0x2) | arg1 << 8))) >> 8))
            u8 tmp49 = ((u8 (tmp74 - (((0x2) | arg1 << 8 | arg1 << 16))) >> 16))
            u8 tmp41 = ((u8 (tmp68 - (((0x2) | arg1 << 8 | arg1 << 16 | arg1 << 24))) >> 24))
            (dmem[u24 , tmp58] = (0x78))
            u8 tmp3 <- r22, u8 tmp22 <- r25, u8 tmp30 <- r24, u8 tmp103 <- r23, Call (0x28)
                (r1 <- arg1, r8 <- tmp87, r9 <- tmp100, r10 <- tmp101, r11 <- tmp102,
                r12 <- arg10, r13 <- arg11, r14 <- arg12, r15 <- arg13, r22 <- tmp43,
                r23 <- tmp53, r24 <- tmp49, r25 <- tmp41, sp <- tmp60, )
            u32 tmp39 = (((tmp3 | tmp103 << 8 | tmp30 << 16 | tmp22 << 24)) +
                ((tmp87 | tmp100 << 8 | tmp101 << 16 | tmp102 << 24)))
            u8 var_r25, tmp38 = ((u8 tmp39 >> 24))
            u8 var_r24, tmp81 = ((u8 tmp39 >> 16))
```

```
            u8 var_r22, tmp85 = (u8 tmp39)
            u8 var_r23, tmp84 = ((u8 tmp39 >> 8))
            u0 tmp37 = var_r0
            u8 tmp63 = var_r25
            u8 tmp67 = var_r24
            u8 tmp69 = var_r23
            u8 tmp73 = var_r22
            u0 tmp34 = var_r1
        }
    }
    Label L8:
    Label L7:
    u24 tmp88 = (dmem[u24 , (arg14 + (0x1))])
    Return (tmp88, r22: tmp73, r23: tmp69, r24: tmp67, r25: tmp63, sp: (arg14 + (0x3)))
    goto L7
}
```

5.3.2 x86

Compiled with GCC with the *-O1* option wields a far smaller result for x86 than for AVR, as all values can be put into single registers and the operations do not need to be performed four times for 32-bit values. 24 instructions were generated, not much compared to the 63 instructions in the AVR example.

```
0x0000053d push esi
0x0000053e push ebx
0x0000053f sub esp, 4
0x00000542 mov ebx, dword [esp+0x10]
0x00000546 test ebx, ebx
0x00000548 je 0x54f
0x0000054a cmp ebx, 1
0x0000054d jne 0x557
0x0000054f mov eax, ebx
0x00000551 add esp, 4
0x00000554 pop ebx
0x00000555 pop esi
0x00000556 ret
0x00000557 sub esp, 0xc
0x0000055a lea eax, [ebx - 1]
0x0000055d push eax
0x0000055e call 0x0000053d
0x00000563 mov esi, eax
0x00000565 sub ebx, 2
0x00000568 mov dword [esp], ebx
0x0000056b call 0x0000053d
0x00000570 add esp, 0x10
0x00000573 lea ebx, [esi + eax]
0x00000576 jmp 0x54f
```

Listing 5.13: x86 assembly of the recursive fibonacci implementation

holodec

Our generated code is quite similar to the iterative solution. Interestingly enough the generated function signature is smaller for the recursive version. The reason for that is that fewer registers are used in the recursive solution, which leads to the function needing to save less register on the stack. The problem with phi-expressions creating lots of unnecessary noise and the problem with the *lea*-instructions still remains.

```
Input (arg1 <- ebx, arg2 <- esi, arg3 <- esp, ){
    Label L2:
    (mem[u32 , (arg3 - (0x3))] = arg2)
    (mem[u32 , (arg3 - (0x7))] = arg1)
    u32 var_esp, tmp53 = (arg3 - (0xc))
    u32 var_ebx, tmp4 = (mem[u32 , (var_esp+(0x0)*(0x1)+(0x10))])
    u0 tmp39 = var_eax
    u32 tmp38 = var_esp
    u32 tmp36 = var_ebx
    if(((tmp4 & tmp4) == (0x0))) {
    }
    else {
        Label L3:
        u0 tmp39 = var_eax
        u32 tmp38 = var_esp
        u32 tmp36 = var_ebx
        if((tmp4 != (Cast[-> s32 ]((0x1))))) {
            Label L5:
            (mem[u32 , (arg3 - (0x1b))] = (tmp4+(0x0)*(0x1)+(0xffffffffffffffff)))
            (mem[u32 , (arg3 - (0x1c))] = (0x563))
            u32 tmp10 = (arg3 - (0x20))
            u32 tmp9 <- esp, u32 tmp21 <- eax, u32 tmp22 <- ebx, \
                Call (0x53d)(ebx <- tmp4, esi <- arg2, esp <- tmp10, )
            u32 tmp61 = (tmp22 - (0x2))
            (mem[u32 , tmp9] = tmp61)
            (mem[u32 , tmp9] = (0x570))
            u32 tmp1 = (tmp9 - (0x4))
            u32 tmp67 <- esp, u32 tmp68 <- esi, u32 tmp73 <- eax, \
                Call (0x53d)(ebx <- tmp61, esi <- tmp21, esp <- tmp1, )
            u32 var_esp, tmp51 = (tmp67 + (0x10))
            u32 var_ebx, tmp30 = (tmp68+tmp73*(0x1)+(0x0))
            u0 tmp39 = var_eax
            u32 tmp38 = var_esp
            u32 tmp36 = var_ebx
        }
        else {
        }
    }
    Label L6:
    Label L4:
    u32 tmp69 = (mem[u32 , (tmp38 + (0x5))])
    u32 tmp72 = (mem[u32 , (tmp38 + (0x9))])
    u32 tmp43 = (tmp38 + (0x14))
    u64 tmp37 = (mem[u64 , tmp43])
    Return (tmp37, eax: tmp36, ebx: tmp69, esi: tmp72, esp: tmp43, )
}
```

Listing 5.14: Recursive Fibonacci decompiled by holodec

Snowman

If we look at the result of *snowman* we can see it has a few problems detecting the correct function signature. The decompiler assumes four arguments for the function, which is obviously not true, especially regarding how the arguments are used. Three of them are not used at all.

The control flow is quite interesting as *snowman* groups together both branches into one, which makes the whole function much more readable. Also, there is less noise compared to our implementation, which is good.

```
int32_t fibonacci(int32_t a1, int32_t a2, int32_t a3, int32_t a4) {
    int32_t ebx5;
    int32_t v6;
    int32_t v7;
    int32_t v8;
    int32_t eax9;
    int32_t v10;
    int32_t v11;
    int32_t v12;
    int32_t eax13;

    ebx5 = a1;
    if (ebx5 && ebx5 != 1) {
        eax9 = fibonacci(ebx5 - 1, v6, v7, v8);
        eax13 = fibonacci(ebx5 - 2, v10, v11, v12);
        ebx5 = eax9 + eax13;
    }
    return ebx5;
}
```

Listing 5.15: Recursive Fibonacci decompiled by Snowman

RetDec

Compared to *snowman*, RetDec correctly detects that the Fibonacci-function has only one argument, but instead other problems manifest. The first is the length of the generated code. There is lots of unnecessary noise, which is in effect useless. Similar to the iterative example callee-saved registers are displayed, which have absolutely no bearing on the result of the function. The only upside to this is that they are not present in the function signature, but the constant writes and reads from *g2* and *g4* which correspond to *ebx* and *esi* are distracting. The branches are not consolidated into one like *snowman* did. A good point is that instead of generating a *goto* statement the return at the end is copied into the first branch.

```
int32_t fibonacci(int32_t a1) {
        int32_t v1 = g4;
        int32_t v2 = g2;
        g2 = a1;
        if (a1 == 0) {
                // 0x54f
                g2 = v2;
                g4 = v1;
                return 0;
        }
        int32_t result = a1;
        if (a1 != 1) {
                // 0x557
                g4 = fibonacci(a1 - 1);
                int32_t v3 = g2 - 2; // bp+565
                g2 = v3;
                result = g4 + fibonacci(v3);
                // branch -> 0x54f
        }
        // 0x54f
        g2 = v2;
        g4 = v1;
        return result;
}
```

Listing 5.16: Recursive Fibonacci decompiled by RetDec

CHAPTER 6

Conclusion

Before we address the possibilities for future additions to our system, we will recap what went well, what was not so good and what went completely wrong.

6.1 The Good

The development of certain features and techniques was met with a varying degree of success. The following parts were implemented successfully and also have proven to be good techniques to solve the relevant problems.

6.1.1 Recursive Descending Disassembler driven by IR-Generation

Our approach of IR-generation driven disassembling turned out to be an efficient one. Our generated language, which would parse instructions into our IR worked surprisingly well. The overarching recursive descending parser which at the same time generated our IR worked well, but the problem with indirect calls are for the moment not addressed. Our first approach to decide depending on information from the disassembler was working but using the IR-generation itself for controlling disassembling meant we did not need to store/generate duplicate information.

6.1.2 IR in SSA-form

Working with an IR in SSA-form was great. It is extremely easy to replace expressions, make changes to the IR and general manipulations. Of course, this only applies once the IR is actually implemented correctly, which took time, but once we reached that point(or close to that point) most optimizations became quite easy. Especially dead code elimination and peephole optimization were easy to implement and brought great improvements.

The approach to completely disassociate the IR from the underlying machine seemed to mostly be a good one. We lose information about what values are stored in what register at a location. This information is good to preserve but outside of certain points(e.g. function-calls) it is not really needed. So we only kept the information about registers at these points.

6.1.3 Expand and Compress

Our basic concept of first generating lots of unoptimized code and then compressing it back down as a concept was originally borrowed from compilers. Most notable from GCC which used this idea for a long time. For complex instructions, our implementation generates a large number of expressions because we try to model everything as accurately as possible. Our implemented peephole optimizer together with DCE then compresses everything back to a manageable level. We are constantly in the process of searching for better peephole rules so the final code gets clearer and shorter. This worked especially well and is surprisingly fast.

6.1.4 Structure Reconstruction

We used an implementation of the goto-free control-flow reconstruction algorithm[61]. Our algorithm needed some tricks to work correctly, but the general idea guided our implementation. The result was an algorithm, which would generate nearly no *goto*s. There are still a few issues with our solution but it already shows great results.

6.2 The Bad

Trying to limit yourself to general optimizations and not going into details of how certain compilers actually generate code was generally tricky to do because it limited the number of optimizations possible. We will again mention the problematic parts of our implementation.

6.2.1 Calling conventions

There are a few possible solutions for how to use standard calling conventions to help improve the result. One possible way would be to assume a calling convention and if a register is used that is not part of the calling convention, then treat it as a global variable similar to what *RetDec* does. A second possible solution would be to strictly assume a calling convention is used and discard everything away that does not adhere to that convention. A third way of solving the problem is to try to fit the calling convention and if a value is used that does not fit it, then mark or hide it, but do not throw it away.

The first works, but it generates a lot of noise which we would like to try to avoid. The second solution is unacceptable because possible correct dataflow gets thrown away. The third is in our opinion the best possible choice, but it has to be purely a cosmetic

one. It should only be applied in the final code generations stage. Hiding such arguments based upon heuristics and pattern matching and later refinement by user input would be the best choice as discussed later in Section 6.3.1.

In our solution, we could not remove lots of arguments because we tracked all registers in an architecture. This includes flags and other types of not used registers. This solution works but creates a lot of visual noise in the final code. A better way would be to treat certain registers as global variables or hide certain registers if they are not used. Especially for flag registers, this is important because typically they are written to often but their use is mostly implicitly part of other operations. Because of that in most cases, they can be hidden and only shown on user intervention.

6.2.2 Code Generator

Our current pseudo-code generation is just a glorified IR-visualizer. We do a few more things to beautify the result like control flow reconstruction, but the rest is still in its beginnings. The generated code is not as readable as it could be. The pseudo code is generated by simply parsing the IR. There are better solutions like completely translating the IR into an abstract syntax tree(AST). This would enable more optimizations which are tricky to do on the fly while parsing an IR.

One such difficult task is to handle phi-expressions. At the moment we simply look at any following basic blocks and generate assignments for every phi-expression that we can find. Maybe we can handle this better once translated into a non-SSA-form IR. At this point, it is hard to tell.

6.2.3 Architecture/Compiler Independence

Our approach in trying to be as architecture independent as possible was quite successful and normal calculations could be optimized, translated and displayed quite well. Boundaries that are dependant on the architecture/compiler, on the other hand, were difficult to process. Function calls were troublesome because we could rarely distinguish between actual arguments and metadata that was passed e.g. the pointer to the stack. These problems were difficult to address.

6.3 The Ugly

Some parts of our solution were especially unsuccessful in the way that we implemented them. These are wholly inadequate or insufficient in solving the problem. We will go over these unsuccessful implementations again and discuss their failings.

6.3.1 Optimizing Away Call Parameters

At call/return-expressions control flow gets passed from one function to another. In this case, we have to also pass all relevant registers to the expression. In order to do this, we

try to analyse all functions and from the information what registers are used we then can remove arguments on call sites. This works to a limited extent because we get a lot of false-negative results because of callee-saved registers. Because of our conservative approach, there is not really a way of solving that issue. In *SecondWrite*[40] the authors claim to have good success with their solution which is very similar to what we do. The major difference is their handling of return values. While we assume all registers written to are return values and then try to reduce the amount they have flipped the assumption by assuming no return argument and then adding them according to some rules.

Other decompilers seem to solve this by guessing function signatures or using heuristics and pattern matching. This is possible but might create other inaccuracies. During development, this single issue has been a major problem throughout, because for most other problems there exist reasonable good algorithms to solve them(e.g. alias analysis, control flow reconstruction, ...). Detecting function-arguments and return-values, on the other hand, is mostly a guessing game and there are no good solutions.

With our current design, this is complicated to implement because we are not able to rebuild the IR with additional information. So if the heuristics are wrong there is no way of correcting the mistakes. In Chapter 7 we discussed a different architecture for decompilation, which is better at handling such cases.

Another major problem with our current design that makes heuristics difficult is the separation of the IR from the instructions. We only refer back to the instructions by an address, which is a part of each IR-expression. Optimizations might remove or combine expressions which can make assignments to registers seem more distant and less relevant to calls. A heuristic might decide wrongly because of these discrepancies between optimized IR and actual instructions.

6.3.2 Inflexible Function Analysing

While once the IR of the function is generated we can change and remove a lot from the IR. The big problem appears when trying to add code sections. For example, let us assume a switch-case. A compiler might decide to implement it using a lookup table and indirectly jumping to the different branches[7]. During analysing we might detect the lookup table. Because of our design, we may not add additional basic blocks to the function after a specific point, because optimizations might have already altered the IR in a way so that we can not correctly add those additional basic blocks. This is problematic. The straightforward solution would be to rebuild the IR of the whole function and start anew. This is extremely excessive and not an elegant solution. In Chapter 7 we will discuss a possible solution which would require a major rebuild of large parts of the decompiler but would make it more flexible to changes.

CHAPTER 7

Future Work

We have already mentioned some features which would improve the generated pseudo-code. Here is a more comprehensive list of possible additions to our decompilation solution.

7.1 Change Memory Read/Writes To Proper Variable Assignments/Reads

Currently, we represent memory loads and stores as loads/stores with an address. Access of memory at specific locations. The next step would be to categorize them all into variables, either in the global scope, the stack or from a passed parameter, and display said variables instead of the loads and stores. This would improve readability a lot as one of the major problems at the moment with the resulting code is exactly those memory manipulating expressions. At the moment each gets turned into an expression which is much more complicated than necessary representing the operation. Changing this representation to variables would help in discerning what is going on in the program.

7.2 Integrate radare2

Radare2 is an extensive binary analysis tool that supports a wide range of binary formats and instruction set architectures. We already mentioned that radare2 has a way of turning instructions into an intermediate representation that describes the instructions. We already started working on writing a front-end that would use radare2 to analyse the binaries for us and use ESIL to parse instructions into our own SSA-form. This would allow us to use the already extensive capabilities of radare2 and build on them. We had major issues in calling radare2 code. The lack of documentation and the lack of comments in the code made it hard to actually figure out how to properly use the radare2-API properly.

7.3 Type reconstruction/Typed Values

Most actual programming languages have types that are used to group memory and give this memory a special meaning. A 4-byte load from a memory region with the address of a function parameter plus 8 without any context has barely any meaning at all. It is hard to understand.

```
struct integer_entry{
    struct integer_entry* next;
    int value:
};
int get_value(struct integer_entry* entry){
    return entry->value;
}
```

Listing 7.1: Access of member of a struct with a pointer

Could with perfectly reconstructed types be converted into:

```
struct type{
    type* ptr1;
    u32 val1:
};
u32 function_xyz(type* param1){
    return param1->val1;
}
```

Listing 7.2: Reconstruction of Listing 7.1

This tells us a lot about what is happening in the program compared to the type-less and variable-less version:

```
function_xyz(u64 entry){
    return mem[u32, entry + 8];
}
```

Listing 7.3: Bad reconstruction of Listing 7.1

Without types, even with anonymized types, we can understand what is happening much faster because the code is much clearer. The examples that we have here are obviously perfect versions of decompiled code. Compared to our results which are not yet at this point.

Also part of type reconstruction is function signature detection. This is difficult and probably not possible in an architecture/compiler agnostic manner. There will probably be some kind of specific analyse pass for each architecture.

7.4 Layered Decompilation Stages

During development, certain issues appeared where the best solution would be to save newly gained information and redo certain passes. We solved the issue by looping over optimization-passes until nothing changed. This worked, but it was not comprehensive enough. From the experience, we had while developing this decompiler we came up with a different design that might help alleviate these issues.

The new approach would consist of different stages that would build upon each other and only add information. One such layer may never change a layer that it depends upon in a way to destructively change the semantics. The only available change is to add information in order to refine results. An example of such a case would be to add invariants to certain locations.

The first layer would be the disassembler. Currently, we disassemble whole functions in one go. We would change this by disassembling basic blocks as we find them. The disassembling can be both recursive descending and a linear sweep as all blocks go into a global pool of basic blocks. Each instruction in a block is also raised into the IR. But different to our current implementation each block stays independent. Invariants and read/write information can be added to a block in order to help any subsequent passes.

The second layer would be the function composition. Basic blocks need to be grouped in order to form functions. This analysis can be done with the help of the generated IR of each basic block. Jump-expressions are analysed if they are calls, returns, branches or relative jumps(we would remove dedicated call- and return-expressions and use a generic jump-expression). Calls are typically easy to detect. They are direct or indirect jumps and the address of the next instruction is previously written in the memory. Branches should also be quite easy to detect, even if they use jump-tables. It is a bit tricky to distinguish between virtual-calls and returns, but if you consider that returns nearly always read the target-address from the stack at the location it was passed into the function then this should also not be too difficult to detect.

The third layer is function-wide IR-generation. We would generate an IR in SSA-form that spans a whole function similar to what we do now. Different optimizations could be run on this layer, like DCE and peephole optimization.

For the fourth layer, we would translate the IR into an AST-like structure. This structure would be used to display the final result. It would be a proper data structure and not just a visualization of the internal IR. The goal for this would be to simplify the information as much as possible into expressions so that it is understandable and easy to visualize.

This layered approach has a few advantages. For example, let us assume we try to decompile a switch-case statement that is implemented via a jump table. Currently, if the first IR-generation pass can not correctly find the jump-table and it is found, later on, the only way of dealing with this information is to throw all information about the

function away and redo everything(disassembling and IR-generation). With this new approach, an analysis pass might add the information about the jump-table and possible invariants to variables used in the calculation to the basic block. Once all analysis-passes are done we can simply throw layer two and three for this function away, create new basic blocks if needed and recompose the function. This might also seem quite wasteful, but we do not need the extra step of disassembling and generating the IR.

A big problem which we might be able to handle better would be the detection of function arguments. At the moment once we remove or change an argument we can not reverse this decision. A layered approach would allow us to gather information about arguments either from later analysis passes or from the user and use these to rerun certain passes. We would also be able to optimize more aggressively because the possibility exists to change these decisions later on without rebuilding everything.

Another way of solving the problem would be to always keep the full state information of the machine. With this, we do not need to rebuild anything. The problem with this approach is that optimizations are nearly impossible. Consider an add-instructions that writes the carry-flag. Can we optimize away the write to the carry flag if it is always overwritten in the next instruction? If we want to keep all state information then we can not do this, because we would lose that information. A newly resolved jump might split the basic block in two in between those two instructions. If we threw away the carry-flag we lost information.

List of Figures

List of Tables

Listings

Bibliography

[1] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37, April 2016.

[2] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *Int. Conf. on Broadband, Wireless Computing, Communication and Applications*, pages 297–300, 2010.

[3] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.

[4] HynekPetrak. Javascript malware collection, 2017. `https://github.com/HynekPetrak/javascript-malware-collection`.

[5] Christopher Krügel, William K. Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX Security Symposium*, 2004.

[6] C. Jämthagen, P. Lantz, and M. Hell. A new instruction overlapping technique for anti-disassembly and obfuscation of x86 binaries. In *2013 Workshop on Anti-malware Testing Research*, pages 1–9, Oct 2013.

[7] C. Cifuentes and M. Van Emmerik. Recovery of jump table case statements from binary code. In *Proceedings Seventh International Workshop on Program Comprehension*, pages 192–199, May 1999.

[8] B. Schwarz, S. Debray, and G. Andrews. Disassembly of executable code revisited. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 45–54, Nov 2002.

[9] Melvin E. Conway. Proposal for an uncol. *Commun. ACM*, 1(10):5–8, October 1958.

[10] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE Computer Society, 2004.

[11] Thomas Dullien and Sebastian Porst. Reil: A platform-independent intermediate representation of disassembled code for static code analysis. 01 2009.

[12] Radare2 book, 2019. `https://radare.gitbooks.io/radare2book/disassembling/esil.html`.

[13] Bil, 2018. `https://github.com/BinaryAnalysisPlatform/bap`.

[14] David Brumley, Ivan Jager, Thanassis Avgerinos, and Edward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 463–469, Berlin, Heidelberg, 2011. Springer-Verlag.

[15] M.J. Van Emmerik. *Static Single Assignment for Decompilation.* University of Queensland, 2007.

[16] Allen Leung and Lal George. Static single assignment form for machine code. *SIGPLAN Not.*, 34(5):204–214, May 1999.

[17] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM.

[18] G. J. Chaitin. Register allocation & spilling via graph coloring. *SIGPLAN Not.*, 17(6):98–101, June 1982.

[19] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using ssa-graphs. *SIGPLAN Not.*, 43(7):31–40, June 2008.

[20] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6:505–526, 1984.

[21] Jason Merrill. Generic and gimple: A new tree representation for entire functions. In *In Proceedings of the 2003 GCC Summi*, pages 171–180, 2003.

[22] Diego Novillo. Gcc- an architectural overview, current status and future. In *Directions," Proceedings of the Linux Symposium*, pages 193–208, 2006.

[23] Chris Lattner and Vikram Adve. Architecture for a Next-Generation GCC. In *Proc. First Annual GCC Developers' Summit*, Ottawa, Canada, May 2003.

[24] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. *SIGSOFT Softw. Eng. Notes*, 21(6):81–92, October 1996.

[25] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of c code in a second. *SIGPLAN Not.*, 36(5):254–263, May 2001.

[26] Amer Diwan, Kathryn S. McKinley, and J. Eliot B. Moss. Type-based alias analysis. *SIGPLAN Not.*, 33(5):106–117, May 1998.

[27] Diego Novillo. Memory ssa- a unified approach for sparsely representing memory operations.

[28] Fred Chow, Sun Chan, Shin Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In Tibor Gyimóthy, editor, *Compiler Construction*, pages 253–267, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[29] Jack W. Davidson and Christopher Fraser. Register allocation and exhaustive peephole optimization. *Softw., Pract. Exper.*, 14:857–865, 09 1984.

[30] Chirag Bhatt and Harshad Bhadka. Peephole optimization technique for analysis and review of compiler design and construction. *IOSR Journal of Computer Engineering (IOSR-JCE) 2278-0661 (impact Factor - 1.69)*, 9:80–86, 03 2013.

[31] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[32] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Comput. Surv.*, 48(4):65:1–65:35, May 2016.

[33] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Proceedings of the 8th European Symposium on Programming Languages and Systems*, ESOP '99, pages 208–223, Berlin, Heidelberg, 1999. Springer-Verlag.

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

[35] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[36] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 119–132, New York, NY, USA, 1999. ACM.

[37] Gogul Balakrishnan and Thomas Reps. Divine: Discovering variables in executables. In *Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'07, pages 1–28, Berlin, Heidelberg, 2007. Springer-Verlag.

[38] Thomas Reps, Gogul Balakrishnan, and Junghee Lim. Intermediate-representation recovery from low-level code. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 100–111, New York, NY, USA, 2006. ACM.

[39] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, August 2010.

[40] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. *SIGPLAN Not.*, 48(6):51–60, June 2013.

[41] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, CERIAS '10, pages 5:1–5:1, West Lafayette, IN, 2010. CERIAS - Purdue University.

[42] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 22nd USENIX Security Symposium*, 01 2011.

[43] A. Chernov, K. Troshina, and Y. Derevenets. Reconstruction of composite types for decompilation. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation(SCAM)*, volume 00, pages 179–188, 09 2010.

[44] E. N. Dolgova and A. V. Chernov. Automatic reconstruction of data types in the decompilation problem. *Programming and Computer Software*, 35(2):105–119, Mar 2009.

[45] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*, pages 353–368. USENIX Association, 2013.

[46] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 353–368, Washington, D.C., 2013. USENIX.

[47] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Network and Distributed System Security (NDSS), ISOC*, 2015.

[48] Hex-rays decompiler, 2019. https://www.hex-rays.com/products/decompiler/index.shtml.

[49] Decompiler internals: microcode, 2019. https://hex-rays.com/products/ida/support/ppt/recon2018.ppt.

[50] Ida, 2019. https://www.hex-rays.com/products/ida/.

[51] Snowman, 2019. http://derevenets.com/.

[52] Retargetable decompiler, 2019. https://retdec.com/.

[53] Cristina Cifuentes and K. John Gough. A methodology for decompilation. In *in Proceedings for the XIX Conferencia Latinoamericana de Informatica, Buenos Aires*, pages 257–266, 1993.

[54] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Softw. Pract. Exper.*, 25(7):811–829, July 1995.

[55] Boomerang decompiler, 2019. http://boomerang.sourceforge.net/.

[56] M. V. Emmerik and T. Waddington. Using a decompiler for real-world source recovery. In *11th Working Conference on Reverse Engineering*, pages 27–36, Nov 2004.

[57] Rec decompiler, 2019. http://www.backerstreet.com/rec/rec.htm.

[58] Ghidra, 2019. https://github.com/NationalSecurityAgency/ghidra.

[59] gereeter. hsdecomp, 2016. https://github.com/gereeter/hsdecomp.

[60] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In Evelyn Duesterwald, editor, *Compiler Construction*, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[61] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations. In *NDSS*. The Internet Society, 2015.