BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DECOMPILATION OF SPECIALIZED AND ADVANCED INSTRUCTION SETS

ZPĚTNÝ PŘEKLAD SPECIALIZOVANÝCH A POKROČILÝCH INSTRUKČNÍCH SAD

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **JURAJ HOLUB**
AUTOR PRÁCE

SUPERVISOR — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2020

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2019/2020

Bachelor's Thesis Specification

Student: **Holub Juraj**
Programme: Information Technology
Title: **Decompilation of Specialized and Advanced Instruction Sets**
Category: Compiler Construction

Assignment:

1. Study the problems of reverse engineering. Focus on the decompilation of a binary code to its high-level representation.
2. Familiarize yourself with RetDec decompiler and research technologies used in this decompiler (LLVM, Capstone, Keystone, etc.).
3. Familiarize yourself with the processor architectures supported by RetDec decompiler. Focus on unsupported or partially supported processor extensions and instruction sets.
4. Analyze current flaws and potential obstacles in adding support for these instruction sets. Select a set of instruction extensions with the best ratio of positive impact for the decompilation quality to difficulty of their implementation.
5. Design new components and propose modifications to RetDec decompiler necessary for the implementation of the instruction sets chosen in point 4.
6. After a discussion with the supervisor and the consultant, implement features from point 5.
7. Carefully test the implemented solution. Include real-world programs making use of the selected instruction extensions in your test suite.
8. Summarize your thesis and discuss future developments.

Recommended literature:

- E. Eilam: Reversing: Secrets of Reverse Engineering, Wiley 2005, ISBN 978-076457481.
- Project documentation for RetDec, LLVM, Capstone, Keystone etc.
- Documentations for architectures of x86, x86_64, ARM, ARM64, MIPS, PowerPC processors.
- According to the supervisor/consultant recommendation.

Requirements for the first semester:

- First five items and partially item 6.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Křivka Zbyněk, Ing., Ph.D.**
Consultant: Matula Peter, Ing., Avast
Head of Department: Kolář Dušan, doc. Dr. Ing.
Beginning of work: November 1, 2019
Submission deadline: May 28, 2020
Approval date: October 21, 2019

Abstract

Nowadays, the process of analyzing malicious software is an important part of information technologies. One of the crucial techniques is decompilation of malicious binary programs. The decompilation is a complex process, and there are multiple projects with such a goal. The project RetDec aims to develop retargetable and flexible decompiler. The goal of this research is to improve the decompilation of advanced instruction sets for architecture x86. The new optimization for FPU register stack manipulation is designed, and the support of FPU and SSE instruction set translation is extended. The new extensions are implemented and tested in the manner of decompilation efficiency and quality.

Abstrakt

V dnešnej dobe je proces analýzy nebezpečného softvéru dôležitou súčasťou informačných technológií. Jedna z kľúčových techník je spätný preklad škodlivých binárnych programov. Spätný preklad je komplexný proces, ktorý rieši niekoľko projektov. Projekt RetDec sa zameriava na flexibilný návrh a riešenie spätného prekladača s možnosťou znovupoužiteľnosti. Cieľom tejto práce je zlepšenie spätného prekladu pokročilých inštrukčných sád pre architektúru x86. Bola navrhnutá nová optimalizácia pre FPU registrový zásobník. Bola rozšírená podpora prekladu inštrukčných sád jednotiek FPU a SSE. Nové rozšírenia boli implementované a otestované z hľadiska efektivity a kvality spätného prekladu.

Keywords

compiler, decompiler, reverse engineering, x86, FPU, SSE, RetDec

Klíčová slova

prekladače, spätné prekladače, reverzné inžinierstvo, x86, FPU, SSE, RetDec

Reference

HOLUB, Juraj. *Decompilation of Specialized and Advanced Instruction Sets.* Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Zbyněk Křivka, Ph.D.

Rozšířený abstrakt

Táto práca sa zaoberá využitím reverzného inžinierstva v oblasti softvérových technológií. Reverzné inžinierstvo je všeobecne metóda získavania informácií alebo plánov o akýchkoľvek objektoch vytvorených človekom. V oblasti informačných technológií je význam tejto disciplíny najmä v rámci kybernetickej bezpečnosti. Táto technika je využívaná tvorcami škodlivého softvéru (tzv. malvér). Malvér využíva reverzné inžinierstvo na získavanie citlivých informácií o operačnom systéme s potenciálnym cieľom získať kontrolu nad zariadenim. Ďalšia rozšírená oblasť je softvérové pirátstvo, kedy sa útočník snaží prelomiť ochranu komerčného digitálneho obsahu ako sú knihy, filmy, hudba, hry alebo rôzne platené programy. Na druhej strane môže pomôcť práve pri analýze malvéru za účelom zvýšenia bezpečnosti voči danému softvéru.

Jedna z kľúčových techník pre analýzu malvéru je analýza pomocou programu všeobecne nazývaneho spätný prekladač. Spätný prekladač je program, ktorý analyzuje spustiteľné binárne súbory a zrekonštruuje vysoko úrovňový výstup, napríklad v podobe grafu alebo kódu v programovacom jazyku. V dnešnej dobe existuje niekoľko projektov spätných prekladačov. Projekt RetDec sa zameriava na vytvorenie open-source nástroja, ktorý je rozdelený na viacero knižníc. Takýto návrh má za cieľ umožniť znovupoužiteľnosť jednotlivých nástrojov spätného prekladača.

Cieľom tejto práce je rozšíriť podporu spätného prekladu v projekte RetDec o špecializované inštrukčné sady FPU a SSE (procesorová architektúru x86). Bol vytvorený návrh nových rozšírení na základe zhodnotenia aktuálnej podpory inštrukčných sád FPU a SSE. Inštrukčná sada FPU bola v rámci práce rozšírená na 100 % inštrukcií. RetDec už v súčasnosti novú implementáciu podporuje. Pre sadu SSE bol vytvorený a čiastočne implementovaný návrh, ktorý rozlišuje inštrukcie na skalárne a vektorové.

Druhé rozšírenie sa zameriava na optimalizáciu spätného prekladu FPU registrov, ktoré tvoria zásobníkovú štruktúru. V rámci práce bola navrhnutá nová optimalizácia, ktorá transformuje prácu s FPU zásobníkom na preurčenú sústavu lineárnych rovníc. V ďalšej časti práce sa zhodnotili rôzne aproximačné metódy na riešenie získaného systému. Bol vykonaný výkonnostný experiment, ktorý bol meraný na spätnom preklade stoviek skutočných binárnych spustitelných súborov. Experiment porovnal efektivitu skutočnej implemntácie pre jednotlivé navhované metódy a zvolil najoptimálnejšiu, ktorá bola následne integrovaná do novej optimalizácie.

Záver práce popisuje testovanie implementácie nových rozšírení v spätnom prekladači RetDec. V rámci testovania boli použité tri testovacie nástroje. Prvé dva nástroje testovali nové rozšírenia pomocou jednotkového a regresného testovania. Do jednotkových testov boli pridané testy zvlášť pre každú novo podporovanú inštrukciu (a jej varianty). Nástroj regresných testov otestoval nové rozšírenie na 822 skutočných binárnych programoch, ktoré boli preložené pre architektúru x86 a manipulovali FPU zásobník. Nástroj vyhodnotil spätný preklad pre zvolenú testovaciu sadu za úspešný. Tretí nástroj sa v rámci projektu RetDec nazýva ako nočné testy. Tento nástroj otestoval nové rozšírenia na tisíckach reálnych spustitelných súboroch. Výsledky nočných testov zaznamenali výkonnostný pokles spätného prekladu. Nové spracovanie FPU registrov je priemerne šesťkrát pomalšie oproti originálnemu riešeniu. Avšak takýto výkonnostný pokles bol očakávaný a je akceptovateľný vzhľadom na komplexnosť novej optimalizácie.

Decompilation of Specialized and Advanced Instruction Sets

Declaration

Hereby I declare that this bachelor's thesis was prepared as an original author's work under the supervision of Zbyněk Křivka. The supplementary information was provided by Peter Matula. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Juraj Holub
May 19, 2020

Acknowledgements

I would like to express thanks to my supervisor Zbyněk Křivka, and to the people from Avast team which provided professional help. Namely I would like to thanks Peter Matula, Jakub Křoustek, and others for their consultation and valuable comments.

Contents

Chapter 1

Introduction

Nowadays, the process of analysing malicious software is an important part of informal technologies. One of the essential methods is decompilation of malicious binary programs. The decompilation is a complex process, and there are many projects with such an intent. The project RetDec intends to develop retargetable and flexible decompiler. The research proposes designs and tests new extensions for RetDec that improve the decompilation of advanced instruction sets for architecture x86. The support of FPU and SSE instruction set translation is extended. The new optimization for FPU register stack manipulation is designed. These extensions are tested in term of decompilation efficiency and quality.

The thesis, besides the general introduction, is split into eight logical Chapters. Chapter 2 introduces the concept of reverse engineering. Expressly, it presents the typical process of decompilation and explains the usual difficulties.

The Chapter 3 discusses the project RetDec, presents its architecture, and technologies applied in this decompiler (LLVM, Capstone, Keystone, and others).

Chapter 4 discuses the processor architecture Intel x86 supported by RetDec decompiler. The Chapter discusses the floating-point extensions FPU and SSE. The Chapter offers information that explains the problems and obstacles of this processor extension decompilation.

The Chapter 5 analyse actual state and potential obstacles of RetDec with the support of FPU and SSE instruction sets. It reviews these deficiencies in term of information obtained from the previous Chapter.

The Chapter 6 proposes a new advanced FPU optimization. It presents several methods for solving the designed task. It also proposes better support of SSE instruction set for RetDec decompiler. At last, the modifications to decompiler necessary for the implementation of the new extensions are discussed.

The Chapter 7 reviews the final implementation of the proposed extensions. The Chapter examines the newly proposed optimization implementation efficiency for various method of task solution, and it selects the best alternative. It also shows the result of the implementation for the advanced support of SSE instruction set.

The Chapter 8 tests implemented extensions in term of efficiency and functionality. The Chapter introduces three testing framework and shows the results of these tests.

Finally, the Chapter 9 summarize the entire work and point out the proposed and actual results.

Chapter 2

Reverse engineering

The concepts of reverse engineering, actual and historical reasons and conditions are presented at the base of the following book [8].

Reverse engineering, in general, is a method of obtaining information or blueprints about any object created by human. The idea of reverse engineering does not associate only to modern computer technologies. The concept has already existed in the era of the industrial revolution. The method has been typically used to examine commercially available technical products. Such a product has been physically decomposed, and each part was investigated to figure out its purpose. The process reveals the secrets of merchandise design without owning the original blueprints. Retrieved designs were commonly used to improve the product of the competing company.

The reverse engineering, in the domain of software technologies, is commonly named just reversing. Reversing is a fully abstract process of looking inside a computer program. There is not any physical object but only binary data, which are executable on a specific processor. Reversing requires knowledge of computers and software development processes.

The software reversing exploitation is useful for a variety of different purposes, the most significant are security-related reversing, and software development reversing. Both of these reversing purposes has discussed in the following Chapter. Software reversing considerably relates with a low-level layer of software architecture. Terms associated to low-level software describe this Chapter. At last, the Chapter introduces the concept of automatized reversing of software by the special program intended for this purpose.

2.1 Reversing in area of software security

In the area of software security, the reversing is used by both malware developers and by those creating security measures. In the area of computer security, the typical application of reversing is to analyses of malicious software. Also reversing of the cryptographic algorithms can discover implementation-dependent deficiency. Analysing of proprietary software program binaries, and searching for security vulnerabilities. Some of these applications are discussed in this Section.

2.1.1 Malicious software

In the beginning, a malicious software spread was fairly slow, and the precautions were much simpler because the human intervention was required to infect computer device. Internet network expansion dramatically changes the security character of computer technology.

Nowadays, nearly every computer on earth is connected to this virtual network. The malicious software spreads much faster, and the protection of computer devices is considerably more difficult. Computer attackers use reversing to capture vulnerabilities of the operating system or some other software. The reversing allows attackers to locate sensitive information about users, or even to take over control of the system. On the contrary, developers of antivirus software use reversing for analyzing malicious programs. They monitor every step of the malicious program to determine the damage it could cause and to find a possible method of protection.

2.1.2 Reversing cryptographic algorithms

Cryptography is a method of preserving information by transforming it into a human unreadable format. Protection of e-mails, credit cards information, or any other sensitive data are obtained by cryptography. [15]

The specific method of data transformation is called a cryptographic algorithm. Cryptographic algorithms in the manner of reversing purposes divide into two groups: key-base and restricted algorithms. The restricted algorithms are secret because the knowledge of the algorithm allows encryption and decryption of the message. Further, the key-based algorithms are typically public and well-known, but it uses the secret key. The secret key is necessary for encrypting and decrypting the message. Reversing try to analyses the restricted algorithm. The restricted algorithms are weak protection of information because exposing the algorithm makes it unsafe. Reversing of the key-based algorithm can look like ineffective. However, there are cases where it makes sense. Understanding of specific implementation can offer some interesting security details.

2.1.3 Digital Rights Management

In contrast with the past, providers of the most kinds of copyrighted materials turned their products into digital content. The products, like books, music, films, or games, are now available digitally. This produces huge benefits for customers, but also enormous complications for providers and content owners. The duplication of digital information, between consumers, is easy and unfortunately common practise. Commonly, the software owners wrap their product with additional copy protection software. Over the years, piracy protection technologies become more advanced, and this type of software are collectively called Digital Rights Management (DRM). DRM technologies are active protection, which decides about the availability of protected digital media. Software pirates use reverse engineering techniques to defeat DRM protection. Reversing of DRM technologies allow pirates to understand the inner secrets of software protection. Their goal is to find out how to modify it to disable the protection.

2.2 Reversing in area of software development

Reversing has as well great importance in the field of software development. Developers use reversing techniques to analyse partially documented or undocumented software, to improving competitive software, or to evaluating software quality and robustness.

2.2.1 Proprietary software documentation

Proprietary software documentation is almost always insufficient. Vendors of proprietary software can make a huge effort to provide adequate documentation. But customers typically encounter a problem with an unclear, or an undocumented solution. Developers in such a situation have to contact the vendor, which is a time-consuming solution. Differently, a developer can use reversing. Reversing can solve several of these problems with small effort. Typically, third-party software contains undocumented proprietary file formats, or networking protocols, which has to be reversed. Consider a famous Microsoft Word document format `.doc`. This format is also undocumented. But there is a lot of programs, which wants to support this format. Someone had to reverse the Microsoft Word file format, to provide support of it.

2.2.2 Development of competitive software

The development of a competitive product is, without a doubt, the most leading utilization of reverse engineering. Although, software engineering industry creates considerably complex products. Reversing whole software to create a competing product is almost always worthless. More often then not, it is effortless to create a new product from scratch or integrate the third-party libraries for more complex parts. Nevertheless, there are exceptions where the application of reversing is reasonable. Some extremely complex algorithms might be reversed, because of time-saving reasons. The legal aspect of reversing competitive software discuss the following book (see [8], Chapter 1, Section 'Is Reversing Legal?').

2.2.3 Software quality metrics

Software development includes techniques and metrics that evaluate software robustness, security and other general qualities of source code. Such techniques require access to the source code of the software. The disadvantage of the proprietary software is that there is no access to source code for customers. The users have to trust the vendor or apply revere engineering. Of course, reversing will never be as effective as analysing of source code itself, but it can be highly informative. The need for evaluating source code of critical software by users is even confirmed by large companies. For example, Microsoft gives access to Windows sources for large customers.

2.3 Low-level software

Generally, software is composed of layered architecture (see [8]). The bottom layer relates with the physical hardware. Hardware control provides assembly language. Usually, the assembler is different for each processor architecture and specific hardware device. Above physical layer is low-level software layer. It consists of an operating system and development tools such as compilers, linkers, or debuggers. The operating system encapsulates specific hardware architecture dependency, and development tools encapsulate assembly language dependency. Today, low-level software is encapsulated by another layer. At the top layer, there are some high-level languages, which greatly simplify development.

Reverse engineering strongly relates with low-level software layer. The reason is that the low-level details about the original program are typically the only pieces of information obtainable from the executable binary program. The Section introduces key aspects of low-level software.

2.3.1 Assembly language

Assembly language (or simply assembler) is a family of languages. Each processor architecture has its assembly language. And these languages usually significantly differ from each other. The knowledge of chosen architecture assembler is the necessary basis for the reverse engineer.

Assembler is a representation of processor instructions in a human-readable form. On the other hand, the machine code, or binary code is a representation of processor instructions in a sequence of bits, which is more effective for the processor itself. The machine code and assembler are just a different representation of the same object.

The illustrative example presents the translation process of assembler instruction to machine code. The instruction belongs to processor Intel 8086 (see [14], Chapter 12, Section 'x86 Instruction Encoding'). The process of encoding instruction example from Table 2.1 includes the following steps:

1. The unique code for instruction `PUSH` with 16-bit register operand is `0x50`.
2. The unique identification for register `CX` is `0x01`. Addition of `0x50` and `0x01` produces `0x51`.

Assembly instruction	Machine instruction
PUSH CX	0x51

Table 2.1: Mapping assembler instruction to machine code for Intel 8086 processor.

For the reversing purposes, the opposite (or backward) process of translation is important. A *disassembler* is a specific type of program that transforms the input binary executable program into a text file. Such a file contains assembler code equivalent to input machine code. It is a relatively simple process that maps binary code into assembler.

2.3.2 Compilers

As described in Section 2.3.1, the software consists of layered architecture, where the assembly language creates the bottom layer. High-level language is an abstraction over the assembly language. However, the high-level languages (for example Java, or C++) have to be transformed into machine code at the end. The reason is that machine code is the only language executable at the processor. The transformation performs a program called a compiler. The resulting machine code classifies into two categories. Either it is standard platform-dependent binary code, which is straight executable by a processor. Or it is a platform-independent format of code that is called bytecode. The specific program called a virtual machine process the bytecode and executes the specific hardware functionality.

Compilers of standard programming languages convert source code into machine code, which is directly executable at the processor (for example C or Pascal). During the conversion, a lot of optimizations over the machine code is applied. They increase program performance, but reversing of the optimized program is considerably more challenging. The reconstruction of the original high-level programming constructions from the optimized machine code is a complicated process. It is not an exception that the reconstruction is not achievable.

On the contrast, the second class of compilers transforms source code into bytecode (for example Java). In comparison to reversing of the standard binary code, the reversing of

bytecode is a completely different process. In general, it is a more straightforward process, because bytecode offers higher abstraction.

2.4 Decompilation

The process of reversing binary executable software into high-level programming language is called decompilation. The decompiler program reverses the executable binary file and produces high-level language code output. The Section introduces common decompiler architecture. It also describes the widely used techniques such as an intermediate representation of the program and static single assignment form of code representation. Finally, the Section discusses existing decompilers and compares them.

2.4.1 Intermediate Representations

Section 2.3.2 introduces the concept of compilers. The result of compilation is machine code that depends on the processor architecture (see Section 2.3.1). *Intermediate Representation (IR)* provides a generic set of instruction independent from architecture but with the ability to adequately represent the reversed program. Some decompilers transform source program to IR and just iteratively eliminate low-level detail. Other decompilers use more IRs, typically one for low-level representation and another for higher-level representation in later stages. Generally, the IR contains the following instruction set: assignment, push, pop, call, ret, branch, and unconditional jump (for more detail information about typical IR instruction set see [8], Chapter 13.). The IR instruction set is considerably smaller than the usual assembler instruction set. However, IR instructions typically represent complex expressions. For the representation of such complex expressions, the decompiler uses a structure called an expression tree. An expression tree effectively represents the sequence of arithmetic instructions. Expression tree provides reasonably more accessible input for generating high-level language expression.

Decompilers must create a *Control Flow Graph (CFG)* to reconstruct high-level control flow information from low-level IR. The CFG always represents the control flow of a single procedure. The reason for CFG representation is a simple transformation to high-level control flow constructs like loops and branches.

2.4.2 Static Single Assignment (SSA) form

SSA is a naming convention for variables in low-level program representation. Program code is in SSA form if each variable is a target of exactly one assignment statement. This lead to referential transparency, which means that for a variable with exactly one definition, the variable value is independent of its position in code. This knowledge is used for code optimizations such as data-flow analysis. For example, the dead code elimination in the fourth version of the GCC compiler is based on SSA intermediate representation, and an earlier version of the GCC compiler does not use SSA. The fourth version of the GCC compiler analyses around 40 % less of code lines then the equivalent optimizer pass with the third version of the GCC compiler. [27]

2.4.3 Typical Decompiler Architecture

The compiler is a special program, which transforms high-level programing language representation of program into a binary executable program. On the other hand, the decompiler

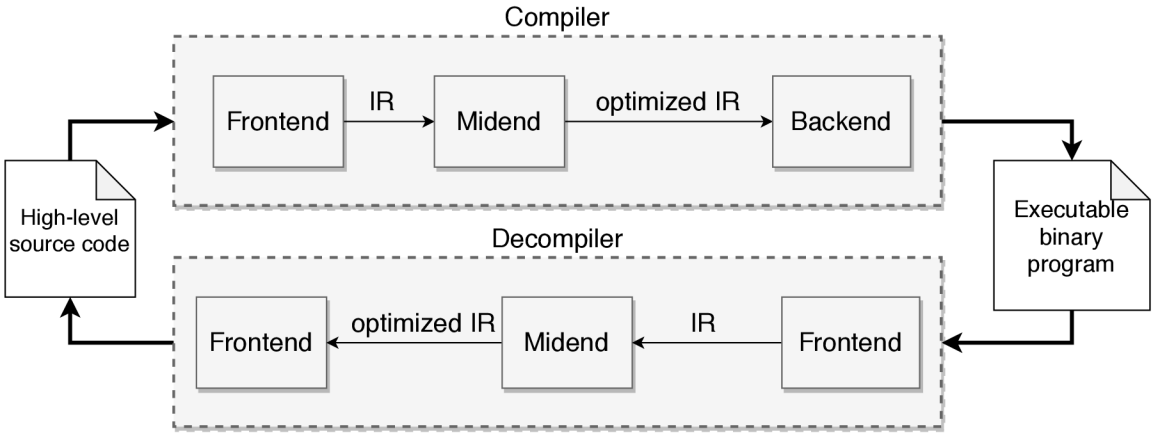

Figure 2.1: The typical architecture of compiler in contrast to decompiler.

reconstructs high-level language representation from some binary program. Decompiler typical architecture consists of similar parts as compiler architecture but in reversed order.

The compiler **frontend** is a component that parses input source code. The decompiler frontend decodes assembler instructions into some IR. In the beginning, there is only an input binary executable program, which has to be parsed. Frontend also provides semantic analysis because a lot of these assembly instructions hardly make sense individually. Instead, many of them create architecture-specific sequences. The output of frontend additionally represents control flow. The IR creates blocks of instructions where each block reference to some other part of code.

The midend of both architectures performs a set of optimizations over IR but with opposite goals. The compiler performs a code analysis to increase the performance speed of the final executable program. On the contrary, the decompiler code analysis aims to transform code into the more abstract form. At this stage, the decompiler eliminates the hardware concepts (registers and low-level conditional code) and converts it into the high-level programming constructions (variables, loops, branches and others). Typically for CFG analysis, the SSA notation is used (see Section 2.4.2). Data flow analysis can also provide information about data type propagation (e.g., the data type propagation of function return value). But before the data type propagation, the decompiler has to find out data types by itself. Registers often do not define data type information, but some instructions are data type sensitive. This information allows decompiler to scan for primitive data types. Decompilers also reconstruct complex data types. For such purposes, decompiler applies various advanced code scanning techniques:

- Certain registers are analysed to find out a memory address pointing to some data structure.
- The program commonly uses a hard-coded constant for manipulating data structure. Identification of such constant allows access to the analysed data structure.
- Detection of an array provides identification of standard loop iteration sequence and others.

Usually, the analysed program contains a lot of library functionality. Identification of such code is very beneficial. It provides very accurate information about data types without type-analysis process.

Finally, **the backend** takes this improved IR and generates output. Separation of output generation brings flexibility benefits. The generator produces various programing language output but always work with the same IR input. Such an approach allows an easy way to generate different programming language product.

2.5 Existing reversing tools

The Section introduces some decompiler projects and discusses their comparison. The goal of this thesis is to design new extensions for RetDec decompiler. As a result, the comparison of existing decompilers relates to this reference project (RetDec is detailed in Chapter 3).

IDA (Interactive Disassembler)[1] is cross-platform, multi-processor disassembler and debugger developed by Hex-Rays company. The part of the project is also the **Hex-Ray Decompiler**[2]. It generates human-readable C-like pseudocode. Currently, the supported input processor architectures are x86, x64, ARM32, ARM64, PowerPC, and PowerPC64. Nowadays, Hex-Rays offers one of the best decompilers on the market, but it is a paid tool, and because it is proprietary software, it cannot be used commercially.

Ghidra[3] is open-source reverse engineering framework developed by The National Security Agency of the U.S. Government. The framework was released in 2019, but it presents functionality comparable to the IDA project. It provides support of multiple processor architectures and operating systems. On the other hand, it is a robust tool, and it does not allow the use of individual framework tools separately.

[1] IDA project: https://www.hex-rays.com/products/ida/index.shtml
[2] Hex-Ray Decompiler: https://www.hex-rays.com/products/decompiler/index.shtml
[3] Ghidra project: https://www.nsa.gov/resources/everyone/ghidra/

Chapter 3

Retargetable Decompiler (RetDec)

RetDec decompiler project is a set of open-source reversing tools that are chained together. The goal of the decompiler is to become architecture, operating system and executable file format independent. The Chapter introduces RetDec architecture and software technologies used by this framework. Figure 3.1 shows the schema of RetDec and technologies used in each part of the decompiler. The core technologies used in decompiler are LLVM IR (see Section 3.1.1) and Capstone (see Section 3.1.2).

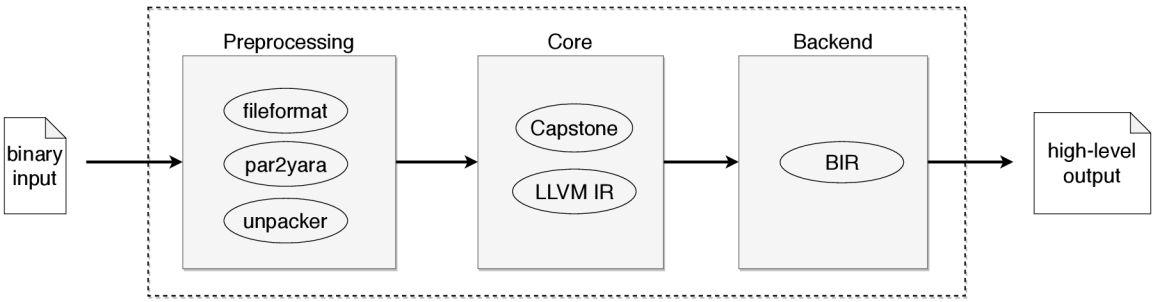

Figure 3.1: The architecture of RetDec and software technologies used by decompiler.

3.1 Technologies used in decomplier

Decompiler contains various open source technologies. The core of the decompiler design uses LLVM project. Capstone and Keystone libraries perform binary parsing and assembler generating. The Section explains details about these technologies.

3.1.1 LLVM IR

Low Level Virtual Machine (LLVM) IR defines common, low-level code representation. It is freely available under a non-restrictive license. LLVM representation uses an SSA form (see Section 2.4.2). LLVM code representation resembles an abstract RISC instruction set with high-level information for efficient analyses. For example, language-independent type system, control flow graphs or typed register set in SSA form. LLVM IR representation is independent of source language because it uses low-level instruction set slightly richer than common assembly language. It is important to note that LLVM IR has not intended to be a universal compiler IR. In particular, it does not provide high-level language features such as

classes, inheritance or exception handling. These features could be provided only indirectly. It also does not guarantee type or memory safety any more than assembly language. LLVM is complementary to high-level virtual machines such as in Self and Smalltalk. Benefits of LLVM are ideal for statically compiled languages like C and C++. [20]

The Section describes the basic concept of LLVM syntax and representation. For further details about LLVM syntax, see the official documentation of the project[1], or book [22]. LLVM representation of the program consists of the following data structures:

1. **Module**: The module is a top-level abstraction. It defines the content of an entire LLVM file. Naturally, the program can consist of multiple modules combined with the language linker. Each module consists of a sequence of functions. It also contains external entities, such as global variables, external function prototypes, or definition of data structures.

2. **Function**: The function representation is similar to C language syntax. There are function definition and a declaration signature syntax. The function declaration signature begins with declare keyword followed with the return type, name of the function and argument list. The function name is the global identifier and always begin with `@` prefix. Each argument consists of a data type and argument label. The argument label needs `%` prefix because it is a local identifier. The body of the procedure sets function definition, which explicitly breaks the function into a sequence of basic blocks.

3. **Basic Block**: The basic blocks form the CFG for the function. Each block begins with a unique identifier. Such identification can be explicitly defined, or an implicit numeral label is assigned. A block represents a sequence of instructions with a single entry point (first instruction) and a single exit point (last one). The terminating instruction changes control flow to another basic block or returns from the function.

4. **Instruction**: The instructions classification split instruction set into several classes: terminator instructions, binary instructions, memory instructions, and other instructions. Terminating instruction are explained previously, together with the basic block concept. Binary instructions perform general operations, for example, arithmetic operations, bitwise shifting, bitwise logical operations, etc.. Memory instructions read, write, or allocate memory. The remaining instructions cover mixed functionality (comparations, special constants, a function call, etc.). Typically, the instructions form a three-address code with two sources and one destination operand.

The code Listing 3.1 shows possible content of LLVM IR module. This module contains one definition of a function with a globally unique label `@foo`. The function has two arguments with explicit type definition (a 32-bit wide integer), and return data type is also an integer. The model also defines one global variable `@GLOBAL_VAR`. The body of the function contains three basic blocks: `%label0`, `%label1`, and `%label2`. The basic blocks `%label1` and `%label2` are terminating blocks of function, and basic block `%label0` ends with branching instructions. The expressive ability of instructions inside the basic block is very similar to assembler, but besides, it explicitly defines data types and variables. Lines 4 and 8 shows a characteristic load/store architecture.

[1] Official documentation of LLVM project: https://llvm.org/docs/

```
1 @GLOBAL_VAR = global i32
2 define i32 @foo(i32 %arg0, i32 %arg1) {
3   label0:
4     %flag = load i1, i32* @GLOBAL_VAR
5     br i1 %flag, label %label1, label %label2
6   label1:
7     %x = add i32 %arg0, %arg1
8     store i1 false, @GLOBAL_VAR
9     return i32 %x
10  label2:
11    %y = mul i1 %arg0, %arg1
12    return i32 %y
13 }
```

Listing 3.1: Example of LLVM IR syntax.

3.1.2 Capstone

Capstone is a disassembly framework for reverse engineering. It is an open-source project under a BSD license. The framework is compatible with multiple platforms. According to the official documentation (see [2]), the engine supports the following hardware architectures: x86 (16-bit, 32-bit, 64-bit), ARM, ARM64, MIPS, PowerPC, Sparc, SystemZ and XCore. Capstone has native support for the Windows operating system and it also supports Linux, OSX, iOS, Android, BSD, and Solaris. The disassembler engine provides architecture-independent Application Programming Interface (API). As shown in Figure 3.2, Capstone disassembler is complementary to Keystone assembler project (see [3]). Keystone is an assembler framework, which compile assembly instructions to binary. RetDec decompiler uses Capstone library for disassembling, and Keystone library as a testing framework.

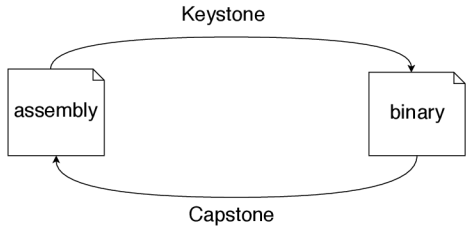

Figure 3.2: Complementary reverse engineering engines Capstone and Keystone.

3.2 Decompiler structure

The Section describes the architecture of the decompiler (for further details see [18]). The decompiler is structured into three main blocks. Every block consists of smaller units. Such a design makes the project units reusable because each unit works as the library with its interface. The three main blocks chains framework into the pipeline in the following sequence:

1. **Preprocessing** part unifies and analyses binary files. Unified binary files and extracted metadata are input for the core block.

2. **The core** block creates an IR and applies dozens of analyses and optimizations. The optimized IR is the output of the core block.

3. **The backend** block creates an *Abstract Syntax Tree (AST)*. It applies optimizations over AST and generates a final high-level representation.

3.2.1 The preprocessing

The structure of the preprocessing part describes Figure 3.3. The input of the preprocessing phase is a set machine code files. There are a lot of different formats for different platforms. *The File format library* analyses and unifies various file formats into uniform representation. Currently, the library supports the following machine code formats: ELF, PE, Mach-O, COFF, AR (archive), Intel HEX, and raw machine code.

Typically, the executable binary program additionally includes debugging data. This metadata creates a relationship between source code and binary data. Such relation is originally created for the debugger program, but also decompiler makes use of it. *The Debug Format library* parses this data and transforms them to debug representation used in next phases of decompilation. The library support DWARF and PDB format. [7]

The compiler that creates analysed binary program might use a tool so-called *packer*. The packing of binary files is done for two main reasons - code compression and code protection. As a consequence, the decompiler uses *Unpacker library*, which examines and identifies possible compression of the binary file. The library contains third-party tool YARA[2]. YARA tool helps identifies and detects binary patterns. The output of preprocessing is a metadata file in JSON format and uniform representation of machine code. The JSON metadata file contains information like compiler type and version, or processor architecture.

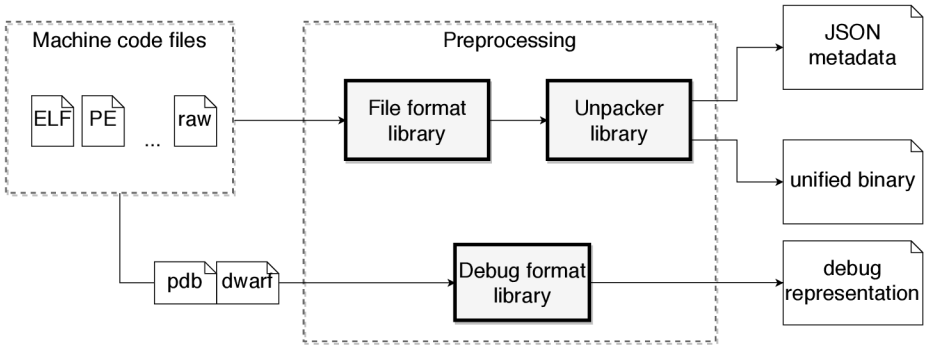

Figure 3.3: Preprocessing phase of decompilation.

3.2.2 The core

The functionality of the decompiler core illustrates Figure 3.4. The core block receives as an input JSON metadata, unified and unpacked machine code, and debug representation. Firstly, the machine code is transformed into LLVM IR. The transformation process performs the *Decoder library*. Decoder starts traversing binary data from the entry point of the program, and it follows the control flow of the reversed program. Capstone library maps the binary code into Capstone IR, which is transformed into LLVM IR.

[2]YARA tool: https://yara.readthedocs.io/

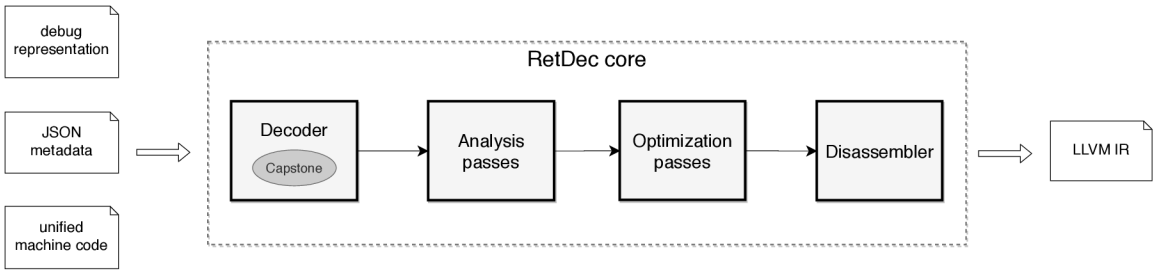

Figure 3.4: Transformation of binary code into LLVM IR by RetDec core.

The main part of the decompiler core performs modifications over obtained LLVM IR. The sequence of passes modifies IR. There are two types of passes: analysis and optimization passes. *Analysis passes* do not modify IR, but they obtain additional information. For example, analysis helps identify global variables, data types, function arguments, or return types of functions. *The optimization passes* iterate over IR and modify it. At last, the transformed LLVM IR is dissasembled. The result of this process is optimized LLVM IR, which is the output of the decompiler core block.

3.2.3 Backend

Backend does not operate with LLVM IR but transforms it into special IR so-called *Backend IR (BIR)*. This transformation is done because LLVM IR is a rather low-level representation similar to the assembler. On the other hand, BIR is a high-level representation based on the AST. AST allows better reconstruction of high-level control-flow patterns like conditional branches and loops. Backend restructures BIR when it identifies high-level constructs like *if-else*, *for-loop*, *while-loop*, *switch*, *break*, or *continue*.

Backend performs many high-level optimizations. It removes redundant variables, reduces constants in arithmetic expressions to simpler form. Backend optimization converts expressions to form more readable for programmers. Consider the following C source code:

```
sock_id = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)
```

As you can see, there are used constants defined in the standard C library. The meaning of this literals depends on the context. But the context of this literals is lost after disassembling. The decompiled code looks like:

```
var_fff = socket(2, 1, 6)
```

The optimizer searches for context literals and refactors them. Backend can generate the output in the following formats: C, Control-Flow Graph (CFG), or Call Graph. [19]

Chapter 4

Floating-point extensions of architecture Intel x86

In the beginning the Intel x86 architecture was developed to manipulate only with integer values. The floating-point calculation was possible to emulate through software but with a considerable performance penalty. As a consequence, a separate floating-point coprocessor unit was introduced. Nowadays, *the Floating-Point Unit (FPU)* is typically part of the main processor. The Section 4.1 explains details about FPU registers and instruction set, and Section 4.2 details calling conventions of function with floating-point values.

Next, the architecture extends support of the floating-point calculation with parallel processing of floating-point vectors. The Section 4.3 introduces SSE extension, the instruction set and their manipulations.

4.1 Floating-point unit x87

The Section details FPU registers and instruction set according to assembly language documentation for x86 processors (see [14]). FPU does not operate with x86 general-purpose registers because it contains its own set of registers. Floating-point instructions manipulate with these registers similarly to the stack data structure.

4.1.1 Floating-point registers

FPU has eight 80-bit general-purpose data registers named R0 through R7. These registers handling differs from manipulation of general-purpose data registers for integer evaluations. Hardware registers like EAX, EBX, ECX, etc., are direct operands of assembly instructions. But floating-point data registers forms an abstract stack data structure, and they cannot be accessed directly. Access to such hardware register is relative as explained in Section 4.1.2. As an addition to floating-point data registers, the unit has six special-purpose registers:

- **Control register** determines the rounding method and precision of FPU.
- **Status register** contains condition and exception flags. A three-bit field of status word so-called TOP identifies the register that is currently at the top of the stack.
- **Tag register** indicates the contents of data registers (valid number, zero, or special value like NaN, infinity, denormalized number, etc.). Register has a three-bit field for each data register.

- **Opcode register** contains the last executed instruction opcode.
- **Last instruction pointer register** points to the last executed instruction.
- **Last data (operand) pointer register** points to operands used by the last executed instruction (if the instruction has any data operand).

4.1.2 Register stack

FPU loads and stores values from the register stack where it performs floating-point arithmetic calculations. The x87 instructions evaluate arithmetic expressions in *postfix* form due to stack evaluation advantages of this form. Consider the following *infix* expression:

$$(A + B) * C$$

And equivalent *postix* expression:

$$AB + C*$$

The *postfix* format does not require parenthesis to override precedence rules. The transformation algorithm from infix to postfix form is not a subject of this thesis.

Figure 4.1 shows the abstraction of FPU stack data manipulation. Stack operands are labelled ST(0) through ST(7), where ST(0) label points to data register on the top of the stack. The value of TOP points to data register labelled ST(0). A *push* (alternatively *load*) instruction decrements TOP and moves the content of operand to ST(0) register. Overriding of existing data in the stack generates a floating-point exception. Decrementation of TOP with value 0 (ST(0) points to R0) leads to underflow TOP value to 7 (ST(0) points to R7). A *pop* (alternatively *store*) instruction moves content of ST(0) register to operand and increment TOP. Incrementation of TOP with value 7 (ST(0) points to R7) leads to overflow TOP value to 0 (ST(0) point to R0).

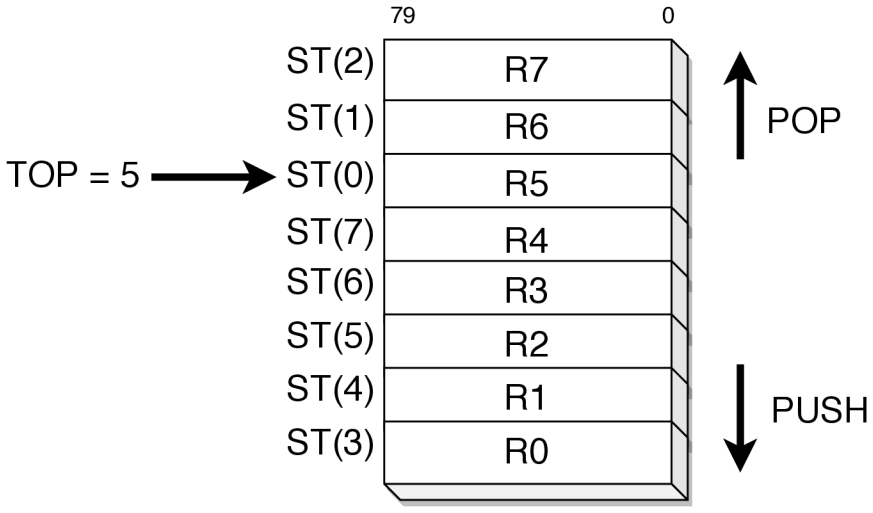

Figure 4.1: FPU data register stack abstraction. [14]

4.1.3 FPU instructions

The floating-point instruction operands allow only one addressing mode. Operands are always in general-purpose data registers. An instruction can inherently manipulate with register stack (implicit push, or store). The set of floating-point instructions contains the following basic instructions categories:

- Basic arithmetic instructions.
- Constant loading instructions.
- Data transfer instructions.
- Exponential, logarithmic and trigonometric instructions.
- Data comparsion instructions.
- FPU control instructions.

4.2 Floating-point conventions for calling functions

The procedure, alternatively function, or subroutine is a fundamental abstraction for general-purpose procedural programming languages. The program is divided into various parts, and such part of code could be used several times. The procedure abstraction eliminates repetitions of program code segments and allows their reusability. An execution of program subroutine is known as a procedure call. The Section explains details about procedures with the floating-point interface and their calling conventions.

4.2.1 Standardization of procedure calls

In past, procedure calling interface differed for every operating system or compiler. It led to compatibility problems. Nowadays, there is an effort on the standardization of the procedure calling interface. The calling conventions determine the following low-level details:

- For interaction between caller (calling program) and callee (a subroutine), the program reserves specific hardware registers.
- The system of arguments transfer between calle and caller. Arguments are typically passed within registers, on the stack or in shared memory.
- Caller has to pass arguments in the right order. Typically, arguments are passed from the first to the last or in the reversed order.
- Arguments could be passed by value or by reference.
- The result of procedure execution (return value) has to be passed to the callee.
- The method of stack pointer restoration after the procedure execution.

Architecture	Calling convention	Passing registers
16 bit	cdecl pascal fastcall	AX
	watcom	Inconclusive.
32 bit	cdecl stdcall pascal fastcall thiscall	ST(0)
	watcom	Inconclusive.
64 bit	Windows	SSE registers
	Linux, BSD, Mac OS	

Table 4.1: Usage of registers for passing floating-point values across calling conventions for x86 architecture.

4.2.2 Architecture x86 calling conventions

Architecture x86 has three modes: 16-bit, 32-bit, and 64-bit mode. The bit-wide of mode specifies wide of registers, memory address, etc. in bits. The 16-bit and 32-bit mode have usually calling conventions independent on operating systems. Instead of the operating system, the calling convention is defined by the compiler. The thesis assumes Microsoft, Borland, Watcom and Gnu compilers brands. On the other hand, the 64-bit mode has a default calling convention for each operating system, while other calling conventions are rare in 64-bit mode. The thesis considers calling conventions for Windows, Linux, BSD, Unix and Mac OS X operating system. [9]

For **16-bit mode**, there is calling conventions so-called *cdecl*, *pascal*, *fastcall* and *watcom*. Watcom is inconclusive because the method of registers usage depends on options in effect. All others calling conventions do not return floating-point value in ST(0). The called function is expected to allocate space for value in memory and write the return value to this address. The address where is the result stored is passed in AX register. [5]

System V (see [30]) application binary interface for a **32-bit mode** of x86 architecture defines the usage of floating-point stack registers. In case that procedure returns a floating-point value, then the value is stored in ST(0) register. It does not matter if the floating-point value is in the representation of single or double precision. If the procedure does not return floating-point value, then register ST(0) must be empty. Also, register ST(0) must be empty before every procedure call. Registers ST(1) through ST(7) are unused in the standard calling sequence of the procedure with floating-point arguments or return value. The standard defines that these registers must be empty before and upon every procedure call. Most used calling conventions for architecture x86 in 32-bit mode are *cdecl*, *stdcall*, *pascal*, *fastcall*, *thiscall* and *watcom*. System V standard follows all of these conventions except *watcom*. Watcom same like in 16-bit mode is inconclusive.

As described in Section 4.3, the x86 platform over time introduced extensions the Streaming SIMD Extensions (SSE). SSE adds new instructions and registers, which also manipulates floating-point values. System V for **the 64-bit mode** of architecture x86 defines that function with floating-point arguments, or return value does not pass these values through FPU registers (see [10]). Preferably, it uses SSE registers. The convention

is followed in Linux, BSD, Unix, and Mac OS X operating system. The Windows operating system uses different conventions then System V. Nevertheless, the Microsoft function calling convention with floating-point values also uses only SSE registers, and it does not specify any convention for FPU registers. [25]

The x86 architecture function calling conventions are summarized in Table 4.1.

4.3 Streaming SIMD Extensions (SSE)

Over the years, the architecture x86 includes multiple extensions, which operate in mode commonly called *the Single Instruction Multiple Data (SIMD)*. These technologies have dedicated to the parallel processing of data. The first extension was MMX, and it created support of basic SIMD processing for the integer arithmetic. The successor of MMX is *the Streaming SIMD Extension (SSE)*, which is a set of the hardware improvements. SSE regularly increases the CPU ability of SIMD processing. It increases integer arithmetic with new registers and instructions, and it extends processor with floating-point SIMD facilities.

Eventually, SSE multiple time upgrades hardware facilities and functionality. The original SSE gradually evolves to SSE2, SSE3, SSE4, AVX (Advanced Vector Extensions), AVX2, and AVX-512. However, this research discusses the general aspects of this hardware extension, and it uses the general label SSE. When the specific version of SSE has discussed, the version name is used.

Following Subsections details SSE, and also explains the standard of compilers built-in functions. The SIMD technology details are obtained from the publication of the x86 assembler programming guide (for further details see [17]).

4.3.1 Idea of SIMD processing

This Subsection introduces the general concept of SIMD technology. The hardware unit allows executing the same operation on the collection of the data elements at the same time. Typically, the performed operations are basic arithmetic computation, for instance, subtraction, addition, multiplication, division, bitwise operations, and conversion. Such parallelism achieves specific interpretation of the register, or memory location content.

To illustrate considers Figure 4.2 that shows 32-bit width register intended to integer data processing. The register can hold the single 32-bit integer value, but SIMD allows to reinterpret it as two 16-bit integers, or four 8-bit integers. The processor handles each subsequence of register separately, but simultaneously. The hardware supports the usual service of data processing for each bit pattern individually, for example, integer overflow, underflow, rounding, and others.

The effectivity of SIMD strongly depends on the compiler. The compiler must correctly detect and splits the program data that can process simultaneously. The Intel provides documentation for the developers of the compilers that allows them to optimize the efficiency of the resulting program binary. The documentation also contains a guide for the Intel built-in functions, as closely explained in Section 4.3.4.

4.3.2 SSE register set and data types

SSE extends the 32-bit architecture of x86 with eight general-purpose registers, which are 128-bit width. The 64-bit architecture appends another eight registers. They are labelled `XMM0` through `XMM15`. These registers allow carrying floating-point values. The original

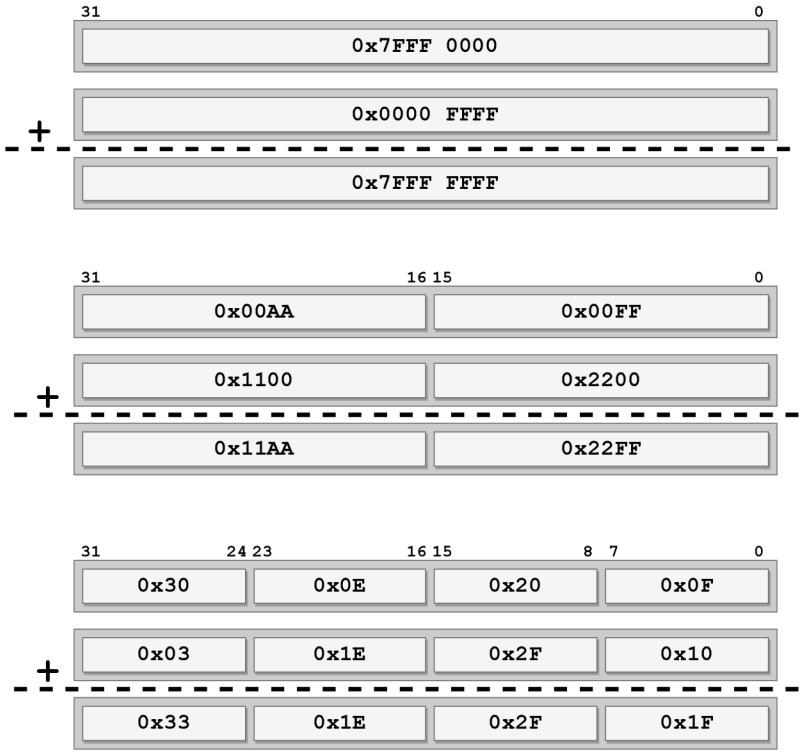

Figure 4.2: SIMD processing demonstration for integers addition with 32-bit width registers. [17]

SSE supports single-precision, but SSE2 starts supporting double-precision floating-point data. As a contrast to FPU, SSE uses direct addressing of registers. SSE does not operate with register as a stack structure (see Section 4.1.2).

SSE supports various integer and floating-point data types. These data types create two categories:

- **Scalar data types**: The `XMM` register or memory location holds 32-bit (single-precision), or 64-bit (double-precision) floating-point value. As mentioned before, the `XMM` register is 128-bit width, but SSE supports maximally 64-bit width floating-point data types. Suppose that SSE performs some double-precision floating-point scalar operation and saves the result value into specific `XMM` registers. In such a case, SSE saves the result value into lower 64-bits of a destination `XMM` register, while the rest of the register content stay untouched.
- **Packed data types**: The register or memory location holds four 32-bit or two 64-bit floating-point values. Also, it can hold integers with various bitlength: 16 bytes, 8 words, 4 doublewords, or 2 quadword integer values.

4.3.3 Instruction set

As detailed in Section 4.3.2, SSE support two categories of data types. In consequence, also SSE instruction set split instructions into these two categories: instructions with *scalar*, or *packed* operands. SSE floating-point instruction has typically four modes:

- Scalar Single-precision mode with instruction suffix `SS`.

Compiler	Built-in ADDPS
GCC	`v4sf __builtin_ia32_addps(v4sf, v4sf)`
Clang	`v4sf __builtin_ia32_addps(v4sf, v4sf)`
Microsoft VS	`__m128 _mm_add_ps(__m128, __m128)`
Intel	`__m128 _mm_add_ps(__m128 a, __m128 b)`

Table 4.2: The example of the built-in function for the assembly instruction that performs vector addition of four float operands.

- Packed Single-precision mode with instruction suffix `PS`.
- Scalar Double-precision mode with instruction suffix `SD`.
- Packed Double-precision mode with instruction suffix `PD`.

SSE support following basic functionality for floating-point data (always in packed and scalar version): data transfer, arithmetical and logical operations, and data type conversions. For packed mode only, it allows data shuffle, data unpack or element insertion.

SSE supports only packed processing of integer operands. The integer processing instructions offer almost the same functionality, as for packed floating-point operands, but each instructions differs four modes (byte, word, doubleword, and quadword). At last, SSE supports text string processing. It performs string compares, and string length calculation. It can accelerate a pattern search and replaces algorithm. For full information about all instructions read Intel Software Developer Manual (see [1]).

4.3.4 Compilers built-in functions

Compilers built-in functions (also known as intrinsic functions) are C/C++ functions that allow calling assembler instruction in the high-level programming language. Built-in functions are equivalent to the inline assembler. However, the built-in functions offer benefits of high-level programming: better code readability, or advantages of debugging. In general, the developers use these functions when they need some very low-level assembly functionality. In the case of the architecture x86, the built-in functions offer instructions related to MMX, SSE or AVX. Typically, these instructions work with vector operands. [1]

The method of use built-in functions in the program source depends on the compiler. The GCC compiler offers built-in functions for architecture x86 with 32-bit mode and 64-bit mode (see [28]). Clang compiler offers very similar built-in functions with the same syntax (see [29]). On the contrary, the Microsoft Visual Studio defines their built-in functions with different syntax (see [26]). The Microsft includes a definition of the x86 built-in functions in header `<intrin.h>`. However, the Intel defines manufacturer-specific built-in functions in header `<immintrin.h>`. The Intel also offers a detailed guide [1] for compiler developers that describes the semantic meaning of these built-in function. These Intel built-in functions are most general equivalent because of their definition shares between all common compilers. Table 4.2 illustrate the example of the built-in function for the assembly instruction that performs vector addition.

[1] The Intel Intrinsics Guide: https://software.intel.com/sites/landingpage/IntrinsicsGuide

Chapter 5

Review of RetDec deficiencies

The chapter reviews the design and implementation details of RetDec core. The chapter analysis the flaws and deficiencies of the current RetDec core design. The process of translation Capstone IR into LLVM IR is introduced, and specific problems and restrictions with the translation of FPU instruction set are presented.

5.1 Decoder of Capstone into LLVM IR

The section generally explains the model of RetDec for the mapping of disassembled instructions into LLVM IR. The section demonstrates Decoder library (briefly introduced in Section 3.2.2) that controls this process. The possible modes of translation and reasons for such a design are presented. [23]

5.1.1 Translation modes

RetDec decompiler does not aim to entirely translate the semantic meaning of the disassembled machine code. The goal of the decompiler is to generate easy and understandable C/C++ output. Such output can be effectively analysed by a reverse engineer. Decoder library performs mapping of assembly instructions in four modes:

1. **Full translation mode**: Instructions are simple enough to capture their full semantics with a sequence of LLVM IR. This mode captures mostly the core instruction set (basic arithmetic and data transfer instructions).

2. **Pseudo assembly functions**: Some instructions cannot be represented through LLVM IR sequence. For example, instruction `FWAIT` checks for pending floating-point exceptions. Library represents instruction like a self-explanatory pseudo function `@__asm_fwait()`.

3. **Partial translation mode**: Some assembler instructions are too complex in LLVM IR representation. As an example, consider instruction `FXSAVE [addr]`, which saves the state of FPU, MMX, SSE units, and their registers to 512-bytes in memory to address `addr`. Entirely mapped instruction produces dozens of LLVM instructions. On the contrary, partial conversion mode produces pseudo assembly function as described previously. But this mode also explicitly informs about storing 512-bytes to memory.

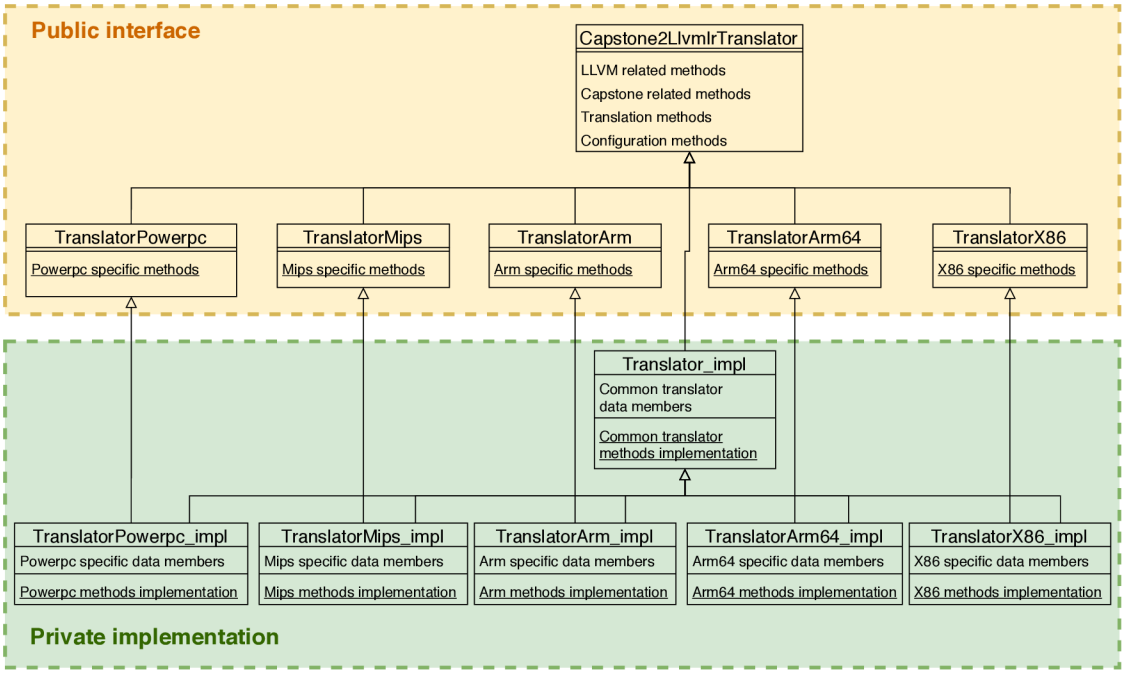

Figure 5.1: Class diagram of the decoder library (*Capstone2LlvmIr* [23]) implementation in RetDec core.

4. **Ignore instruction**: Instruction `FNOP` performs no FPU operation. The occurrence of this instruction in C output is unnecessary, and decompiler skips it.

RetDec project is developed mainly for decompilation purposes. For some other use-cases, where full semantic meaning is needful, there are other projects like QEMU[1] or McSema[2], which allow better alternative in such situations.

5.1.2 Decoder library structure

As mentioned in Section 3.2.2, the first unit of RetDec core is Decoder library, which transforms Capstone IR into LLVM IR. The library design illustrates the class diagram in Figure 5.1. The library *encapsulates* design into two parts:

- **Public interface**: The library provides public headers without data members and implementation.
- **Private implementation**: The implementation of the decoder (source code and hidden headers) is hidden for the library users.

Also, the design divides library, by *inheritance*, into the two types of modules. Such library design allows simple and flexible expandability by another processor architectures.

- **Generic translator**: Module with common translation interface and implementation that is independent from processor architecture. It includes LLVM, Capstone, general translation and configuration related methods and data members (*Translator_impl*, see Figure 5.1).

[1]QEMU project: https://www.qemu.org/
[2]McSema project: https://github.com/lifting-bits/mcsema

- **Processor architecture-specific translator**: It includes individual modules for each supported processor architecture by a decompiler. For example, there are specific modules for ARM, MIPS, or x86 architecture *TranslatorArm_impl, TranslatorMips_impl, TranslatorX86_impl*, see Figure 5.1).

5.1.3 Translation process

The translation process of Capstone IR into LLVM IR work similarly for each specific processor architecture module of the decoder library. The Section demonstrates the translation process of two x86 instructions: `sub eax, ebx` and `je 0x1000`.

In the first place, decoder creates the instance of the translator module for x86 architecture. The constructor of decoder instantiation accepts an empty LLVM IR module. It initializes a Capstone engine and other internal structures. LLVM IR module initializes with *the architecture-dependent environment*:

- *The specific global variables* have been created. They represent concrete hardware registers. Listing 5.1 shows an example of generated global variables. Some registers have internally divided into bit sequences with special meaning. Such special bit is commonly called a flag. A typical example represents the EFLAG register (see [14], Chapter 2). This register consists of flags that crucially control the operation of the processor. Due to the importance of these bits, the equivalent global variables in LLVM IR are generated.

```
1  @_asm_program_counter = internal global i64 0
2  @eax = internal global i32 0
3  @ebx = internal global i32 0
4  ; ...
5  @st0 = internal global x86_fp80 0xK00000000000000000000
6  @st1 = internal global x86_fp80 0xK00000000000000000000
7  ; ...
8  @cf = internal global i1 false ; The Carry flag (CF)
9  @pf = internal global i1 false ; The Parity flag (PF)
10 @ac = internal global i1 false ; The Auxiliary Carry flag (AC)
11 @zf = internal global i1 false ; The Zero flag (ZF)
12 @sf = internal global i1 false ; The Sign flag (SF)
13 @of = internal global i1 false ; The Overflow flag (OF)
```

Listing 5.1: Example of the specific architecture-dependent global variables.

Special attention belongs to the global variable `@_asm_program_counter`. The global variable value is updated at the beginning of each translated assembly instruction. It stores integer value that denotes an address of the current reversed assembler instruction in the program. Every sequence of LLVM IR that represents one assembly instruction begins with such a store operation. For example, instruction at address 1234 begins with the following store operation:

```
store volatile i64 1234, i64* @_asm_program_counter
```

- Except for architecture-specific global variables, the module also generates *the control flow pseudo functions* (see Listing 5.2). These pseudo functions represent control flow operation: function call, return from the function, branching, and conditional branching. Naturally, LLVM IR provides build-in instructions for such functionality. But this stage of decompilation cannot use them, because they accept targets only

in the form of a label. However, the labels reconstruction is subject of advanced optimization at later stages of decompilation. Currently, the library has only integer address of destination targets.

```
1  ; address is 64-bit integer because of 64-bit architecture
2  declare void @__pseudo_call(i64 %addr)
3  declare void @__pseudo_return(i64 %addr)
4  declare void @__pseudo_branch(i64 %addr)
5  declare void @__pseudo_cond_branch(i1 %condition, i64 %addr)
```

Listing 5.2: Example of the specific architecture-dependent (x86-64) control flow pseudo functions.

- A module defines *an architecture-specific data layout* string that determines the format of stored data in memory. As an example, the data layout string specifies if the data lays out in big-endian, or little-endian form. It specifies the size of the memory address pointer, or it defines an alignment of various integer and floating-point types.
- At last, it specifies individual settings intended for particular processor functionality. For instance, the pseudo functions that manipulate FPU stack (see Section 5.2.1).

After module initialization, the translator traverses over binary data and transforms them. Put simplistically, the Decoder processes particular binary data of specific size and at the exact address and the result of transformation places at the relevant position in LLVM IR module. The transformation process utilizes Capstone engine. To illustrate, Capstone receives the binary data `29 d8` (hexadecimal represention) of size 2 bytes at address `0x1000`. The library performs the following steps:

1. Capstone disassembles `29 d8` into `sub eax, ebx`. But, disassembler offers much more metadata about reversed instruction than just textual assembler representation. In particular, it provides detail information about operands like which processor supporting units are active for this instruction (e.g. SSE, AVX), or general info about reading and writing into registers.
2. Each Capstone instruction has a unique ID. The module defines the mapping for each Capstone ID into specific translation routine. The module executes the corresponding routine that implements LLVM IR template for the given Capstone IR. Capstone ID is mapped into one of the three types of translation routine:
   a) Mapping one ID into one specific routine. In case of similar instructions, the more IDs have mapped into one routine. Such routine implements full, or partial semantic meaning of assembler instruction.
   b) Mapping Capstone ID into particular pseudo assembly generation routine. These type of routine generates pseudo assembly function call, but with additional information about instruction data flow.
   c) The last possibility is that there is no specific service routine for Capstone ID. In such a situation, the translator executes universal routine that generates pseudo assembly function call.
3. After execution of the selected translation routines, the two representative assembly instructions would transform into LLVM IR sequence in Listing 5.3.

```
1   ; ...
2   ; sub eax, ebx
3   store volatile i64 4096, i64* @_asm_program_counter
4   %0 = load i32, i32* @eax
5   %1 = load i32, i32* @ebx
6   %2 = sub i32 %0, %1 ; eax - ebx
7   %3 = and i32 %0, 15
8   %4 = and i32 %1, 15
9   %5 = sub i32 %3, %4
10  %6 = icmp ugt i32 %5, 15
11  %7 = icmp ult i32 %0,%1
12  %8 = xor i32 %0, %1
13  %9 = xor i32 %0, %2
14  %10 = and i32 %8, %9
15  %11 = icmp slt i32 %10, 0
16  store i1 %6, i1* @az
17  store i1 %7, i1* @cf
18  store i1 %11, i1* @of
19  %12 = icmp eq i32 %2, 0
20  store i1 %12, i1* @zf
21  %13 = icmp slt i32 %2, 0
22  store i1 %13, i1* @sf
23  %14 = trunc i32 %2 to i8
24  %15 = call i8 @llvm.ctpop.i8(i8 %14)
25  %16 = and i8 %15, 1
26  %17 = icmp eq i8 %16, 0
27  store i1 %17, i1* @pf
28  store i32 %2, i32* @eax
29
30  ; je 0x1000
31  store volatile i64 4096, i64* @_asm_program_counter
32  %0 = load i1, i1* @zf
33  call void @__pseudo_cond_branch(i1 %0, i32 4096) ; 32-bit target addresS
34  ; ...
```

Listing 5.3: Result of translation.

The resulting LLVM IR sequence for the subtraction instruction describes the full semantic of equal assembly instruction. The subtraction itself represents lines 4 to 6 (see Listing 5.3), but the rest of the IR sequence describes evaluating of flags in EFLAG register. The second translated instruction is a conditional jump (or branching instruction). As described earlier, the control flow pseudo function was generated instead of LLVM IR build-in branching instruction. The simple generation branch to target address `0x1000` (or 4096 in decimal format, which is used in Listing 5.3, line 33) instead of a particular label does not require the expertise of the entire module context. Designed Decoder library translates without control flow context, which is much more straightforward and effective in this stage of the reversing process.

5.1.4 Advanced instruction sets of x86 architecture

As shown in Figure 5.1, there is support for x86 architecture in Capstone into LLVM IR decoder. Decoder of x86 module supports entire architecture instruction set with all specialized extensions. However, the decoder translates most of the advanced instruction sets (like FPU, SSE, MMX, or AVX instruction sets) into pseudo assembly functions. In other words, there are not service routines for the majority of x86 extension units. It is important to note that we cannot evaluate such translation support as a huge decompiler

```
1  .data
2      float0 REAL8 0.0
3      float1 REAL8 1.0
4      float2 REAL8 3.1415
5  .code
6      FLD float0          ; ST(0) = R7
7      FLD float1          ; ST(0) = R6, ST(1) = R7
8      FADD ST(0), ST(1)   ; ST(0) + ST(1)  ==  R6 + R7
9      FLD float2          ; ST(0) = R5, ST(1) = R6, ST(2) = R7
10     FADD ST(0), ST(1)   ; ST(0) + ST(1)  ==  R5 + R6
```

Listing 5.4: Example of x87 assembler FPU stack usage.

deficiency. The x86 processor family is CISC assembly architecture, which means that the majority of instructions provides a very specific functionality, and they are rarely used in general. Yet, larger support of partial (or even full) semantic translation of these extensive instruction sets is beneficial.

The goal of this research is to extend support of advanced instruction, especially for FPU and SSE units. Currently, the decoder supports 65 % of FPU instruction set and 20 % of SSE instruction set. An instruction is marked as a supported when there is a service routine that implements it, and also there is at least one unit test for this instruction. The extension aims to add full support of FPU instruction set and SSE floating-point instructions. Full semantic description of packed SSE instructions (see Section 4.3.3) is beyond the facility of LLVM IR. Still, this thesis investigates the possible improvements of scalar instruction representation.

5.2 LLVM IR optimization for FPU instruction set

As described earlier (see Section 4.1.2), FPU instructions manipulate operands through the register stack. Such relative indexing of data registers leads to the problems with mapping of disassembled instructions to LLVM IR semantic model. The section describes currently implemented optimization that tries to solve this problem. At last, the section analyses defects and potential disadvantages of the current implementation.

5.2.1 Semantic model of FPU instruction set

Straightforward mapping of FPU instruction with register operands into LLVM IR sequence template is not possible, therefore correct mapping requires more advanced analysis. To illustrate, let us consider the assembler code in Listing 5.4. An instruction at line 8 manipulates with registers labelled as ST(0) and ST(1). These labels refer to concrete hardware data registers (let us assume that they are R6 and R7). Following instruction, at line 9, loads constant to FPU register stack and decrements the value of an actual stack top. As a result, labels ST(0) and ST(1) now refer to different data registers than before. At last, an instruction at line 10 is syntactically identical to instruction at line 8, but this time they refer to different hardware registers (R5 and R6). This leads to a problem because, without the stack top context, disassembled code represents different registers with equal labels. Therefore, RetDec semantic model cannot represent shown assembler instruction at line 10 as LLVM IR sequence in Listing 5.5.

```
@st0 = internal global x86_fp80
@st1 = internal global x86_fp80
; ...
; FADD ST(0), ST(1)
%op0 = load x86_fp80, x86_fp80* @st0
%op1 = load x86_fp80, x86_fp80* @st1
%res = fmul x86_fp80 %op0, %op1
store x86_fp80 %res, x86_fp80* @st0
```

Listing 5.5: Incorrect LLVM IR of instruction `FADD ST(0), ST(1)`.

```
@fpu_stat_TOP = internal global i3 0
; ...
; FADD ST(0), ST(1)
%0 = load i3, i3* @fpu_stat_TOP
%1 = add i3 %0, 1
%2 = call x86_fp80 @_pseudox87DataLoad(i3 %0)
%3 = call x86_fp80 @_pseudox87DataLoad(i3 %1)
%4 = fmul x86_fp80 %2, %3
call void @_pseudox87DataStore(i3 %1, x86_fp80 %4)
```

Listing 5.6: Correct LLVM IR of instruction `FADD ST(0), ST(1)`.

To represent correct FPU registers, load/store instructions have to contain information about FPU stack top. Decoder library, which maps Capstone IR into LLVM IR, translates these instructions into pseudo functions similarly to control-flow pseudo functions (as described in Section 5.1.3). Decompiler has metadata about these pseudo functions and passes them into later analyses. The core of RetDec contains dozens of optimizations passes (see Section 3.2.2). One of them reconstructs FPU stack context for each function. It replaces pseudo functions call for load/store instruction with particular FPU registers. The decoder generates four pseudo functions:

- The pseudo function stores float value into FPU register denoted by `TOP` value: `void @_pseudox87DataStore(i3 %TOP, x86_fp80 %ST)`
- The function stores tag value (see Section 4.1) of FPU register denoted by `TOP` value: `void @_pseudox87TagStore(i3 %TOP, i2 %TAG)`
- The pseudo function returns the float value of the register from FPU stack with position `TOP`: `x86_fp80 @_pseudox87DataLoad(i3 %TOP)`
- The pseudo function returns the value of the tag register adequate to FPU register denoted by the `TOP`: `i2 @_pseudox87TagLoad(i3 %TOP)`

5.2.2 The current state of FPU stack optimization

As explained in Section 5.2.1, the decompiler core optimization contain the all necessary data to restores FPU physical register operands. The problem is that existing implementation maps pseudo functions straight to instructions for each reversed function without knowledge about function control flow. The current optimization assumes that FPU stack

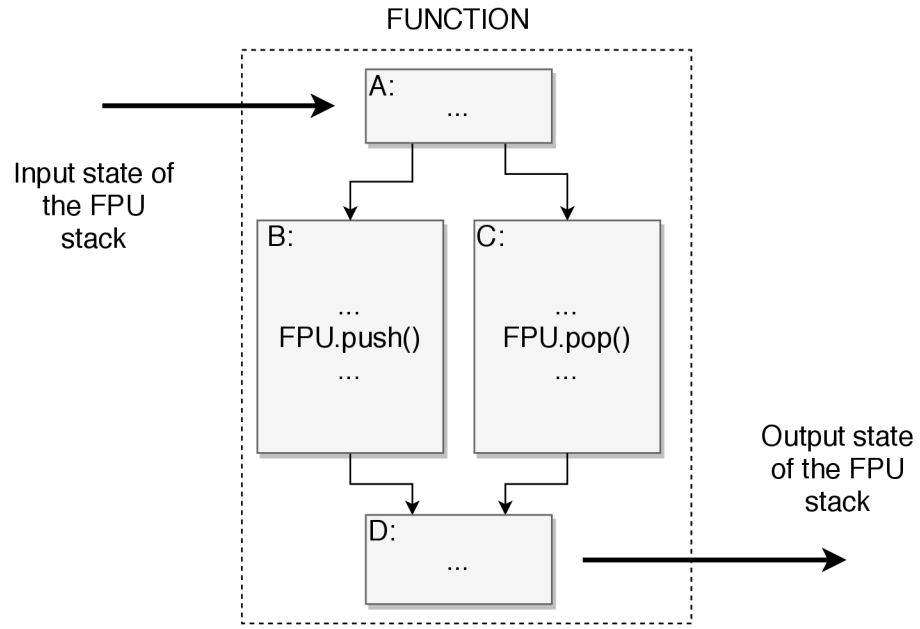

Figure 5.2: Illustration of the incorrect manipulation with FPU stack value.

is empty at the beginning of every function. The optimization performs one sequential traverse over instructions and tracks FPU stack position. Each occurrence of a pseudo function replaces with the calculated value. A potential error of such solution demonstrates Figure 5.2, which presents the control flow between basic blocks in some function. The basic block *B* pushes the value to FPU stack, and block *C* pops a value from the stack. However, these two basic blocks form alternative branches. The control flow always visits only one of these blocks. As a consequence, the stack state at the exit block is nondeterministic. However, the sequential pass through basic blocks ignores control flow dependency. As a result, such an algorithm does not detect this type of error.

Chapter 6

Proposed extensions of RetDec

The chapter proposes extension for RetDec support of architecture x86 advanced instruction sets (FPU and SSE). Section 6.1 introduces the idea of SSE translation extension in the manner of vector and scalar floating-point instructions. Section 6.2 outlines advanced FPU instruction set optimization that reconstructs register stack.

6.1 The x86 decoder extends of the advaced instruction set.

As described in Section 5.1, RetDec core contains the library that decodes Capstone into LLVM IR. As mentioned in Section 5.1.4, the decoder offers many upgrade opportunities for the advanced instruction sets of the architecture x86. This section proposes semantic of the floating-point related instruction within SSE.

6.1.1 SSE extension

Section 5.1.1 describes the dilemma of semantic complexity for some advanced instructions. The decompiler tries to simplify the description of the instruction hardware behaviour. As explained in Section 4.3, SSE allows manipulating the single register operand as a vector of the values. For example, consider a simple vector addition of two registers, as illustrated in Figure 4.2. Such a simple hardware operation is nontrivial for the software emulation. Section 4.3.4 explains built-in functions that are equivalent to the C inline assembler. The proposed extension translates SSE vector instructions into the call of the built-in functions. On the contrary, the scalar instructions can translate with a full semantic meaning. Figure 6.1 illustrates both variants of instruction translation. The vector addition of floats (instruction `ADDPS`) translates into the built-in function (see line 4).

```
1  ;; ADDPS
2  %0 = load i128, i128* @xmm0
3  %1 = load i128, i128* @xmm1
4  %2 = call i128 @_mm_add_ps(i128 %0,
   i128 %1)
5  store i128 %2, i128* @xmm0
```

```
1  ;; ADDSS
2  %0 = load float*, float** @xmm0_f3
3  %1 = load float, float* %0
4  %2 = load float*, float** @xmm1_f3
5  %3 = load float, float* %2
6  %4 = fadd float %1, %3
7  store float %4, float* %0
```

Figure 6.1: The example of translation SSE floating-point addition for packed and scalar instruction.

On the other side, the scalar float addition (instruction `ADDSS` in Figure 6.1) does not use a built-in function. The instruction uses lower 32-bits of two SSE registers. However, extracting of these register subsequences for each instruction call is inefficient, considering a large number of this specific instruction occurrence in the program. This problem partially solves the following decompiler extension proposal (see [16]). The proposed extension generates global views to specific subsequences of SSE registers. Listing 6.1 illustrates the example of such views. The advantage of these register views is that the decoder generates all of them at the beginning of translation. Next, the specific translated instructions use them as operands.

```
1 @xmm0 = internal global i128 0; register XMM0
2 @xmm1 = internal global i128 0; register XMM1
3 @xmm0_f3 = internal global float* bitcast (i8* getelementptr (i8, i8* bitcast (i128*
      @xmm0 to i8*), i64 12) to float*) ; pointer to lower 32-bits of XMM0
4 @xmm1_f3 = internal global float* bitcast (i8* getelementptr (i8, i8* bitcast (i128*
      @xmm0 to i8*), i64 12) to float*) ; pointer to lower 32-bits of XMM1
```

Listing 6.1: The example of the generated views to SSE register subsequences. [16]

6.2 Advanced reconstruction of FPU stack

This section proposes FPU instruction set optimization extensions for RetDec with the best ratio of positive impact for the decompilation quality to the difficulty of their implementation. The newly designed optimization resolves problems of currently used optimization that is described in Section 5.2. The section expects basic knowledge of numerical linear algebra and matrices (for required information, see Elementary Linear Algebra [4], Chapter 1).

6.2.1 Function-based optimization

Reconstruction of FPU stack is always resolved separately for each function. Without exception, FPU stackis empty at the beginning of each function execution. The premise is based on the analysis of the standards for function calling conventions (see Section 4.2, and Table 4.1). All conventions proclaim that before every function call, the stack must be empty. The precondition is true even when a function contains parameters with any floating-point data type. FPU stack is at the end of the function call either empty, or it holds a single value. The stack contains one value in case of function with the floating-point return type, but only in case of the 32-bit mode for the x86 architecture. Obtaining this information for the decompiled program is easily achievable because RetDec provides metadata about the module architecture, and functions calling convention.

As a result, the optimization analyses each function as a separate context. The state of FPU stack at the entry and exit of this context is predetermined. This approach split the program module into independent and smaller parts. In case of the optimization fails for one function, that is an advantage because others function analysis results are not affected. Also, if a function does not manipulate FPU register, then the optimization skips the whole function and reduces analysis duration.

6.2.2 Function Control Flow Graph analysis

Proposed optimization tries to find out the state of FPU stack at the entry point of each basic block. With this information, the optimization can sequentially traverse all instruc-

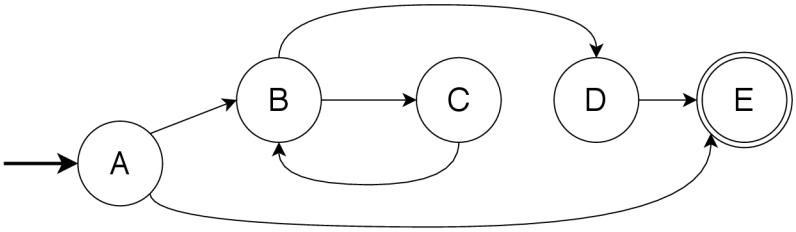

Figure 6.2: Example of possible CFG for some function.

tion in the basic block and analyses FPU stack related instructions. These instructions optimization divides into two categories:

- The first category modifies FPU stack. They read and increment, or decrement stack top and saves the new value of top. Every stack modification has internally saved with context to the modification position in a basic block. This information uses the second group of instructions.
- The second category read and write to FPU registers (and their tags). As described in Section 5.2, the register load/store operation represents a pseudo function call. The optimization mark pseudo call with the current state of the stack related to it.

At this point, the replacement of the pseudo call has not performed because the current state of the stack is only an offset relative to the start of the basic block. Optimization calculates the exact stack value at the block entry at the next step. To find out stack state at the block entry, an optimization forms *a linear equation system* that reflects CFG of basic blocks. Each basic block defines two variables:

- BB_{in} : Stack value at the entry into the basic block BB.
- BB_{out} : Stack value at the entry into the basic block BB.

To demonstrate, suppose CFG in Figure 6.2. It shows basic blocks dependencies for potential function. Let us consider that function holds floating-point arguments and returns the floating-point value. Then optimization transforms the CFG of function into the following Equation system (6.1).

$$\begin{aligned} B_{in} - A_{out} &= 0 \\ C_{in} - B_{out} &= 0 \\ B_{in} - C_{out} &= 0 \\ D_{in} - B_{out} &= 0 \\ E_{in} - D_{out} &= 0 \\ E_{in} - A_{out} &= 0 \end{aligned} \tag{6.1}$$

As described in the previous step, each basic block is analysed separately. Independently on the stack state at the entry point, the analysis can determine the difference between input and output stack value. Such a computed difference produces one more equation for each basic block. For the illustrative CFG example (see Figure 6.2), the analysis extends system with Equations (6.2), where the value A_{Δ} is a particular constant difference resolved by a traverse of basic block A.

$$
\begin{aligned}
A_{out} - A_{in} &= A_{\Delta} \\
B_{out} - B_{in} &= B_{\Delta} \\
C_{out} - C_{in} &= C_{\Delta} \\
D_{out} - D_{in} &= D_{\Delta} \\
E_{out} - E_{in} &= E_{\Delta}
\end{aligned}
\tag{6.2}
$$

The following step encapsulates function context by definition of stack state at the function entry and exit point (detailed in Section 6.2.1). The optimization creates two and more equations (more in case that function contains multiple terminating basic blocks). For our example, it creates Equations (6.3). The stack is empty at the function entry point (A_{in}). The stack holds single value at the end of function terminating block (C_{out}) because of the illustrative function (see Figure 6.2) pass floating-point return value through FPU stack.

$$
\begin{aligned}
A_{in} &= 0 \\
E_{out} &= 1
\end{aligned}
\tag{6.3}
$$

CFG for each particular function contains a different number of nodes and edges. The produced system has more equations than variables. Such a system of linear equations is called *overdetermined*. An overdetermined system typically has no solution but in this case, there is a high probability of exactly one solution. RetDec expects that the analysed binary file is the product of some unknown compiler. In the majority of the cases, such compiler produces a valid binary. If the system has no solution, then the examined code contains errors. Optimization transforms extracted equations system to a matrix Equation (6.4). Equation (6.5) universally represents this system, where the $\mathbf{A}$ is a matrix of system coefficients, and $\overline{\mathbf{x}}$ is a vector of unknowns.

The matrix $(\mathbf{A}|\overline{\mathbf{b}})$ is called the *augmented matrix*. The analysis determines the *rank* of matrix A and the rank of augmented matrix $(\mathbf{A}|\overline{\mathbf{b}})$. The system has exactly one solution if these ranks are equal. In such a case, the optimizer solves Equation (6.6). Section 6.3 discuss various methods for solving the overdetermined system of linear equations. Calculated vector $\overline{\mathbf{x}}$ contains values of FPU stack at the entry and end of each basic block.

$$
\begin{pmatrix}
0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
-1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
A_{in} \\ A_{out} \\ B_{in} \\ B_{out} \\ C_{in} \\ C_{out} \\ D_{in} \\ D_{out} \\ E_{in} \\ E_{out}
\end{pmatrix}
=
\begin{pmatrix}
0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ A_{\Delta} \\ B_{\Delta} \\ C_{\Delta} \\ D_{\Delta} \\ E_{\Delta} \\ 0 \\ 1
\end{pmatrix}
\tag{6.4}
$$

$$
\mathbf{A}\ \overline{\mathbf{x}} = \overline{\mathbf{b}}
\tag{6.5}
$$

$$\overline{\mathbf{x}} = \mathbf{A}^{-1}\ \overline{\mathbf{b}} \tag{6.6}$$

After analyzing of all functions, optimizer replaces previously marked pseudo function calls with specific registers load/store instructions. Naturally, substitution is applied only for successfully analysed functions. The concrete register index is calculated by addition of previously obtained stack offset relative to the start of the basic block, and value of stack at the block entry calculated in vector $\overline{\mathbf{x}}$.

6.3 Methods for solving linear systems

A system of linear equations in matrix notation (see Equation (6.7)), where the $\mathbf{A}$ is a matrix of system coefficients, $\overline{\mathbf{b}}$ is the right-hand side, and $\overline{\mathbf{x}}$ is a vector of unknowns. In our application, the system is overdetermined because the number of equations is bigger then number of unknowns (see Section 6.2.2). In general, there are many applications where the consistent overdetermined system is measured (it has exactly one solution). Such type of problem is typically solved by approximation methods. There are various methods suitable for that problem. The Section discusses the most common approximation method called the least squares solution, and three optimized modification of this method. [6]

$$\mathbf{A}\overline{\mathbf{x}} = \overline{\mathbf{b}} \tag{6.7}$$

6.3.1 Least squares solution

Let a given linear system (6.7) be overdetermined. Since the exact solution is improbable, the method looks for a vector $\overline{\mathbf{x}}$ that is as close as possible to the exact solution. The $\mathbf{A}\overline{\mathbf{x}}$ is an approximation to $\overline{\mathbf{b}}$, and $||\overline{\mathbf{b}} - \mathbf{A}\overline{\mathbf{x}}||$ is an error in that approximation. Such an $\overline{\mathbf{x}}$ vector is called *the least squares solutions* of the system (6.7). The vector $\overline{\mathbf{b}}$ is called *the least square error*. In case that a linear system is consistent, then the least squares solutions are equivalent to the exact solutions. In other words, the least squares error is zero. For our purposes, we expect that this is our case. [4]

Matrix *decomposition* is a process that solves linear systems in a numerically stable way. Additionally, it can provide matrix inversion or reveals matrix rank. The following subsection describes three commonly used decompositions (Cholesky, QR, and SVD decompositions). Cholesky and QR decompositions transform a system of linear equations (6.7) into a system with *an upper triangular coefficient* matrix: $\mathbf{U}\overline{\mathbf{x}} = \overline{\mathbf{b}}$. The SVD decomposition transforms such a system into *a diagonal coefficient* matrix: $\mathbf{D}\overline{\mathbf{x}} = \overline{\mathbf{b}}$. As a result, the transformed system is easier to solve and even with higher accuracy by back substitution. [6]

6.3.2 Cholesky decomposition

There is the least squares solution for each linear system (6.7) if and only if it is a solution of the associated *normal system* (6.8). [21]

$$\mathbf{A}^T\mathbf{A}\overline{\mathbf{x}} = \mathbf{A}^T\overline{\mathbf{b}} \tag{6.8}$$

Therefore, to find the least square solution, the system can be reduced to a system of the normal equations. The normal system (6.8) is always consistent, and all solutions of (6.8) are least squares solutions of (6.7). The normal equations system solves *Cholesky*

decomposition (also called *factorization*) that transforms the nonsquare matrix $\mathbf{A}$ into an upper triangular matrix $\mathbf{U}$ for the system (6.7), where holds $\mathbf{A} = \mathbf{U}^T\mathbf{U}$. The normal equations system solves the following method:

1. Calculate $\mathbf{A}^T\mathbf{A}$ as a $\mathbf{C}$.
2. Cholesky decomposition of matrix $\mathbf{C}$ into $\mathbf{U}^T\mathbf{U}$.
3. Original system $\mathbf{A}^T\mathbf{A}\overline{\mathbf{x}} = \mathbf{A}^T\overline{\mathbf{b}}$ results into $\mathbf{U}^T\mathbf{U}\overline{\mathbf{x}} = \mathbf{A}^T\overline{\mathbf{b}}$
4. Transformed system has a least square solution $\overline{\mathbf{x}} = (\mathbf{U}^T\mathbf{U})^{-1}\mathbf{A}^T\overline{\mathbf{b}}$

The Cholesky factorization is generally the fastest method of solving least squares, but it is numerically unstable. The method is sensitive to inaccuracy in matrix $\mathbf{A}$. Small inaccuracy of matrix can lead to large changes in the solution. Generally, this method offers half accuracy on the contrary to other methods. [21]

6.3.3 QR decomposition

The *QR decomposition* is a very important matrix transformation that splits general matrix $\mathbf{A}$ to *an upper triangular matrix* $\mathbf{R}$ and *an orthonormal matrix* $\mathbf{Q}$. Orthonormal matrix has columns *orthogonal* to each other and its *Euclidian norm* equals to 1. If $\mathbf{A}$ is substituted by $\mathbf{QR}$, then for each $\overline{\mathbf{b}}$ in the system $\mathbf{A}\overline{\mathbf{x}} = \overline{\mathbf{b}}$ is a least squares solution given by Equation (6.9). [6, 4]

$$\overline{\mathbf{x}} = \mathbf{R}^{-1}\mathbf{Q}^T\overline{\mathbf{b}} \tag{6.9}$$

The obtaining least squares solution by QR decomposition is more suitable for numerical computation than the Cholesky decomposition (see Section 6.3.1). In general, the QR decomposition guarantees numerical stability because it minimizes errors caused by machine roundoffs. The QR method is more accurate in comparison to Cholesky, but such an advantage is not beneficial when the system is well-conditioned (our system expects to be well-conditioned and deterministic). In most cases, the QR decomposition takes twice more time then Cholesky. [21]

6.3.4 SVD decomposition

The Singular Value Decomposition (SVD) is a method that decomposes a matrix into numerous matrices, revealing important properties of the source matrix. The detailed algorithm of SVD is not presented, but in general, the decomposition obtains *pseudoinverse* matrix $\mathbf{A}^\dagger$ (called the More-Penrose pseudoinverse matrix) from matrix $\mathbf{A}$ of the system (6.8) (see [13]). The method proceeds in the following steps:

1. Change linear equations system $\mathbf{A}\overline{\mathbf{x}} = \overline{\mathbf{b}}$ into normal system $\mathbf{A}^T\mathbf{A}\overline{\mathbf{x}} = \mathbf{A}^T\overline{\mathbf{b}}$.
2. Normal system has a least square solution $\overline{\mathbf{x}} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\overline{\mathbf{b}}$.
3. The matrix $(\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T$ is obtained by SVD as a pseudo inverse matrix $\mathbf{A}^\dagger$.
4. Transformed system has a least squares solution $\overline{\mathbf{x}} = \mathbf{A}^\dagger\overline{\mathbf{b}}$

The SVD is a robust method, and it evaluates roughly about 10 times slower in comparison to other methods. However, it is numerically stable, and it offers additional properties about the matrix. It can also handle greater inaccuracy. [21]

6.3.5 Summary

This section analysed the various method for matrix decomposition. The matrix decomposition offers a numerically stable method for solving overdetermined systems of linear equations by the method of least squares. The efficiency and accuracy of three decomposition methods have discussed. The fastest method is Cholesky decomposition, second is QR decomposition, and the slowest method is SVD. On the contrary, the most accurate method is SVD, followed by QR decomposition, and the most inaccurate method is Cholesky decomposition. For our application (see Section 6.2.2), the accuracy lack is acceptable because the solved system produces only integer results. The efficiency is the most important parameter because the optimization expects huge functions with hundreds or even thousands basic blocks. Another factor is the need for matrix rank revealing because the analysis computes it due to evaluation of system consistency. Matrix rank evaluation can be a separate process, but the SVD and QR decomposition allow to solve this task. Merge of these tasks can lead to higher optimization efficiency.

Chapter 7

Implementation of extensions

Chapter 6 proposes the new extensions of advanced instruction set support for the RetDec (FPU and SSE). This chapter summarizes the result of new instructions implementation. Chapter 6 also designs the new FPU stack optimization that transforms FPU stack into a linear system. This chapter details implementation details, and Section 7.2.2 select solver of a linear system in term of implementation efficiency.

7.1 Decoder support of advanced instruction sets

Section 5.1.4 shows the available support of advanced instruction set for RetDec. The research focuses on FPU and SSE. In consequence, the full FPU instruction set is implemented. Each instruction is defined by specific translation routine, which generates LLVM IR sequence (see Section 5.1.3). Every instruction, with each possible type of operands, is covered with a unit test (see Section 8.1.2). The new implementation is already included in a stable version of the decompiler.

In the case of proposed SSE extension (see Section 6.1.1), the instruction core is implemented (arithmetic, data manipulation, and comparison instructions). New translation subroutines are not fully covered with unit tests. The unit test framework used in the project offers insufficient functionality for reasonable validation (see Section 8.1.2). For example, it does not allow to check modification of register subsequences, as required with SSE instructions.

7.2 Linear algebra library

The optimization proposed in Section 6.2.2 transforms FPU stack into a linear system. For such an approach, RetDec requires efficient module with the support of the linear algebra evaluation. The implementation of the linear algebra module within RetDec project is a robust task. The module has to support basic matrix algebraic evaluation and, in addition, advanced methods (decomposition of overdetermined systems and rank evaluation for nonsquare matrices). The design, implementation, and maintaining of such a robust module is too challenging for the entire project when it is used only by one optimization. As a consequence, the project includes an external library for these purposes. The third-party project must be C++ open-source library that supports compilation with CMake[1] because the entire RetDec uses this tool to build all components.

[1] CMake project: https://cmake.org/

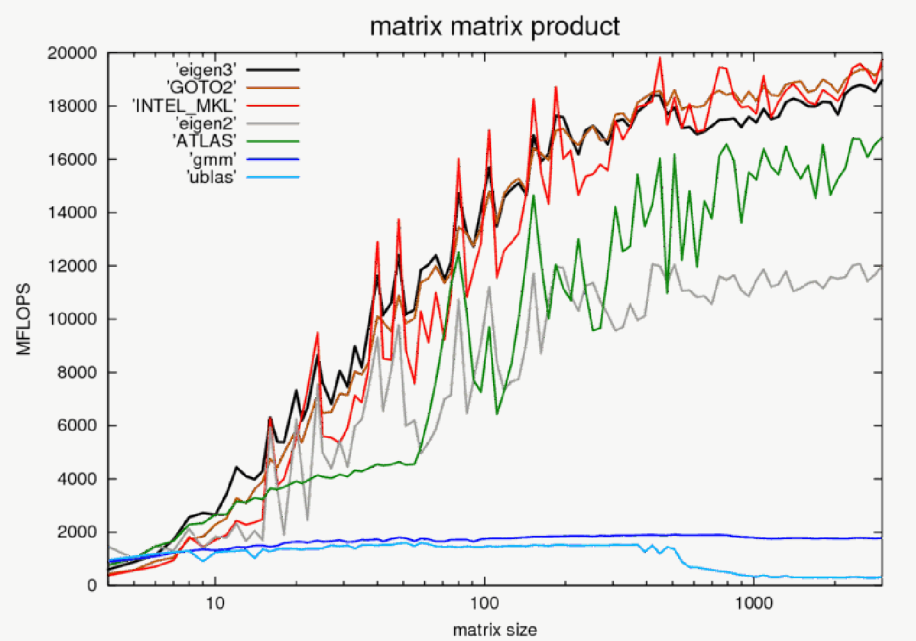

Figure 7.1: Benchmark comparison of math libraries with Eigen (retrieved from official project benchmark [11]).

7.2.1 Eigen3 project

The chosen library is called Eigen3 (see official project web page [11]). It is a multiplatform (GCC, Clang, x86 SSE, ARM Neon, PowerPC) library for linear algebra: basic vector and matrix manipulations, evaluating of various decompositions (Cholesky, LU, QR, SVD), solving of linear systems, least squares solutions, eigenvalues, or singular values. The library analyses object size and optimize evaluation methods for them. Another optimization increases performance with the classification of matrices to dense and sparse. Figure 7.1 shows a benchmark comparison of various algebraic libraries (GOTO[2], ATLAS[3], and others[4]) for operation with two matrices. Unit MFLOPS means millions of arithmetic operations per second. The graph shows Eigen3 as an efficiency favourable tool for large matrices, which is our use case. RetDec source repository does not include Eigen3 sources, but it links it as an external dependency (for further detail see CMake external projects documentation[5]).

7.2.2 Selection of solver for a linear system

Designed FPU optimization proposed three methods for determining the least squares solution of overdetermined linear systems:

- **Cholesky** decomposition: see Section 6.3.2.
- **QR** decomposition: see Section 6.3.3.
- **SVD** decomposition: see Section 6.3.4.

[2]GOTO project: http://www.csar.cfs.ac.uk/user_information/software/maths/goto.shtml
[3]ATLAS project: http://math-atlas.sourceforge.net/
[4]Eigen3 benchmark: http://eigen.tuxfamily.org/index.php?title=Benchmark
[5]CMake external project https://cmake.org/cmake/help/latest/module/ExternalProject.html

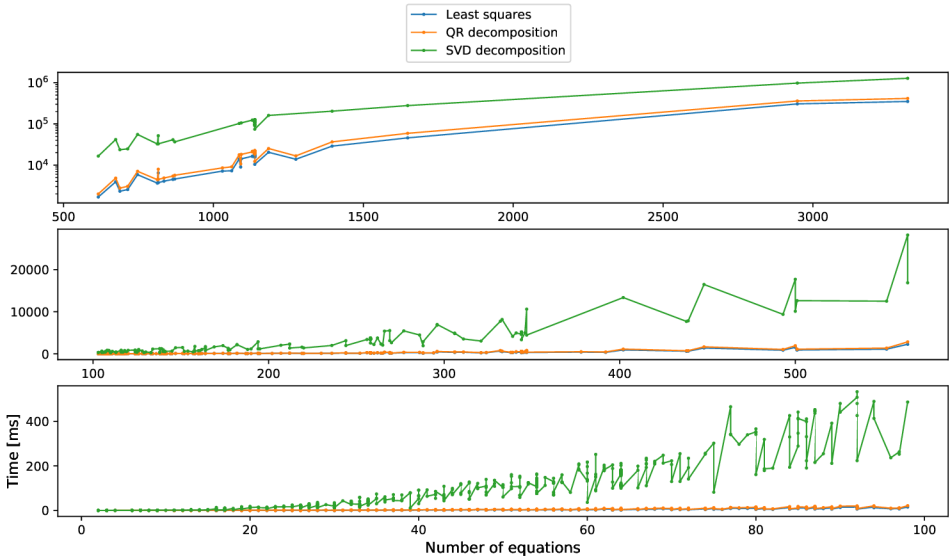

Figure 7.2: Comparison of computation timing

Eigen3 library supports all three decomposition methods. An empiric experiment, described below, measured the performance effectiveness of each method, to select the decomposition algorithm with the best ratio of performance efficiency and sufficient solution accuracy. The experimental data were real executable binary files retrieved from the project regression test framework database (the framework and testing binaries are explicitly discussed in Section 8.3). The experiment included 822 binaries. These binaries contain over 4000 functions that manipulate FPU. For each individual function, the experiment measured the time of the linear system decomposition. The experiment measured each function three times (separately for each proposed method). The accuracy of the least squares solution evaluated RetDec regression test framework, which decided about decompilation success (framework supported accuracy criteria are presented in Section 8.3). For better experiment evaluation, the measured data split into three classes by the number of equations in the particular function system:

- **Low**: For each function, the number of equations in the system is maximally 100. An experimental set consists of 3 173 functions.
- **Medium**: For each function, the number of equations in the system belongs to $(100, 600)$. An experimental set consists of 644 functions.
- **High**: For each function, the number of equations in the system is more than 600. An experimental set consists of 276 functions.

	Cholesky	**QR**	**SVD**
Low	1.29 ms	1.88 ms	43.89 ms
Medium	264.34 ms	345.11 ms	3 948.95 ms
High	18 741.14 ms	23 332.06 ms	125 831.64 ms

Table 7.1: Average time of linear system solving for different methods.

The Figure 7.2 shows result of experiment. Independently on the number of equations of the system, the duration of evaluation Cholesky decomposition is very slightly better than QR decomposition. For small systems, the difference is even unimportant. However, the SVD indicates significant performance lack that is unacceptable for large systems. Table 7.1 shows the average evaluation duration separately for each method and experiment class. The SVD evaluation duration is around 10 times slower in comparison to the other two methods. From this point of view, the SVD is not a suitable method. In terms of solution accuracy, the testing framework successfully decompiles all binaries for Cholesky and QR decomposition. The SVD solves system with the highest accuracy. But the duration of evaluation is unacceptable, and for huge systems (category with 600 and more equations) the testing framework terminates decompilation with a failure status. The accuracy of SVD cannot be qualified as a benefit because the system least squares solution should approximate to integer results. As a result of the experiment, the implementation of the optimization selects the QR decomposition. In addition to efficient system solving, QR implementation offers an evaluation of matrix rank with a small performance lack (the optimization must calculate matrix rank because of the determination of system inconsistency).

7.2.3 Summary

This chapter summarizes the results of FPU and SSE instruction set implementation in decoder library. Next, due to FPU optimization, the suitable library for linear algebra was selected. The experiment with real binary programs decompilation compares and select the most efficient implementation of the proposed system solver method (see Section 7.2.2). Next chapter tests the functionality of new extensions.

Chapter 8

Testing of extension

RetDec project uses various methods of testing that try to determine decompiler errors. The project contains three main utilities that test the decompiler with different goals. The proposed and implemented extensions, described in Chapters 6 and 7, are tested in the manner of decompilation success, quality, and performance. Section 8.1 describes the project unit testing framework. Section 8.2 introduces the night test framework, and Section 8.3 describes the regression testing framework. Each section summarizes the testing results of the new extensions. The notions and definitions related to software testing are retrieved from [24].

8.1 Unit tests

RetDec applies unit testing as a fundamental testing process. The basic concept is to test the decompiler subcomponents separately and independently. RetDec is mainly created in C++ language, and it uses Google Test framework[1] that offers unit testing in this programming language (separate testing of the classes and their methods). In the case of RetDec core (see Section 3.2.2), each analysis or optimization defines individual class. Typically, such a class contains an adequate unit test suite.

The approach of unit testing brings many advantages. Firstly, it allows to test part of the program independently from the rest. Also, it is reasonably a fast testing technique: RetDec contains 6 819 unit tests, and their execution takes around 6.5 sec for a specific CPU[2]. Generally, it allows to detects bugs faster and easier. Moreover, each unit test offers a sort of documentation for developers because it shows concrete examples of class instantiation and their methods use.

8.1.1 FPU optimization

Unit testing is an essential key for the agile methodology such as *Extreme programming* (see [12]). This methodology applies a technique commonly named *Test-Driven Development (TDD)*. The development of the extended optimization for FPU instruction set (see Section 6.2) applies the key TDD principle:

1. Write unit tests based on the specified requirements, and executes them to check that their fails.

[1] Google Testing and Mocking Framework: https://github.com/google/googletest
[2] Experimental CPU: Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz

2. Continuously implement individual requirements and checks improvements in the tests success ratio.

The input of FPU optimization contains an LLVM IR module and various configuration properties (processor architecture specification, calling conventions for each function, and others). The unit test framework must prepare an input LLVM IR module and the entire environment. Next, it executes unit under tests, which is the new FPU optimization. The output of the optimizations is modified LLVM IR module and return code (success or fail). The framework compares the modified LLVM IR module and the return value with the expected preprepared module and the return value. The detailed approach of FPU unit test writing is described in Appendix A.

8.1.2 Decoder of Capstone into LLVM IR

Decoder is a considerably large component, and it is inherited by specific CPU architecture modules (see Section 5.1, Figure 5.1). Each architecture module contains its own unit test suite (see Table 8.1). In general, RetDec tries to create at least one unit test for each instruction of every architecture. Within FPU instruction set extension for the x86 architecture, the test suite is extended by cases for each instruction and its variant (architecture bit width and all possible instruction operands type combinations).

The unit test framework takes as an input a single LLVM IR module with a single instruction, the architecture specification and values of registers or memory values that are necessary for the instruction under test. Next, the framework interprets LLVM IR and checks postconditions (expected values of registers and memory).

Architecture	**Unit tests [-]**
Arm	422
Arm64	419
Mips	572
PowerPC	766
x86	1749

Table 8.1: The number of the unit tests for each architecture module supported in Decoder library.

8.2 Night tests

RetDec night tests framework is the largest testing tool used in the entire project. It tests thousands of program samples. The goal is to check decompilation success and quality. The project uses night tests to monitor impact of development changes. The test suite consists of the large pallet of different program samples. The entire testing process is significantly time-consuming, and it can take dozens of hours. The program test samples are divided into two groups:

- **Source files in C**: These programs are compiled by the framework for all supported architectures with different options of optimization used by the specific compiler. The advantage of these samples is that it allows to evaluate decompilation output quality. The quality is evaluated by the comparison of the original and decompiled C source.

- **Binary executable files**: These test samples are large and complex programs. Typically, it is some obtained malware binary program. The goal is to determine decompilation performance efficiency.

The night tests execution splits into two phases. The first phase performs the decompilation of each sample, and it logs the running process. The second phase evaluates obtained decompilation logs (the size of these files are generally in gigabytes unit) and the results are presented as a web page. The web page summarizes each decompilation phase success (return code), the decompilation output quality, time summary, memory consumption summary, and others. The night test samples and the framework itself are proprietary, and only internal developers of the project have access to this tool. The resulting web page allows comparing different night test executions. In the case of comparison, the web page shows which decompilation phases improve or worsen.

8.2.1 FPU optimization evaluation

Figure in Appendix B.1 presents the result of night tests with the new FPU optimization. The new optimization considerably extends the time complexity of the analysis. As a result, the runtime of entire decompilation averagely grows by 4.3 %. Figure 8.1 shows the most performance influential operations in the new FPU optimization. The graph summarizes the result of optimization profiling. Profiler[3] measures decompilation executed over one of the biggest binary programs in the database. The program contains a function with more than 3 000 basic block that manipulates FPU stack. The graph shows that most of the time, the optimization evaluates QR decomposition to determine system solution (see Section 6.2).

As a result, the implementation was modified. Due to decompilation performance requirement, the newly proposed extension does not optimize the huge binaries. If optimization detects that the analysed function generates a system with more than 1 000 equations, then the new optimizations skip the search for a system solution. In such a case, optimization decreases to an old optimization algorithm that ignores CFG. After this new approach, the night tests results show that average FPU optimization runtime decrease by 92.9 % in comparison to optimization without performance restriction (see Figure in Appendix B.2).

To summarize, the final implementation of FPU optimization takes averagely 19 minutes while old optimizations around 3 minutes. The results of both night tests run shows Appendix B.

8.3 Regression tests

RetDec project also includes a regression test framework[4]. Generally, regression testing tries to detect errors introduced in the new version of the software (see [24]). Similarly, RetDec regression test framework checks each new version of the decompiler. Each phase of decompilation contains a specific subset of the regression tests. Typically, every proposed and implemented feature in some decompiler component must pass a specific regression test suite intended for this component. The night tests contain significantly bigger test set than regression tests, but regression framework also tests the entire decompilation process.

[3]Performance analysis tool perf: http://man7.org/linux/man-pages/man1/perf.1.html

[4]RetDec regression tests framework: https://retdec-regression-tests-framework.readthedocs.io/en/latest/

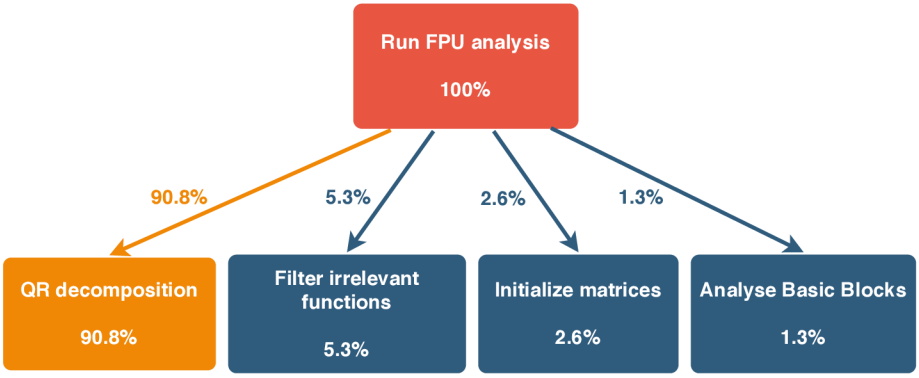

Figure 8.1: Profiling summary of large program decompilation (program includes function with more than 3 000 basic blocks).

Further, the regression test framework is freely available and open-source, on the contrary to the proprietary night test tool. Regression test utility allows to create tests that check for errors in the decompilation process by some of the following criteria:

- Evaluation of the expected return value in comparison to decompilation result. However, the framework allows for a finer diagnosis of the return value. It separately distinguishes the return value for each phase of the decompilation.
- The framework tries to compare the original program with the decompiled in terms of functionality equality. The framework compiles reversed C source and run both, the original and recompiled program. If programs produce the same outputs, then it declares the programs equality. This approach can be applied typically only for simple programs because decompiled programs often contain pseudo-code that is understandable for humans but not for compilers.
- Regression tool can search for key string patterns in decompilation output. It searches for definitions and usage of identificators, data types, functions and their parameters, variables and similar structures.

Entire regression test suite contains 5 464 samples, where 822 samples are binaries for x86 architecture with FPU instruction set usage. The framework tests new optimization extension with this test suite. All binaries decompilation run successfully and without error detections.

Chapter 9

Summary

The thesis outlines the topic of decompilation in term of software reverse engineering. It describes the typical decompiler structure and corresponding terminology. Next, the open-source decompiler project called RetDec was introduced. The specific technologies used in the decompiler were detailed.

Following chapters describe the x86 processor architecture supported by RetDec. The CPU extensions FPU and SSE were detailed. The research analyses RetDec decompilation deficiencies, and obstacles in connection with previously explained x86 extensions. As a result, it proposes the new extension for the reconstruction of FPU stack. Also, it suggests the extension of decompiler support for the floating-point instructions related to FPU and SSE.

The support of FPU instruction set in the decoder library was developed and integrated into the stable version of the decompiler. The decoder now provides semantic translation routines for 100 % of FPU instructions. Additionally, the partial support of core packed and scalar SSE instructions were implemented. However, the design is not properly tested.

The new FPU optimization extension was implemented and tested. The various experiments determine the best method for the proposed FPU optimization implementation in the matter of decompilation performance. The three testing framework tests the new extensions: unit, regression, and night test framework. All three tests suites pass successfully. The night tests measure performance decrease because of the new analysis complexity. The new FPU optimization is around six times slower than original, but such a performance decrease was expected, and it is acceptable concerning analysis improvement.

Bibliography

[1] *Volume 2: Instruction Set Reference, A-Z*. Intel 64 and IA-32 Architectures Software Developer's Manual. Intel. 2016.

[2] Anh, Q. N.; et al.: *Capstone the Ultimate Disassembler*. [Online; visited 01.05.2020]. Retrieved from: http://www.capstone-engine.org/

[3] Anh, Q. N.; et al.: *Keystone the Ultimate Assembler*. [Online; visited 01.05.2020]. Retrieved from: http://www.keystone-engine.org/

[4] Anton, H.: *Elementary Linear Algebra: Applications Version*. Wiley Plus Products. Wiley. 2010. ISBN 9780470560020.

[5] de Boyne Pollard, J.: *The gen on function calling conventions.* [Online; visited 01.05.2020].
Retrieved from: https://jdebp.eu/FGA/function-calling-conventions.html

[6] Čížek, Pavel and Čížkova, Lenka: *Numerical Linear Algebra*. Jan 2004.

[7] Eager, M. J.: *Introduction to the DWARF Debugging Format*. 2012. [Online; visited 01.05.2020].
Retrieved from:
http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

[8] Eilam, E.: *Reversing secrets of reverse engineering*. Indianapolis: Wiley. 2005. ISBN 0-7645-7481-7.

[9] Fog, A.: *Calling conventions for different C++ compilers and operating systems.* Technical University of Denmark. 2019. [Online; visited 01.05.2020].
Retrieved from: https://www.agner.org/optimize/calling_conventions.pdf

[10] Girkar, M.; Lu, H.; Kreitzer, D.; et al.: *System V Application Binary Interface AMD64 Architecture Processor Supplement Version 1.0*. 2018. [Online; visited 01.05.2020].
Retrieved from: https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI

[11] Guennebaud, G.; Jacob, B.; et al.: *Eigen v3*. http://eigen.tuxfamily.org. 2010.

[12] Hamill, P.: *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media. 2004. ISBN 9780596552817.

[13] Ientilucci, E. J.: *Using the Singular Value Decomposition*. Rochester Institute of Technology. 2003. [Online; visited 01.05.2020].
Retrieved from: http://www.cis.rit.edu/~ejipci/Reports/svd.pdf

[14] Irvine, K. R.: *Assembly language for x86 processors*. Upper Saddle River: Prentice Hall. 6 edition. 2010. ISBN 978-0-13-602212-1.

[15] Kapoor, V.; Yadav, R. K.: *A Hybrid Cryptography Technique for Improving Network Security*. 2016.

[16] Kubov, P.: *Zpětný překlad aplikací pro architekturu x86-64 v nástroji RetDec*. 2019. Retrieved from: https://www.fit.vut.cz/study/thesis/22058/

[17] Kusswurm, D.: *Modern x86 assembly language programming : 32-bit, 64-bit, SSE, and AVX*. The expert's voice in programming. New York]: Apress. 2014. ISBN 978-1-4842-0065-0.

[18] Křoustek, J.; Matula, P.: *RetDec: An Open-Source Machine-Code Decompiler*. [talk]. July 2018. Presented at Pass the SALT 2018, Lille, FR.

[19] Křoustek, J.; Matula, P.; Zemek, P.: *RetDec: An Open-Source Machine-Code Decompiler*. [talk]. December 2017. Presented at Botconf 2017, Montpellier, FR.

[20] Lattner, C.; Adve, V.: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California. Mar 2004.

[21] Lee, D. Q.: *Numerically efficient methods for solving least squares problems*. The University of Chicago. 2012. [Online; visited 01.05.2020]. Retrieved from: https://math.uchicago.edu/~may/REU2012/REUPapers/Lee.pdf

[22] Lopes, B. C.; Auler, R.: *Getting Started with LLVM Core Libraries*. Packt Publishing. 2014. ISBN 1782166920, 9781782166924.

[23] Matula, P.: *Capstone2LLVMIR*. [Online; visited 01.05.2020]. Retrieved from: https://github.com/avast/retdec/wiki/Capstone2LlvmIr

[24] Myers, G. J.: *The art of software testing*. Hoboken, New Jersey: Wiley. third edition. 2012. ISBN 978-1-118-03196-4.

[25] Robertson, C.: *x64 calling convention*. Microsoft Docs. 2018. [Online; visited 01.05.2020]. Retrieved from: https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=vs-2019

[26] Robertson, C.: *Compiler intrinsics*. Microsoft Docs. 2019. [Online; visited 01.05.2020]. Retrieved from: https://docs.microsoft.com/en-us/cpp/intrinsics/compiler-intrinsics?redirectedfrom=MSDN&view=vs-2019

[27] Singler, J.; et al.: *Static Single Assignment Book*. [Online; visited 01.05.2020]. Retrieved from: http://ssabook.gforge.inria.fr/latest/book-full.pdf

[28] Stallman, R. M.: *Using the GNU Compiler Collection*. 2019. [Online; visited 01.05.2020]. Retrieved from: https://gcc.gnu.org/onlinedocs/gcc-9.3.0/gcc.pdf

[29] Team, T. C.: *Clang 11 documentation*. 2020. [Online; visited 01.05.2020]. Retrieved from: https://clang.llvm.org/docs/LanguageExtensions.html#builtin-assume

[30] *Unix and UNIX System Laboratories*: *System V Application Binary Interface Intel 386 Architecture Processor Supplement*. Prentice Hall PTR. 1994. ISBN 9780131046702.

Appendix A

FPU optimization unit testing

The single unit test of the FPU optimization performs the following steps:

1. The test parses an input string that contains the LLVM IR module. The module always holds definitions of the FPU registers and variable function definitions (see Listing A.1). The content of the functions is the subject of the test. For example, Listing A.3 shows possible function content. Note that the lines 6, 11, 16, and 24 (in Listing A.3) carry the FPU pseudo load/store functions. The lines 7, 12, 17, and 25 show the expected substitution.

```
1  @fpu_stat_TOP = internal global i3 0
2  @st0 = internal global x86_fp80 0xK00000000000000000000
3  @st1 = internal global x86_fp80 0xK00000000000000000000
4  @st2 = internal global x86_fp80 0xK00000000000000000000
5  @st3 = internal global x86_fp80 0xK00000000000000000000
6  @st4 = internal global x86_fp80 0xK00000000000000000000
7  @st5 = internal global x86_fp80 0xK00000000000000000000
8  @st6 = internal global x86_fp80 0xK00000000000000000000
9  @st7 = internal global x86_fp80 0xK00000000000000000000
10
11 declare void @__frontend_reg_store.fpr(i3, x86_fp80)
12 declare x86_fp80 @__frontend_reg_load.fpr(i3
```

Listing A.1: Example of the specific architecture-dependent global variables.

2. Define the module environment configuration. The RetDec preprocessing generates a file with JSON metadata that defines processor architecture and their bit width, data endianity, name and calling convention for each function and other information. The unit test holds his JSON metadata. To illustrate, see Listing A.2.

```
1  {
2    "architecture" : {
3      "bitSize" : 32,
4      "endian" : "little",
5      "name" : "x86"
6    },
7    "mainAddress" : "0x1000",
8    "functions" : [
9      {
10       "callingConvention" : "cdecl",
11       "name" : "foo"
12     },
13     {
```

```
14          "callingConvention" : "cdecl",
15          "name" : "boo"
16        }
17      ]
18    }
```

Listing A.2: Example of the specific architecture-dependent global variables.

3. Unit test associates the predefined pseudo functions with RetDec core metadata because the core must detect these functions as special-purpose pseudo calls.

4. The test defines the application binary interface. In our case, it associates the FPU registers to the appropriate global variables:

   (a) General-purpose data registers `ST0` - `ST7`.

   (b) Tag registers `TAG0` - `TAG7`.

   (c) Pseudo status register, which indicates the FPU stack top value.

5. The test executes the FPU optimization in the prepared LLVM IR module.

6. Next, it evaluates the return value of executed module with the expected return value. The optimization modifies the LLVM IR module. Compares the modified module with the prearranged module. Such a module hold the same functions, but the pseudo load/store functions are already replaced with the correct registers load/store.

```
1  define void @foo() {
2    A:
3      br i1 1, label %B, label %C
4    B:
5      %0 = load i3, i3* @fpu_stat_TOP
6      call void @__frontend_reg_store.fpr(i3 %0, x86_fp80 0xK3FFF8000000000000000)
7      ;store x86_fp80 0xK3FFF8000000000000000, x86_fp80* @st0
8      %1 = sub i3 %0, 1
9      store i3 %1, i3* @fpu_stat_TOP
10     %2 = load i3, i3* @fpu_stat_TOP
11     call void @__frontend_reg_store.fpr(i3 %2, x86_fp80 0xK3FFF8000000000000000)
12     ;store x86_fp80 0xK3FFF8000000000000000, x86_fp80* @st7
13     br i1 1, label %D, label %E
14   D:
15     %3 = load i3, i3* @fpu_stat_TOP
16     %4 = call x86_fp80 @__frontend_reg_load.fpr(i3 %3)
17     ;%4 = load x86_fp80, x86_fp80* @st7
18     br label %E
19   E:
20     %5 = load i3, i3* @fpu_stat_TOP
21     %6 = add i3 %5, 1
22     store i3 %6, i3* @fpu_stat_TOP
23     %7 = load i3, i3* @fpu_stat_TOP
24     %8 = call x86_fp80 @__frontend_reg_load.fpr(i3 %7)
25     ;%8 = load x86_fp80, x86_fp80* @st0
26     br label %C
27   C:
28     ret void
29 }
```

Listing A.3: Example of the specific architecture-dependent global variables.

Appendix B

FPU optimization night tests

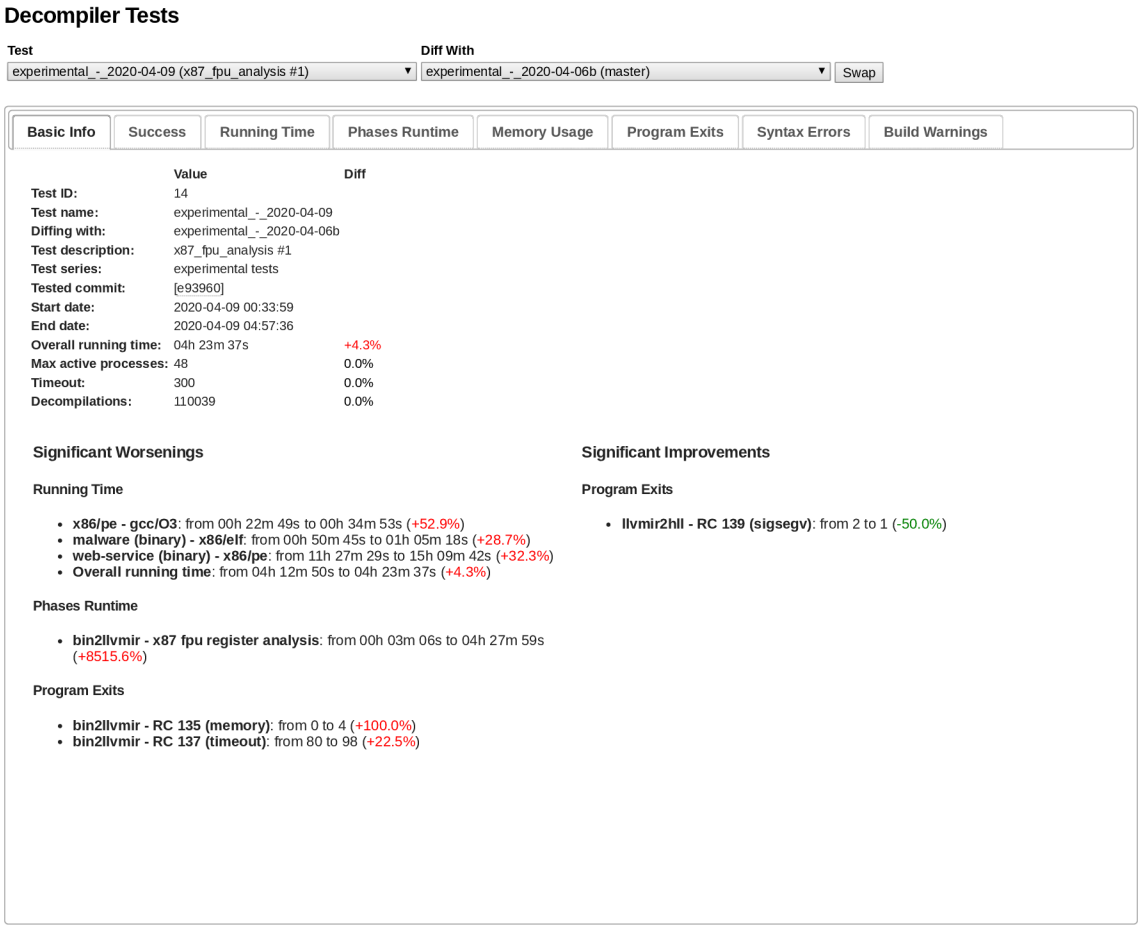

Figure B.1: Results of night tests without FPU optimization performance restriction.

The significant improvements of second night tests run (see Figure B.2) in other phases then *x87 FPU register analysis* are not caused by new FPU optimization. The improvements are the product of the latest developments in the project released since the previous measured night test run.

Decompiler Tests

Test experimental_-_2020-05-04 (x87-fpu-analysis #2) ▾ **Diff With** experimental_-_2020-04-09 (x87_fpu_analysis #1) ▾ Swap

Basic Info | Success | Running Time | Phases Runtime | Memory Usage | Program Exits | Syntax Errors | Build Warnings

	Value	Diff
Test ID:	19	
Test name:	experimental_-_2020-05-04	
Diffing with:	experimental_-_2020-04-09	
Test description:	x87-fpu-analysis #2	
Test series:	experimental tests	
Tested commit:	[e93960]	
Start date:	2020-05-04 21:19:29	
End date:	2020-05-05 01:20:42	
Overall running time:	04h 01m 13s	-8.5%
Max active processes:	48	0.0%
Timeout:	300	0.0%
Decompilations:	110051	+0.1%

Significant Worsenings

Phases Runtime

- **bin2llvmir - LLVM instruction optimization using RDA**: from 00h 00m 00s to 02h 01m 49s (+100.0%)

Memory Usage

- **web-service (binary) - arm64/macho**: from 650 MB to 826 MB (+27.1%)

Program Exits

- **llvmir2hll - RC 134 (abort)**: from 53 to 65 (+22.6%)

Significant Improvements

Success

- **mips/ihex - gcc/O0 - C syntax result**: from 13.8% to 91.4% (+562.2%)
- **mips/ihex - gcc/O1 - C syntax result**: from 13.8% to 91.1% (+560.0%)
- **mips/ihex - gcc/O2 - C syntax result**: from 12.9% to 89.6% (+595.2%)
- **mips/ihex - gcc/O3 - C syntax result**: from 12.9% to 89.0% (+590.5%)

Running Time

- **mips/ihex - gcc/O0**: from 00h 35m 45s to 00h 24m 56s (-30.3%)
- **mips/ihex - gcc/O1**: from 00h 30m 57s to 00h 20m 25s (-34.0%)
- **x86/pe - gcc/O3**: from 00h 34m 53s to 00h 22m 10s (-36.5%)
- **malware (binary) - x86/elf**: from 01h 05m 18s to 00h 39m 48s (-39.1%)
- **web-service (binary) - arm/pe**: from 00h 34m 36s to 00h 24m 24s (-29.5%)
- **web-service (binary) - x86/pe**: from 15h 09m 42s to 10h 15m 00s (-32.4%)
- **Overall running time**: from 04h 23m 37s to 04h 01m 13s (-8.5%)

Phases Runtime

- **bin2llvmir - Input binary to LLVM IR decoding**: from 12h 58m 40s to 09h 28m 46s (-27.0%)
- **bin2llvmir - Simple types recovery optimization**: from 01h 29m 24s to 00h 35m 05s (-60.7%)
- **bin2llvmir - Assembly mapping instruction removal**: from 00h 36m 27s to 00h 20m 52s (-42.8%)
- **bin2llvmir - x87 fpu register analysis**: from 04h 27m 59s to 00h 19m 00s (-92.9%)

Program Exits

- **Unpacking - RC 4 (unpacker failed, other not succeeded)**: from 19 to 5 (-73.7%)
- **bin2llvmir - RC 134 (abort)**: from 19 to 17 (-10.5%)
- **bin2llvmir - RC 137 (timeout)**: from 98 to 67 (-31.6%)
- **bin2llvmir - RC 139 (sigsegv)**: from 13 to 5 (-61.5%)
- **C Syntax - RC 1 (syntax error)**: from 17438 to 14431 (-17.2%)

Syntax Errors

- **Generated C**: from 129103 to 101353 (-21.5%)

Figure B.2: Results of night tests with FPU optimization performance restriction.