VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

REKONFIGUROVATELNÁ ANALÝZA STROJOVÉHO KÓDU

RETARGETABLE ANALYSIS OF MACHINE CODE

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. JAKUB KŘOUSTEK
AUTHOR

VEDOUCÍ PRÁCE Doc. Dr. Ing. DUŠAN KOLÁŘ
SUPERVISOR

BRNO 2014

Abstrakt

Analýza softwaru je metodologie, jejímž účelem je analyzovat chování daného programu. Jednotlivé metody této analýzy je možné využít i v dalších oborech, jako je zpětné inženýrství, migrace kódu apod. V této práci se zaměříme na analýzu strojového kódu, na zjištění nedostatků existujících metod a na návrh metod nových, které umožní rychlou a přesnou rekonfigurovatelnou analýzu kódu (tj. budou nezávislé na konkrétní cílové platformě). Zkoumány budou dva typy analýz – *dynamická* (tj. analýza za běhu aplikace) a *statická* (tj. analýza aplikace bez jejího spuštění). Přínos této práce v rámci dynamické analýzy je realizován jako *rekonfigurovatelný ladicí nástroj* a dále jako dva typy tzv. *rekonfigurovatelného translátovaného simulátoru*. Přínos v rámci statické analýzy spočívá v navržení a implementování *rekonfigurovatelného zpětného překladače*, který slouží pro transformaci strojového kódu zpět do vysokoúrovňové reprezentace. Všechny tyto nástroje jsou založeny na nových metodách navržených autorem této práce. Na základě experimentálních výsledků a ohlasů od uživatelů je možné usuzovat, že tyto nástroje jsou plně srovnatelné s existujícími (komerčními) nástroji a nezřídka dosahují i lepších výsledků.

Abstract

Program analysis is a computer-science methodology whose task is to analyse the behavior of a given program. The methods of program analysis can also be used in other methodologies such as reverse engineering, re-engineering, code migration, etc. In this thesis, we focus on program analysis of a machine-code and we address the limitations of a nowadays approaches by proposing novel methods of a fast and accurate retargetable analysis (i.e. they are designed to be independent of a particular target platform). We focus on two types of analysis — *dynamic analysis* (i.e. run-time analysis) and *static analysis* (i.e. analysing application without its execution). The contribution of this thesis within the dynamic analysis lays in the extension and enhancement of existing methods and their implementation as a *retargetable debugger* and two types of a *retargetable translated simulator*. Within the static analysis, we present a concept and implementation of a *retargetable decompiler* that performs a program transformation from a machine code into a human-readable form of representation. All of these tools are based on several novel methods defined by the author. According to our experimental results and users feed-back, all of the proposed tools are at least fully competitive to existing solutions, while outperforming these solutions in several ways.

Klíčová slova

strojový kód, analýza kódu, reverzní inženýrství, zpětný překladač, ladicí nástroj, simulátor, zpětný assembler, gramatiky s rozptýleným kontextem, Lissom, jazyky pro popis architektur, ISAC, škodlivý kód.

Keywords

machine code, code analysis, reverse engineering, decompiler, debugger, simulator, disassembler, scattered context grammars, Lissom, architecture description languages, ISAC, malware.

Citation

Jakub Křoustek: Retargetable Analysis of Machine Code, PhD thesis, Brno, FIT BUT, 2014

Contents

Chapter 1

Introduction

Program analysis is a computer-science methodology, whose task is to analyse the behavior of a given program [24]. By using various methods of program analysis, it is not only possible to understand program behavior, but it can also help with detection and removal of program flaws (i.e. *debugging*[1]), optimization of program execution, program verification, etc. These methods are also used in other methodologies such as reverse engineering, re-engineering, program comprehension, code migration, and many others.

The analysed program can be stored in one of the following forms[2] — *source code*, *bytecode*, or *machine code*. In compiler terminology, the source-code form is the original program form written in a particular programming language. The bytecode and machine-code forms are binary representations of the source code created by a compiler. Code representation in the last two formats is not textual (unlike a source code), but it is in a binary form containing machine instructions. Bytecode is usually interpreted in a particular virtual machine (VM) while machine code is executed on a target processor architecture.

The vast majority of the existing methods of program analysis is related to the source-code form. Roughly speaking, analysis of the source code is easier than the others because it is more or less platform independent and all the information necessary for analysis is available within the source code. The source-code representation contains complete behavioral description of the program including details like variable names or user comments. These methods of source-code analysis are well described in literature, e.g. formal verification, program slicing, or abstract interpretation. They are used in all aspects of software engineering like static error checking, unit testing, debugging, etc.

The bytecode format is employed only in a fraction of architectures, almost always in virtual machines, e.g. Java VM or Android Dalvik VM. For this reason, methods of bytecode analysis are not so common. The important fact about bytecode program representation is that there is only a minor information loss during compilation from source-code into the bytecode [25,26], e.g. removal of comments and code formatting. Therefore, it is often possible to apply the same program-analysis methods as in the source-code case. Furthermore, the decompilation from bytecode back into the original language is also plausible with only a small effort, see [25] for details.

The term *decompilation* refers to a reverse process to compilation, i.e. translation from an executable-code representation back into the source code. With a few exceptions, we will

[1]The term *debugging* denotes "methodology of finding, isolating, and removing defects (bugs) from software programs", according to [32].

[2]There are also other forms of code representation (e.g. object code); however, those are rarely used for program analysis.

only use this term for translation from the executable machine-code files into a high-level language (HLL) form in this thesis.

Finally, the machine-code program representation is used in all types of electronic devices around us (personal computers, smart phones, automation, car industry, etc.). Sometimes, it is even used for program acceleration in virtual machines, e.g. Java Native Interface (JNI) in Dalvik VM.

Machine code is almost always created by compilers that use several levels of optimizations. The major difference to the previous types of code representation is that the machine-code representation lacks a significant amount of information. Compilers transform the original program to a different form in order to produce a fast and compact code for a particular target architecture. There are dozens of existing optimization techniques used for this task, e.g. dead-code elimination, tail-call optimization, usage of instruction idioms [1,41]. Furthermore, compilers usually remove any information that is not necessary for execution (e.g. information about original functions that were inlined, information about user-defined data types). Therefore, program analysis of these files is more difficult because it is necessary to reconstruct all the missing information and to deal with a highly-optimized code that is dissimilar to its original form.

At present, we can find several existing techniques and tools for analysis of machine code, like disassemblers, dumpers, or debuggers [3]. They differ in a level of presented information about the analysed program. For example, a disassembler only transforms the machine code into assembly code that is still hardly readable for most analysts. Moreover, most of these tools are limited to a particular target architecture (mostly Intel x86) and they do not support the others. In other words, they are not *retargetable*. This is unfortunate because some architectures are not covered by these tools and program analysis has to be done manually by user.

In this thesis, we address the limitations of present day methods of machine-code analysis and we propose several new methods that are focused on a fast and accurate retargetable analysis. These methods are designed to be independent of a particular target platform, i.e. a combination of a target architecture, file format, and used compiler. We focus on two types of analysis — *dynamic analysis* and *static analysis*.

Dynamic analysis executes the application and analyses its behavior during its run-time. The execution can be done either directly on the target architecture (e.g. remote debugging) or via simulation or emulation. Profilers, simulators, and debuggers are the most common tools within this category.

Static machine-code analysis examines applications without their execution and it is usually focused on program transformation from machine code into a human-readable form of representation. The most common tools are disassemblers that produce assembly code and the more modern decompilers producing an HLL code (C, Java, etc.). Static analysis can also perform control-flow and data-flow checking, see [10] for more details.

Within the dynamic analysis, the contribution of this thesis lays in the extension and enhancement of existing methods and their implementation as two types of a *retargetable translated simulator* and a *retargetable debugger*. In order to enhance existing static-analysis methods, we implemented a *retargetable decompiler*. This tool is based on several novel methods defined by the author.

As an application of these tools, we should mention a high-speed simulation (or emulation/instrumentation) of executable files for new ASIP chips, debugging of a highly-optimized code, or deep analysis of malicious code (malware). We successfully validated our approaches via implementation of proposed methods and their testing on real-world examples within the Lissom[3] project at Brno University of Technology.

1.1 Chapter Survey

The thesis is organized as follows. Firstly, Chapter 2 gives preliminaries and basic definitions, including used notation. These definitions are fundamental for methods presented in Chapters 5 and 6.

The focus of this thesis is specified in Chapter 3. Within this chapter, we briefly describe architectures, file formats, and other standards that we focus on within our methods of machine-code analysis. Afterwards, Chapter 4 discusses the state of the art of machine-code analysis methods, such as related approaches, projects, and tools. Then, in Chapter 5, we present our own methods of dynamic machine-code analysis and their implementation — two different types of translated simulator and the source-level debugger. Afterwards, in Chapter 6, the retargetable decompiler is presented in detail. Experimental results related to these tools are given in Chapter 7. In the conclusion of this work, given in Chapter 8, obtained results are summarized, several final remarks are made, and possible further research is discussed. It also states several open problems closely related to this work.

[3] http://www.fit.vutbr.cz/research/groups/lissom/

Chapter 2

Preliminaries

In this chapter, we present the terminology, notation, and fundamental definitions of the formal language theory and compiler design used in this thesis. The reader is assumed to have basic knowledge of these topics.

Section 2.1 is related to the formal language theory. We pay special attention to scattered context grammars that are often referenced in the latter text. More details about this formalism can be found in [23]. Afterwards, in Section 2.2, we review the terminology used within the area of compiler design, such as basic blocks, control-flow graph, etc. See [1] for further reference.

2.1 Theory of Formal Languages

Definition 1. For an alphabet V, V^* denotes the free monoid generated by V under the operation of concatenation, with the unit element ε. Set $V^+ = V^* - \{\varepsilon\}$. For $w \in V^*$, $|w|$ denotes the length of w.

Definition 2. A *phrase-structure grammar* is a quadruple $G = (V, T, P, S)$, where

- V is a *total alphabet*;
- $T \subset V$ is a finite set of *terminal symbols* (*terminals*);
- $S \in V - T$ is the *start symbol* of G;
- P is a finite *set of productions* (*rules*) $p = x \to y$, $x \in V^*(V - T)V^*$, $y \in V^*$.

The symbols in $V - T$ are referred to as *nonterminal symbols* (*nonterminals*).

Definition 3. A *context-free grammar* (CF grammar) is a phrase-structure grammar $G = (V, T, P, S)$, such that every production $p = x \to y \in P$ satisfies $A \to x$, where $A \in V - T$ and $x \in V^*$.

Definition 4. A *scattered context grammar* (SCG, see [23]) is a quadruple $G = (V, T, P, S)$, where

- V is a total alphabet;
- $T \subset V$ is a finite set of terminals;
- $S \in V - T$ is the start symbol;

- P is a finite set of productions of the form $(A_1, \ldots, A_n) \to (x_1, \ldots, x_n)$, where $A_i \in V - T$, $x_i \in V^*$ for all $i : 1 \leq i \leq n$.

Definition 5. A *propagating scattered context grammar* (PSCG) is a SCG $G = (V, T, P, S)$, in which every $(A_1, \ldots, A_n) \to (x_1, \ldots, x_n) \in P$ satisfies $x_i \in V^+$ for all $i : 1 \leq i \leq n$.

2.2 Theory of Compiler Construction

Within this section, we define the basic terminology used within the compiler construction theory. This section is based on [1].

Definition 6. *Basic block* (BB) is a sequence of statements[4] that are always executed together and it has a single entry point (*start address*), single exit point (*end address*), and no branch statement within it. In other words, only the last statement of a particular basic block may branch[5] to other basic block and only the first statement of a basic block can be a destination of any branch statement.

Definition 7. *Control-flow graph* (CFG) for a program P is a directed graph $G = (N, E)$ that satisfies the following conditions:

1. Each node $n \in N$ represents a basic block of P;
2. Each edge $(u, v) \in E$ represents a branch from one basic block (u) to another (v);
3. N contains two distinguished nodes: the *initial node* v_I, which has indegree zero, and the *exit node* v_E, which has outdegree zero and through which the control flow leaves (e.g. return from application to operating system).

Definition 8. *Call graph* (CG) for a program P is a directed graph $G = (N, E)$ that satisfies the following conditions:

1. Each node $n \in N$ represents a *function*[6] of P;
2. Each edge $(u, v) \in E$ represents a function call from function u (*caller*) to function v (*callee*);
3. The *initial node* $v_I \in N$ (indegree zero) represents an entry function of the program P, denoted as `main` function in the following text[7].

Often, a call graph is *fully connected*, i.e. the entry point (EP) of the program reaches all procedures.

[4]The term *statement* is used in the meaning of HLL statement (e.g. assignment, loop, conditional statement). Moreover, all the following definitions in this sections hold for machine-code instructions too.

[5]In the following text, the term *branch* stands for every statement that can change the flow of execution (e.g. (un)conditional branch, function call, return from function, interruption).

[6]With a few exceptions, the term *function* represents also other similar HLL constructions (such as *procedures*) in the following text.

[7]This definition stands for executable files with one entry point. However, this is not true for libraries, because they have multiple entry points, and the call graph may not be fully connected.

Chapter 3

Focus of the Thesis

This thesis is focused on the retargetable analysis of executable files. The retargetability of analysis means that it has to handle different target processor architectures, executable-file formats, operating systems, compilers, etc. More precisely, we are focused on analysis of binary executable files containing machine code. This means that we are not focused on analysis of byte-code files or virtual machines.

In the following chapters, we often reference existing architectures and standards. For this reason, we briefly describe the most common architectures, executable-file formats (EFF), and debugging-information standards.

CPU Architectures Overview

At present, almost every electronic device is controlled by one or multiple microprocessors. We can find thousands of different types because each of them suits a different area (signal processing, general-purpose computing, etc.). In order to characterise the existing type, we use classification based on the type of instruction set – *CISC*, *RISC*, and *VLIW*.

Executable-File Formats Overview

Machine code of an application is usually not stored as-is, but it is wrapped in a platform-specific *executable-file format*. The term *executable-file format* (EFF) refers to a format of an executable code, library code, or not yet linked object code. In the following text, we focus mainly on the binary executable code. A generic EFF usually consists of the parts such as *header*, *sections*, or symbols. Unfortunately, there is no such generic format and each platform has its own format, or a derivative of an existing one. At present, we can find two major EFFs — UNIX ELF and WinPE.

Debugging-Information Formats Overview

Program comprehension and analysis can be much more effective whenever there another source of information is available about the analysed application — debugging information. This information is commonly used for support of fast and accurate debugging, tracing, profiling, etc. Within this information, we can for example find information about mapping of machine instructions to source-code lines and functions, positions of variables and other useful details. Two of the present day most common formats are DWARF and PDB.

Chapter 4

Machine-Code Analysis: State of the Art

In this chapter, we briefly describe the existing approaches and tools of machine-code analysis. Each of them has a different complexity and scenario of usage.

Firstly, we describe existing tools and techniques for static analysis of a machine code (e.g. disassemblers, decompilers). These tools are mostly platform dependent with only a few exceptions that try to be retargetable (e.g. the Boomerang decompiler). Afterwards, we give a brief overview of existing projects focused on dynamic analysis (e.g. debuggers, simulators). Finally, we describe the Lissom research project because the novel methods presented in Chapters 5 and 6 are closely tied to this project.

4.1 Compiler and Packer Detectors

During the preliminary analysis of a particular executable file, it is important to gather basic information that can be used within any further analysis. For example, we can try to detect the target architecture, EFF, originally used language, and a particular compiler. Moreover, applications can also be packed or protected by so-called *packers* or *protectors*. Note: in the following text, we use the term *packing* for all the techniques of executable-file creation, such as packing, compression, protection, etc. PEiD, Exeinfo PE, and DiE are the most common tools used for analysis of EFF.

4.2 EFF Parsers and Converters

We can find several projects focused on parsing and binary conversion of EFFs. They are used mostly in reverse engineering or for a code migration between two particular platforms. Basically, every binary reverse-engineering tool must have some kind of EFF parser or converter into its internal format in order to extract program's machine code and obtain all necessary information (e.g. EP address, symbols, debugging information). As an example, we can mention Binary File Descriptor library (BFD) or the BFF grammar (Binary File Format Grammar).

4.3 Disassemblers

Disassembler is a static-analysis tool that translates machine-code instructions back into assembly language. These tools usually support only one particular EFF and target architecture. The conversion is done statically, without execution of the input file. These tools are used for a low-level analysis of machine code (checking correctness of code generated by assembler, linker or compiler, reverse engineering of a third-party software, etc.).

We can find dozens of existing disassemblers because their development is not so complex. At present, the most common disassemblers are `objdump`, IDA Disassembler, and ODA.

4.4 Machine-Code Decompilers

One of the most challenging tasks of reverse engineering is the machine-code decompilation, i.e. translation of binary executable files into an HLL representation. It can be used for source-code reconstruction, binary-code migration, malware analysis, etc. The most important modern decompilers are mentioned in the following text.

Boomerang[8] is an open source project established in 2002. Boomerang supports decompilation of applications for Intel x86, SPARC, and PowerPC architectures and UNIX ELF, WinPE, and Apple Mach-O EFFs. The Boomerang decompiler is probably the first attempt to create a retargetable decompiler by using the ADL description of the target architecture. According to the official site, the development was discontinued in 2006.

REC Studio—Reverse Engineering Compiler[9] is a freeware, but not open-source, interactive decompiler, which is still under development. It supports Windows, Linux and Mac OS X executables, and attempts to produce a C-like representation of the code and data used to build the executable. It supports 32-bit and 64-bit executables.

The *Hex-Rays Decompiler*[10] is a recent decompilation "standard". It is implemented as a plugin to the aforementioned IDA disassembler. The Hex-Rays Decompiler supports the x86, x64, and ARM target architectures. It also supports both major EFFs—UNIX ELF and WinPE. The output is generated as a highly readable C code. Furthermore, this software is commercial and distributed without sources.

The original name of this decompiler is *TyDec*. Later, it was renamed to *SmartDec*[11]. The work on the decompiler was based on research related to automatic type reconstruction in disassembled programs [4]. This decompiler is focused on decompilation of executables produced by C++ compilers.

4.5 Debuggers

In comparison with the previously mentioned methods, debugging is a dynamic analysis, i.e. the debugged application (also called *debuggee*) is executed during its analysis. Debugging is a very important part of application testing because it can be used for analysis of a program's run-time behavior as well as for detection of its flaws.

A basic set of application state information includes the source code position, stack backtrace (i.e. a sequence of function calls and their arguments), processor state (e.g. a

[8] http://boomerang.sourceforge.net/
[9] http://www.backerstreet.com/rec/rec.htm
[10] www.hex-rays.com/products/decompiler/
[11] http://decompilation.info/

content of memory, inspection of registers and pipeline), and an inspection of application variables. The user of debugger also needs to control the application execution. This can be done by several types of breakpoints, and by the modification of the system environment (e.g. processor registers, variables).

At present, the most common debuggers are GNU Debugger `gdb`, Microsoft Visual Studio Debugger, WinDbg, and OllyDbg.

4.6 Tools for Simulation and Emulation

Application debugging is not the only type of dynamic analysis. We can find other approaches like simulation, emulation, or instrumentation. These also analyse an application during its run-time, like debuggers, but they are not primarily focused on isolation of bugs or run-time interaction with users.

The main purpose of simulators and emulators is to create an application's run-time environment of one particular system (the guest) running in another system (the host). However, both approaches differ in how to achieve this task, see [27,32]. Roughly speaking, *simulation* is a model-based approach that behaves similarly to guest, but it is implemented in an entirely different way. On the other hand, *emulation* is focused on reproduction of the same exact external behavior like the emulated system. The most common tools are QEMU, Bochs, and Intel Pin.

4.7 Lissom Project

In this section, we describe the Lissom research project that creates a background for the newly designed methods proposed in this thesis. The Lissom project is located at FIT BUT, Czech Republic and it started in 2004. It is focused on two basic scopes. The first scope is a development of the ISAC ADL for the ASIP description. The second scope is a transformation of ASIP description into a complete toolchain and hardware realization.

During the start of a work on this thesis, the development of new ASIP chips was the primary target of the Lissom project. However, we realized that we can re-use this concept for description of existing architectures, such as Intel x86, ARM, and many others. Furthermore, the author of this thesis designed and developed additional tools that extend the original toolchain (e.g. decompiler, translated simulator, source-level debugger). These are described in Chapters 5 and 6. In the rest of this section, we describe the fundamentals of this project.

4.7.1 ISAC Language

The ISAC (Instruction Set Language C) language [21] belongs to the so-called *mixed ADL* that captures both structure and behavior of the architecture. It means that the processor instruction-set with processor micro-architecture is described in one model.

The processor model in ISAC consists of two basic parts. In the first part, processor resources, such as registers or memories, are described. The processor instruction set and processor micro-architecture can be described in the second part. One of the author's contributions is the extension of the ISAC language for modelling parallel architectures, with primary focus on the VLIW architecture [30,33].

4.7.2 Lissom Object File Format

Applications created by users of the Lissom toolchain (mentioned in the next subsection) are stored in an internal format. The Lissom project uses its own EFF that is based on the Common Object File Format (COFF). It was designed in reference to independence on any particular architecture, universality, and to be well readable. Therefore, it is possible to describe architectures with different types of endianity, byte sizes, instruction lengths, or instruction alignments. It is also possible to store executable, object, or library code within this format. The full specification can be found in [11].

4.7.3 Toolchain

During the toolchain generation, the user-described ISAC model is further processed by two tools – *toolchain generator* and *semantics extractor*. The toolchain generator produces tools like C compiler, debugger, simulator, etc. The semantics extractor is a necessary prerequisite for building the C/C++ compiler (and later the decompiler as well). The semantics extractor is used for extraction of semantics, assembler syntax, and binary encoding of each instruction from the instruction set, for details see [8]. It transforms description of the instruction into a sequence of LLVM IR-like instructions[12], which properly describe its behavior, see Figure 4.1.

```
instr instr_add__gpregs__gpregs__gpregs,
  ; semantics
  %tmp1 = i32 read_register(gpregs1)
  %tmp2 = i32 read_register(gpregs2)
  %i    = add(%tmp2, %tmp1)
  write_register(gpregs0) = %i,
  ; syntax
  "ADD" gpregs0 "," gpregs1 "," gpregs2,
  ; coding
  0b000000 gpregs1[4,0] gpregs2[4,0] gpregs0[4,0] 0b00000100000
```

Figure 4.1: Example of an extracted semantics for MIPS instruction `ADD`.

At the beginning of this thesis, applications compiled via the Lissom compiler could be simulated in two versions of simulator – *interpreted* and *compiled*. These two types differ in speed and area of usage. The compiled simulator improves performance of the interpreted simulation by removing constant fetching and decoding of instructions. Therefore, the compiled simulator decodes and analyses the instructions within the target application before the simulation starts.

Because the target platforms exist only as models in the early phases of development, there are no physical devices where the application can be remotely debugged. Therefore, the simulator is used for their execution and debugging. At the beginning of this thesis, the Lissom debugger supported two levels of debugging – instruction level and microarchitectural level. Source-level debugging was not supported in that time, e.g. it was possible to inspect value of register or memory cell, but inspecting the value of a variable was not supported.

[12] http://llvm.org/docs/LangRef.html

Chapter 5

Retargetable Dynamic Analysis

This chapter is based on [17,19,28,29] and it describes the author's contribution within the area of dynamic machine-code analysis.

As we mentioned in Section 4.5, the dynamic analysis of machine code is focused on analysis of a particular application during its run-time. This type of analysis can be used for debugging, profiling, detection of malicious behavior, etc.

In this chapter, we present several of our own methods and tools that are focused on this type of analysis and that are used in practice within the Lissom project. Within this project, they are used mainly for a processor and application design and testing and we will refer to this type of usage within this chapter. For example, these tools can be used for simulating and debugging an application's machine-code generated via the Lissom toolchain for a particular target architecture.

The structure of this chapter is as follows. Firstly, in Section 5.1, we describe the methods used for extraction and processing of debugging information from executable files. This information is essential for the proposed retargetable source-level debugger, but it is also used in all other tools presented in this thesis. Furthermore, the extracted debugging information is also used in other tools developed within the Lissom project, such as within the C profiler for obtaining information about source level functions [27].

Furthermore, in Section 5.2, we present the concept and implementation of a new simulator type — the translated simulator. More precisely, we present two versions of the translated simulator — static and just-in-time. In comparison with the existing types of simulators developed within the Lissom project, the newly designed methods provide higher simulation speeds as is demonstrated in Chapter 7.

Finally, in Section 5.3, we present an extension of the retargetable debugger. The contribution lays in the source-level debugging that is built on the top of the existing debugger. It is used for debugging of applications generated by the Lissom C compiler.

5.1 Obtaining and Processing Debugging Information

In Section 3, we have briefly mentioned two standards of debugging information — DWARF and PDB. Information stored in these formats is highly valuable for program analysis and we have to obtain all available information in order to produce the most accurate results. However, the representation in these formats is very low-level and its extraction and usage may be tricky. For example, the PDB format is proprietary and it has to be reverse-engineered, the DWARF processing libraries are mostly tied to the ELF format, etc. Therefore, we created our own libraries for parsing and processing these formats. For implementation details see [17].

5.1.1 DWARF Parsing

DWARF is independent of a particular EFF, programming language, operating system, or target architecture. The DWARF debugging information is stored in data sections together with the application's code in a single (executable) file. The name of each DWARF section begins with a fixed prefix "`.debug_`", but purpose of each section differs. The content of these debug sections is stored in a binary form defined in DWARF standard [5]. It contains several types of DWARF low-level description elements that have to be assembled together in order to obtain a high-level information, such as function name or type of variable.

Parsing of this DWARF information is non-trivial because the DWARF standard [5] is quite complex and creating a parser from scratch may be a time-consuming task. Nevertheless, the existing libraries and tools (e.g. `libdwarf`, `readelf`) for parsing this format support only the ELF format in most cases. Therefore, we created our own DWARF parsing library called `dwarfparserl` that is also capable to obtain DWARF information from the Lissom COFF and transform the low-level elements into a high-level representation allowing easy handling of debugging information and its usage in our tools.

Parsing of debugging information from an object file to structures defined in the DWARF specification is done by the `libdwarf` library[13]. As we noticed, the original `libdwarf` library is using the `libelf` library to access sections of interest in ELF binaries. However, the input of most of our tools is in the uniform COFF-based EFF. In order to parse information from such files, we exploited `libdwarf` object-access capabilities and provided a new interface by using our object manipulation library. The overview of this library is depicted in Figure 5.1.

However, `libdwarf` creates a low-level representation of debugging data, whose usage in a decompiler or debugger would be very complicated and unintuitive. That is why we decided to create a new, mid-layer library called `dwarfparserl`, that builds high-level, object-oriented data structures and provides convenient access methods. The `dwarfparserl` library represents all debugging information about a binary file as one entity consisting of several vectors of objects such as compilation units, lines, types, functions, and global variables. In this way, the information that was in a low-level representation is grouped together allowing a very natural and easy way of processing complex DWARF structures.

5.1.2 PDB Parsing

In comparison with DWARF, Microsoft PDB format is stored separately from the executable file. Theoretically, this should make its parsing and processing easier because it may be independent of a file format and the debugging information does not interfere with the code. However, none of these claims is true because PDB is used only with the WinPE format and its processing is much harder because it is a proprietary format.

A PDB file structure is similar to a file system because each PDB file consists of many streams (i.e. sub-files), each of them contains a different kind of information. There is a stream with type information (e.g. common types, enumeration, structures), compilation units (i.e. modules), symbol tables and PE sections.

The PDB file parsing consists of two steps. (1) Firstly, the streams have to be extracted and separated. They are divided into constant-size data blocks. (2) The main stream

[13]`http://www.prevanders.net/dwarf.html`

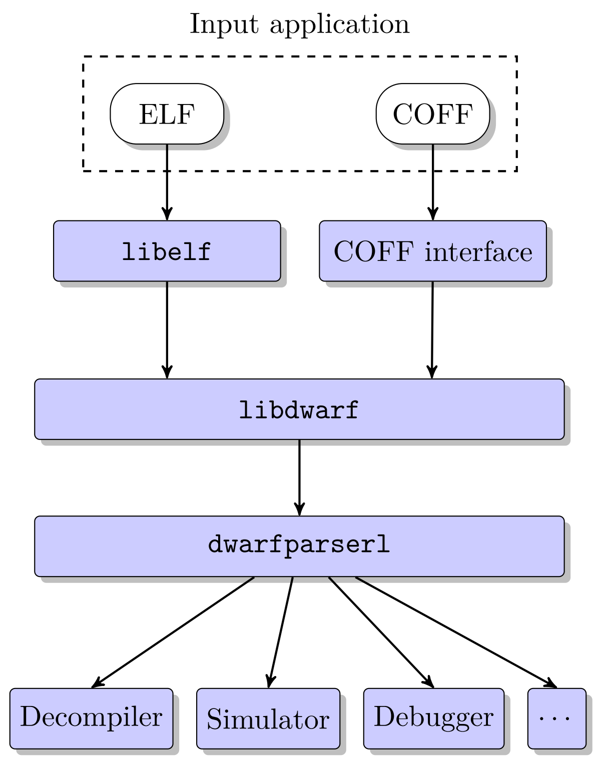


Figure 5.1: Structure of the `dwarfparserl` library.

processing is done in a consequent step. Most of the streams are organized into *symbols*, which are data structures with a type, size, and data. While processing all the symbols, the parser fills the internal data structures. By using the `pdbparserl`, we are able to obtain information similar to the DWARF format. At present, we are able to extract and utilize most of the PDB streams, but a few remaining streams are still unknown for us.

5.2 Translated Simulator

The fast and accurate processor simulator is one of the essential tools for effective design and testing of modern high-performance application-specific instruction set processors. At present, the trend of ASIP design is focused on automatic simulator generation based on a processor description in an ADL. The simulator is used for testing and validation of a designed processor or target application. Furthermore, the simulator creates the necessary simulation platform for a retargetable (source-level) debugger (described later in Section 5.3) because debugging on a target chip (e.g. JTAG debugging) is not possible until the chip design is finalized and the chip is synthesized.

We can find several types of automatically generated simulators that are useful in different design phases. The basic type of simulator is the *interpreted simulator* [31]. The concept of this simulator is based on a constant fetching, decoding, and execution of instructions from the memory. Therefore, the simulation itself is relatively slow.

If the developer wants to increase the speed of a simulation, he or she can use the second type of simulator, the *compiled simulator* [27]. It is created in two steps. In the first step the simulated application is analysed and the C code simulating the application's behavior is emitted based on the analysis. In the second step, the emitted C code is compiled together with the static parts of the simulator, such as processor resources, etc. It is clear that this version of a compiled simulator is dependent on the simulated application and the self-modifying code is not supported.

The second version of the compiled simulator is the *just-in-time compiled simulator*. It supports self-modifying code and it is not dependent on the simulated application. It is created in only one step and works in the following way. At the beginning of simulation, the simulator works as the interpreted simulator. The main task of this phase is to find the so-called *hot-spots* (i.e. parts of the simulated application in which the most of the simulation time is spent). Afterwards, these parts are compiled and the subsequent simulation of these parts will be faster.

5.2.1 Translated Simulation Overview

The speed of the compiled simulator can be further improved by the *translated simulation*, which enhances the concept of the compiled simulation. During the creation of the translated simulator, information about basic blocks (see Definition 6) in the target application is used for enabling a batch simulation of all instructions within a particular basic block. This batch execution can omit or optimize some checks and operations that are necessary within the compiled simulation and it leads to a better performance.

Since we need to pre-process information about the basic blocks, the translated simulator has to be pre-compiled; therefore, in its basic form (i.e. *static translated simulator*), it is dependent on a particular simulated application and SMC simulation is not supported as well as systems with external memories. Sometimes, this can be a serious limitation. Hence, we also provide an enriched version of this simulator (i.e. *just-in-time translated simulator*), which removes these limitations.

The following notation is used in this section. A *target C compiler* is the retargetable C compiler developed within the Lissom project [8]. A *target application* is the simulated application, which will later run on the designed processor. A *host C compiler* is the GCC compiler, which is used for compilation of the generated simulator.

5.2.2 Static Translated Simulation

The generation process of the static translated simulator has three parts. The first part is performed only once for any particular processor description[14]. The next two parts are target-application specific. Therefore, they have to be performed every time when the target application is changed.

(1) In the first part, the analyser of the target application is generated. It is generated only once and it is based on the processor description (i.e. it does not have to be re-generated until the processor description has been changed). This analyser is similar to the disassembler because it also accepts an application in a form of a machine code (in the Lissom COFF format). However, instead of emitting the assembly code, it emits a C-code that describes behavior of this program. This approach is similar to the approach used in the compiled simulators, see [27] for details.

The analyser itself is based on the following formalisms: *enhanced lazy finite automata* and *two-way coupled finite automata*. The two-way coupled finite automata C used in the analyser is a triple $C = (M_1, M_2, h)$, where $M_i = (Q_i, \Sigma_i, R_i, s_i, S_i, F_i, z_i)$ is a lazy finite

[14]The ISAC processor description has to be on the instruction-accurate level.

automaton for $i = 1, 2$, and h is a bijective mapping from R_1 to R_2. This automaton is created based on the ADL description of a particular processor architecture. In this case, the set S_i of semantic actions is based on the behavioral model of the processor. This set also contains the internal semantic actions, such as various conversions of the attributes used in the operation, etc.

The automaton M_1 is used as a machine-code instruction parser, whose input alphabet contains only binary digits, i.e. $\Sigma_1 = \{0, 1\}$. The second automaton M_2 is used as a C-code generator. The set S_2 contains the modified C code from the ISAC processor description (i.e. from the behavioral description of particular instructions).

Each edge of this automaton has a label containg two items – the input symbol and the semantic action. The semantics actions can be either internal (e.g. conversion of binary number to integer number via internal function `bin2unsig()`) or external (e.g. usage of function `fprintf()` to generate a C code of the analyser).

(2) In the second part, the core of the translated simulator is created (i.e. the analyser generates a C code based on the target application). However, the generated output C code from the first part has to be properly structured. For example, if the C code generated by the analyser is stored within a single C function, it may be non compilable in a case of a large target application (e.g. because of problems with optimizations, problems with virtual memory). Therefore, the address space of the designed processor is partitioned into so-called *segments*.

Each segment has the same fixed size, which is set during the creation of an analyser by the developer. The size has to be equal to some power of two (e.g. 512 or 1024)[15]. Each segment is generated as one C-code function that simulates instructions within this segment. This function has one argument that is is used for passing the program counter. Furthermore, each such function contains a single `switch` statement, which uses this argument as its expression. Finally, each `case` statement is generated for every instruction within the particular segment. The bodies of each `case` statement are generated by the M_2 automaton. The functions are called from the main simulation loop.

In a straight-way approach, each `case` is generated for every particular instruction and it always ends with the `break` statement. However, this approach has two performance drawbacks related to the host-compiler optimizations.

(A) Firstly, each `break` ends the basic-block or even the generated function. In other words, the computed values (e.g. values of variables or expressions), which can be used in the next simulated instruction (i.e. the following `case`), are moved from the host registers to the main memory. From the host-processor point of view, it would be better to keep these values in processor registers as long as possible. (B) Secondly, the presence of a `case` construction in a sequence of statements does not allow additional optimizations because it presents a beginning of a basic block to the simulator code, which prevents usage of some host-compiler optimizations. Furthermore, these problems also lead to a worse cache hit/miss ratio. Therefore, the following enhanced approach is used.

We can effectively use the DWARF debugging information generated by the target C compiler for this task. More precisely, the analyser can use information about the starting and ending addresses of basic blocks in the target application and generate the `break` statement only if an address of an analysed instruction is equal to an ending address of some basic block. Otherwise, the analyser generates code related to the `BEHAVIOR` section of the `main` operation, which is used for preparing execution of the next instruction in the

[15]The reason for this action is explained later.

ISAC ADL. The code of this section is inlined to the generated C code instead of the `break` statement and it serves for synchronizing execution of instructions.

An illustration of the aforementioned principle is shown in Figure 5.2. In this example, we can see an application for some RISC processor with 32-bits long instructions, where the address space can be addressed by 8 bits. Firstly, the target application's code is divided into segments of the size 1024 bytes (i.e. segments 1-3). Each such segment may contain multiple basic blocks (i.e. basic blocks 1-4) that belong to an original target function (e.g. `target_functionA()` contains basic blocks 1 and 2).

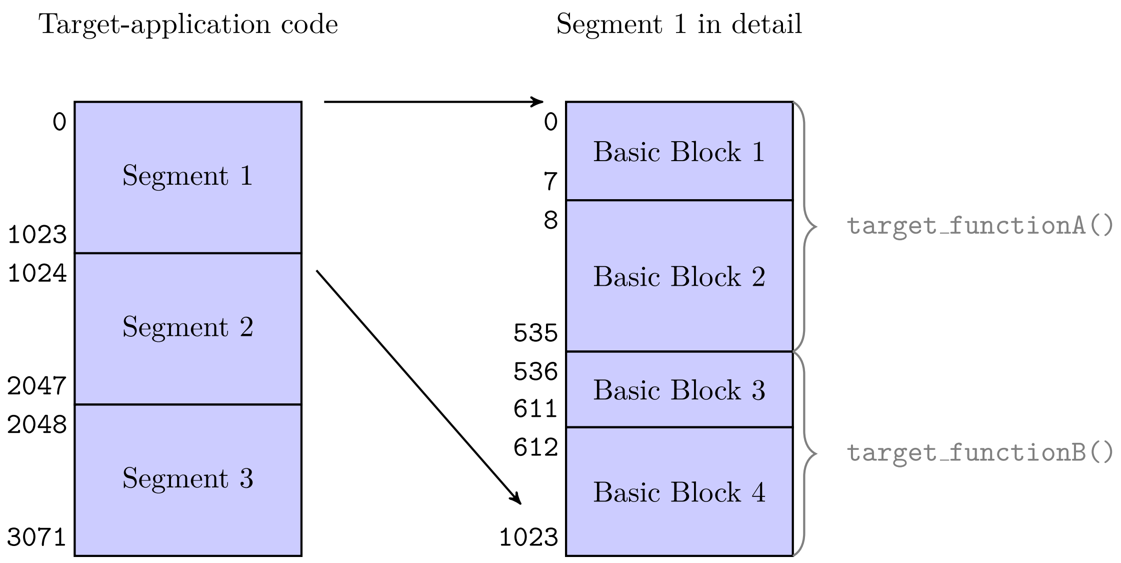


Figure 5.2: Example of an address-space partitioning in the translated simulator.

The count of bits needed for shifting is computed from the segment size (square root of segment size). The limitation of the segment size (power of two) guarantees fast transformation from the addresses to the keys used in the table. The number of bits, which is used for shifting, is obtained from the segment size (e.g. $log_2(1024) = 10$). This principle is similar to the approach used within the compiled simulator [27]. However, it differs in a set of allowed addresses. Within the compiled simulator, every instruction address is marked as a valid target address. Contrariwise, only addresses of basic-block borders are valid within the translated simulator.

(3) Finally, in the third part, the simulator itself is created via the compilation of the target application independent parts, such as the representation of the resources, and target application dependent part (i.e. functions generated by the analyser).

The main advantage of this simulator type is its speed (see Chapter 7). However, because of its aforementioned limitations (e.g. lack of SMC simulation), we provide an enriched version of this simulator, which removes these disadvantages. It is described in the following subsection.

5.2.3 Just-In-Time Translated Simulation (JIT)

The basic idea of a translated JIT simulator is to remove the first step of the static translated simulator creation. Compilation of the generated functions can take quite a long time; this especially applies to larger applications, or whenever the optimizations are enabled in the host compiler. Therefore, we enhanced the original approach by moving the analyser (i.e. the first phase) in the simulator itself. In this approach, the simulator accepts the target application as its input. As well as in the original approach, the application must contain information about the basic blocks (i.e. debugging information).

At the beginning of the simulation, the simulator operates as the interpreted simulator. The primary purpose of this simulation part is to find the hot-spots in the application. It is usually a function (e.g. with some calculation in a loop) or a set of functions. Because the address space of the processor is divided into segments, we do not mark the single function as a hot-spot, but we mark the whole segment in which the function is placed. The hot-spot detection algorithm uses a threshold value, which is used to distinguish whether the code is or is not a hot-spot. This threshold is set by the developer at the beginning of the simulation and it is represented as an integer value indicating how many times the segment has to be hit in order to be marked as a hot-spot. When the first hot-spot is found, the recompilation of the segment starts.

The segment recompilation is done in three steps. (1) Firstly, the hot-spots are analysed. (2) Afterwards, the result of this analysis is compiled. (3) Finally, the output of the compilation is dynamically loaded into the running simulator and the simulator performs the function-table update. Stopping the simulator in order to recompile the segment would be ineffective. Therefore, we used an advanced approach when the simulator continues in the interpreted simulation. Meanwhile, the segment is recompiled in a separate thread. This solution has a very good performance on modern multi-core processors because the recompilation and the interpreted simulation are running in different cores, see Chapter 7 for experimental results.

Now we can illustrate this concept in deeper detail. Firstly, for the sake of simplicity, let us assume the target application does not contain SMC. When the simulator detects that the threshold for some segment is met, the simulation is paused, and the content of the segment is stored into an internal variable. Afterwards, the recompilation thread is created and the simulation is resumed (in order to avoid races between the simulation thread and the recompilation thread in the case of the self-modifying code).

The recompilation thread executes the internal analyser and the internal variable is given as its input. The analyser is based on the same formal model and it works in the same way as in the static translated simulation. In other words, it also emits the C code representing behavior of a given segment. Because only one segment was given, the output file contains only one function. In the second step, this file is compiled as a shared dynamic library (i.e. `.so` file on Unix-like systems or `.dll` file on MS Windows). The library contains position independent code, which means that it can be loaded by a program while the program is running. This feature is supported by all major operating systems (`dlopen()` function on Unix-like systems or the `LoadLibrary()` function on MS Windows). The last step of segment recompilation is an update of the function table. As was mentioned in the previous section, the function table contains the functions that simulate a particular segment. Note that in the case of the JIT translated simulator, this table is empty at the beginning of the simulation and it is filled during the simulation. The new function is simply stored into the table according to the address of the segment. The next time, when the segment is hit, the stored function is executed (i.e. the segment is no longer simulated in the interpreted way).

Now, let's assume the target application contains the self-modifying code. The simulator has to have a possibility to detect a code change. We use an approach that detects modification of a memory. Whenever this detection module detects a write access, it informs the simulation core. Afterwards, when the simulation hits a modified segment, the analyser has to emit a different code for this segment because of the following reasons.

Initially, the information about basic blocks is no longer valid for the changed segment and we cannot use this information anymore. For example, some particular instruction (e.g. `move` instruction) is stored within a basic block and it is rewritten as a `call` instruction during application's run-time. This `call` instruction breaks the original basic block in two, but the simulator does not have such information.

Therefore, the analyser has to emit code in the form, which was mentioned in the previous subsection in the part dealing with the straight approach (i.e. each `case` statement ends with the `break` statement and there are `cases` for all instructions within the segment in the `switch` statement). In other words, the `case` in the generated `switch` simulates only one instruction instead of possibly several instructions within the basic block. In fact, this is the compiled version of the simulation [27]. Furthermore, there can be a situation, when we need to analyse changed adjacent segments. For instance, it happens if instructions in the instruction set have different bit width (e.g. 16-bit and 32-bit instructions in the ARM/Thumb architecture). Such instruction can overlap from the end of one segment into the next segment (also there is no way of how to distinguish the instructions from data [3]).

5.3 Source-Level Debugger

A classical computer-programming aphorism says that despite all programmers efforts, every meaningful, non-trivial program contains at least one bug. The process of finding and removing these bugs can be done in several different ways (e.g. stepping through instructions, printing debug messages, inspecting called functions, displaying values of registers or memory). In Section 4.5, we mentioned different types of debuggers that can be used for this dynamic machine-code analysis and we stated that the source-level debugging is the most appropriate one.

This type of debugging is interactive and it is visualized on a level of the original application's HLL source code. Therefore, it differs from the other debugging methods like program instrumentation, post-mortem analysis, profiling, or memory allocation checking. The source-level debugger is usually implemented as an extension to an existing instruction-level debugger. Furthermore, it creates some sort of abstraction over the underlying machine code and other architecture-specific attributes. Therefore, the user has an illusion that he or she is debugging on the level of HLL statements; however, the debugged application (also called as *debuggee*) is running on the machine-code level in real.

In the rest of this section, we present a concept of the retargetable source-level debugger together with a description of the steps needed for its proper implementation. Firstly, in Subsection 5.3.1, we mention the process of debugging-information generation within the Lissom C compiler. Implementation of the source-level debugger is presented in Subsection 5.3.2.

5.3.1 Obtaining Debugging Information

In comparison with the solution described in the previous subsection, the proposed source-level debugger needs an additional source of information – the debugging information describing the relation between the application's machine code and its original source code. We have chosen the DWARF debugging format for our solution because it is one of the most complex standards and its generation is already supported within the LLVM compiler that is used as a core of the Lissom C compiler.

However, it was necessary to slightly extend this compiler toolchain (e.g. C compiler front-end and back-end, assembler, and linker) for proper DWARF information generation and handling. Furthermore, it was necessary to extend the Lissom assembler tool to generate the DWARF debugging sections (e.g. `.debug_line`) that are not generated directly by the C compiler because some information is not available in this phase. Similarly, it was necessary to create mapping between the DWARF abstract registers and the real processor registers. Those abstract registers are used internally by DWARF to compute different expressions, such as location of a local variable on stack, position of a function return address, etc.

The debug information is now stored in the same way as in the ELF file format, i.e. as binary data in named sections [6]. This concept allows source-level debugging for all languages supported in LLVM (e.g. C++, Objective-C) out of the box. However, support of other languages than C is marked as future research because it has to be properly tweaked and tested.

5.3.2 Implementation

The initial phase of source-level debugging is a process of obtaining and processing the debugging information. We have solved this task via the aforementioned library `dwarfparserl` presented in Subsection 5.1.1. After the DWARF information is successfully extracted and parsed into the internal data structures, the debugger proceeds with a application debugging in a newly created simulator. Consequently, this simulator is controlled by the debugger and it can be stopped or resumed whenever needed. Furthermore, it is possible to obtain resource values of the simulated system (e.g. memory, program counter, other registers).

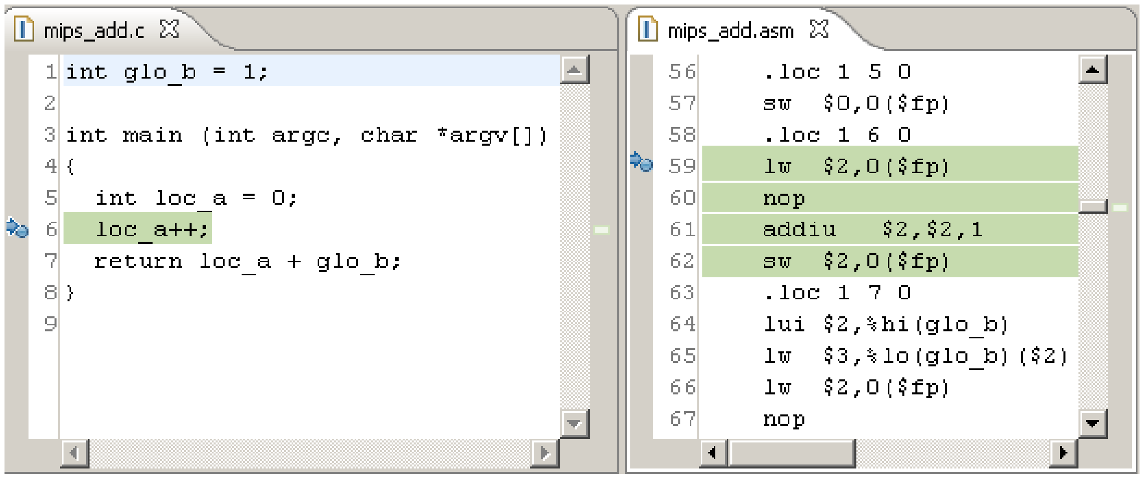
In what follows, we briefly describe the debugger functionality and the internals behind it. For a more detailed description, see [14] and [19].

Line-Information Mapping

Visualization of an actual position within the original source code (i.e. a line of code or a particular sub-expression within the line) is an essential feature of each source-level debugger. This task is done via the aforementioned DWARF information stored in the `.debug_line` section. Afterwards, whenever the program is stopped (e.g. a breakpoint is hit), the source-code location is visualized either in GUI or CLI. Within GUI, this is implemented as highlighting of a particular HLL code line. Furthermore, it implies highlighting of one or more assembly code lines (i.e. the original source statement is usually translated to multiple machine-code instructions) whenever such a file is available and opened, see Figure 5.3.

Inspection of Variables

The inspection of HLL variables is implemented in a very similar way. The variable location changes dynamically during the application execution. For example, the variable is usually mapped to a particular memory location or stack, but the compiler often optimizes this situation and it stores the variable temporarily into a processor register. This task is once again solved by using the DWARF debugging information, but we use the `.debug_ranges` section this time.



```
int glo_b = 1;

int main (int argc, char *argv[])
{
  int loc_a = 0;
  loc_a++;
  return loc_a + glo_b;
}
```

```
    .loc 1 5 0
    sw  $0,0($fp)
    .loc 1 6 0
    lw  $2,0($fp)
    nop
    addiu   $2,$2,1
    sw  $2,0($fp)
    .loc 1 7 0
    lui $2,%hi(glo_b)
    lw  $3,%lo(glo_b)($2)
    lw  $2,0($fp)
    nop
```

Figure 5.3: Screenshot of the source-level debugger (lines mapping).

Breakpoints and Program Stepping

After a successful simulator initialization, which means loading a debugged application into the memory and initiating internal data structures, it is possible to control its execution. The basic tool for controlling execution is a program breakpoint. The breakpoint can be applied either at the source-code level or on assembler level. In both cases, it invokes placement of a *physical* breakpoint to the program machine code[16].

Breakpoints placed in the HLL code are called *logical* breakpoints. Setting the logical breakpoint implies the creation of several *internal* (i.e. not visible to the developer) physical breakpoints because the HLL statement is usually executed as a sequence of machine instructions. The algorithms for breakpoint mapping and managing are very similar to algorithms listed in [32].

It also is necessary to allow proper stepping over the debugged HLL code by using the features such as step over, step into, step out, etc. Some of these features related to function calls (e.g. step over function call) are actually quite challenging for the retargetable approach because the debugger has no specification about which instruction is actually a function call as is usual in the platform-specific debuggers. Therefore, we have to use other DWARF hints stored in the CFI structure (call frame information). The CFI structure is also used within the stack backtrace feature described in the last part.

Stack Backtrace

In order to provide an accurate information about the actual program state, it is necessary to track all function calls. Such information is implemented as a function call hierarchy. Each item represents one function call together with its arguments and the original calling address. This information is gathered from the call stack by the unwinding algorithm [32]. Note the stack backtrace structure is platform dependent as well as the application binary interface (ABI) used for calling conventions. Its description must be obtained from the debugging information and from the processor description in the ADL.

[16]For further details about *physical* and *logical* breakpoints see [32].

Chapter 6

Retargetable Static Analysis

This chapter is based on publications [16–18, 20, 39, 40] and reports [35, 36]. It describes the author's contribution within the area of the static machine-code analysis.

In comparison with the previous chapter, the current chapter presents an overview of only one newly created tool – the retargetable machine-code decompiler. However, as we will see, its development and implementation is a far more challenging task than the previously mentioned tools. Although, the original design of the decompiler is the author's idea [13], the implementation and further extension has been done in cooperation with the other members of the Lissom team, most importantly with Petr Zemek, Lukáš Ďurfina, and Peter Matula.

The structure of this chapter is as follows. Firstly, in Section 6.1, we describe the motivation and fundamentals of machine-code decompilation. Afterwards, in Section 6.2, the structure of the decompiler is presented as well as a description of its inputs, outputs, and intermediate forms of code representation. Each of the decompiler parts is described separately. The preprocessing phase is described in Section 6.3. After that, in Section 6.4, we describe the front-end part. The following Section 6.5 deals with the middle-end phase. The last decompilation phase, the back-end part, is described in Section 6.6.

Furthermore, the proposed decompiler can be tested via our on-line decompilation service (see [37] for details):

http://decompiler.fit.vutbr.cz/decompilation/

6.1 Motivation

Machine-code decompilation is a reverse-engineering discipline focused on reverse compilation [3, 7]. It performs an application-recovery from binary executable files containing machine code back into an HLL representation (e.g. a C source code), see Figure 6.1 for illustration. The decompiled code must meet two essential criteria. It has to be functionally equivalent to the input binary code, and it has to be highly readable.

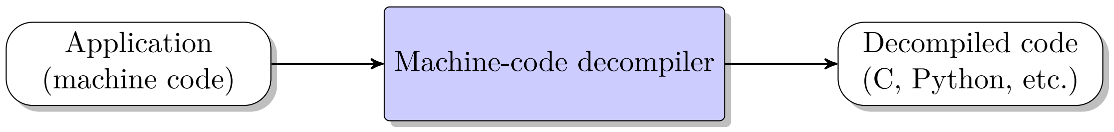


Figure 6.1: Illustration of a decompilation process.

As we can see, the process is similar to compilation, except the inputs and outputs are reversed. However, in contrast to compilation, the process of decompilation is much more difficult because the decompiler must deal with a massive lack of information on its input. Furthermore, the input machine code is often heavily optimized by one of the modern compilers (e.g. GCC, MSVC), which makes decompilation even more challenging. The situation is also aggravated by the fact that the executable application may be originally written in any programming paradigm, language, and it could be compiled by different compilers.

6.2 Overview of the Retargetable Decompiler

In this section, we present an overview of the retargetable decompiler developed within the Lissom project. This description is based on [18, 39, 40]. A more detailed description of all decompilation phases is provided in the remaining sections of this chapter. It should be noted that at present, there is no other competitive retargetable decompiler.

Our decompiler is supposed to be independent of any particular target architecture, operating system, or EFF. The decompiler is designed as a toolchain consisting of four basic parts—a *preprocessing*, a *front-end*, a *middle-end*, and a *back-end*, as we can see in Figure 6.2.

The interconnection of these phases is done by using several shell scripts written by the author, which are unimportant for this thesis, and their description is omitted, see [36].

Before we discuss these phases, we should mention the IR format used in these phases for representing the application's behavior. As we already mentioned, the decompiler is similar to a classical compiler at a point of the decompilation process when the input application is transformed into an intermediate code. Some of the transformations and optimizations performed over this IR are identical to those used in compilers. We decided to choose the LLVM project for this task. Therefore, we use the LLVM IR format as our internal IR across the front-end, middle-end, and back-end phases.

Firstly, in the *preprocessing phase*, the input application is deeply analysed in order to obtain as much information as possible. As we will see, every available information is useful during decompilation. Within this initial phase, we try to detect the application's EFF (e.g. ELF, WinPE), the target architecture (e.g. x86, ARM), information about code and data sections, presence of debugging information, information about the original programming language, information about the used compiler, and many others.

Moreover, applications can be also packed or protected by the packers or protectors (see Section 4.1 for definitions of these terms). This is a typical case of malware. Therefore, such input must be *unpacked* before it is further analysed; otherwise, its decompilation will be inaccurate or impossible at all.

The last tool of the preprocessing phase is a binary-file converter that is used for transforming binary applications from a platform-specific EFF into an internal, uniform COFF-based file format. The conversion is done via our plugin-based converter. Currently, we support conversions from Unix ELF, WinPE, Apple Mach-O, and other EFFs.

Other, non-standard, file formats can be supported via implementation of the appropriate plugin, or by using a state-of-the-art automatic parser based on the formal description of this EFF in our description language. It should be noted that other than the previously listed formats are rarely used in practice.

Afterwards, the converted application is processed by the *front-end phase*, which is the only platform-specific part of the decompiler because its instruction decoder is automatically

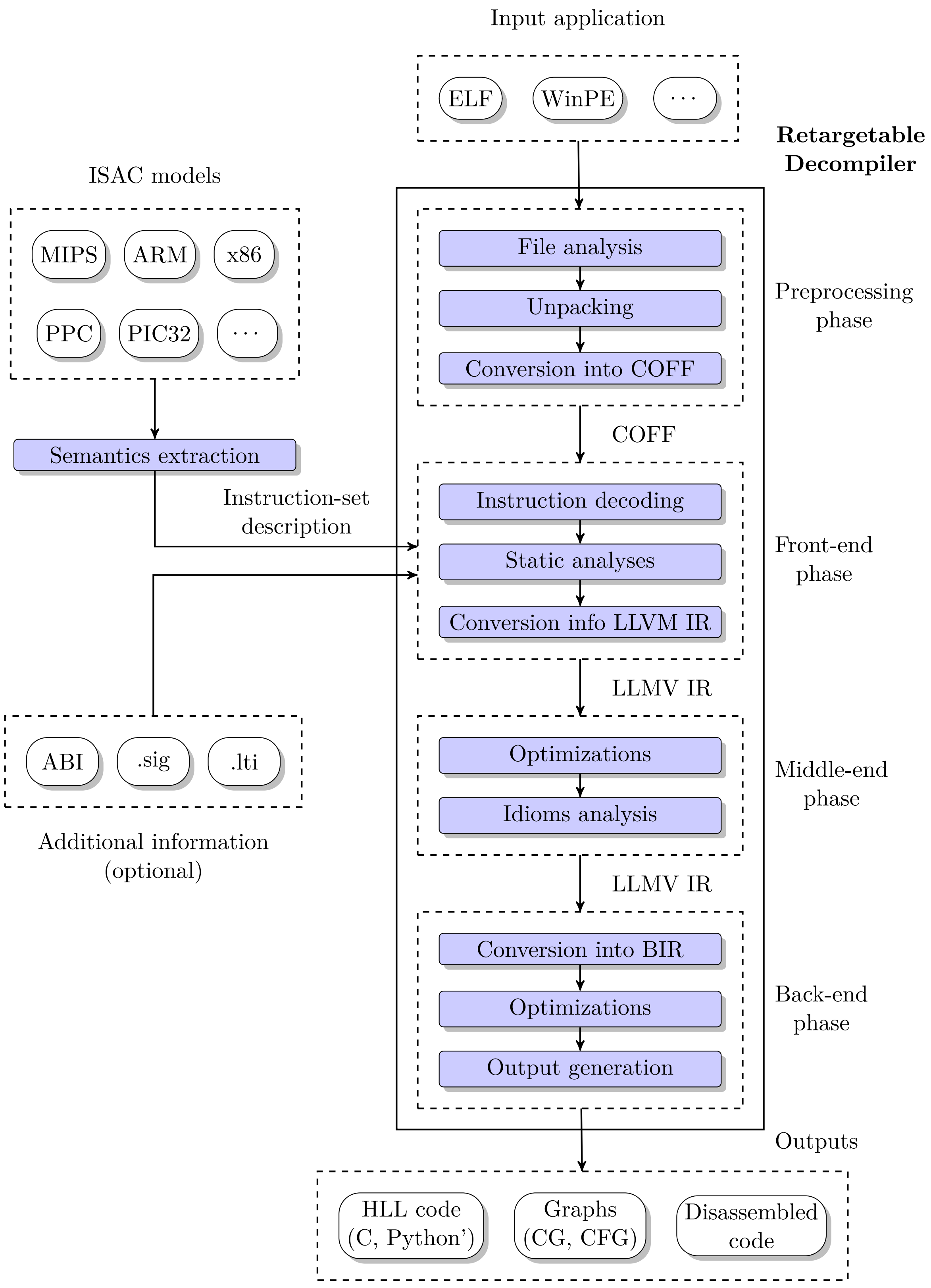


Figure 6.2: Simplified structure of the retargetable decompiler.

configured based on the target architecture model in the ISAC ADL described in Section 4.7. The ISAC model is transformed by a semantics extractor [8], which transforms the semantics description (i.e. the snippets of C code) of each instruction into a sequence of LLVM IR instructions, which properly describe its behavior. This complex information is used for automatic configuration of the instruction decoder. The instruction decoder translates the application's machine code into LLVM IR instruction sequence, which characterizes its behavior in a platform-independent way. This intermediate program representation is further analysed and transformed by several analytical methods within the front-end phase. Finally, the front-end passes the LLVM IR of the input application to the *middle-end phase* for further processing.

Within the middle-end phase, the LLVM IR program representation is optimized by using many built-in optimizations available in LLVM and our own passes (detection and transformation of the so-called *instruction idioms*).

Finally, the optimized LLVM IR representation is passed to the *back-end phase*. This phase is responsible for a reconstruction of HLL constructions and emission of the resulting code as the target HLL. Currently, we support C and a Python-like language as the target languages. Furthermore, the back-end can also generate a call-graph and control-flow graphs of the decompiled application. Another output, the disassembled code, is generated directly from the front-end part. This code is enhanced by the HLL information (e.g. mapping of instructions to functions, distinction whether the instruction belongs to a user code or to a standard-library function) and it is more valuable for analysis than outputs generated by other disassemblers.

At present, the decompiler supports decompiling applications for the following architectures: MIPS, ARM (with Thumb extension), PIC32, PowerPC, and Intel x86. Creation of these models takes usually one to three months for one person, depending on the complexity of the architecture. This will be difficult to achieve by a manual extension of an existing decompiler.

6.3 Preprocessing Phase

In this section, we present the design of the preprocessing phase. This phase is responsible for an initial analysis of the input application, its unpacking, and conversion into internal COFF-based file format. In addition to the converted application, this phase also produces a configuration file that contains gathered information about the file and settings of the decompilation process (e.g. user specified output language, enabling emission of graphs). This configuration file is used in all the subsequent phases (e.g. selection of an appropriate ISAC model based on the detected application's architecture).

This concept consists of four basic parts that are depicted in Figure 6.3. (1) Firstly, the input application in one of the supported file formats (e.g. ELF, WinPE) is deeply analysed by one of our tools called `fileinfo`. Its task is to extract all the available information, as we revealed in the previous section, and it stores this information into the internal configuration file. Moreover, this tool is also responsible for a detection of the originally used language and compiler. This feature is described in detail in Subsection 6.3.1.

(2) Afterwards, the file has to be unpacked whenever the usage of a known packer has been detected in the first part. This is done by our plugin-based unpacker that is described in Subsection 6.3.2.

(3) Finally, the (unpacked) application is transformed into the aforementioned COFF-based internal EFF. This is primarily done by our another plugin-based application called

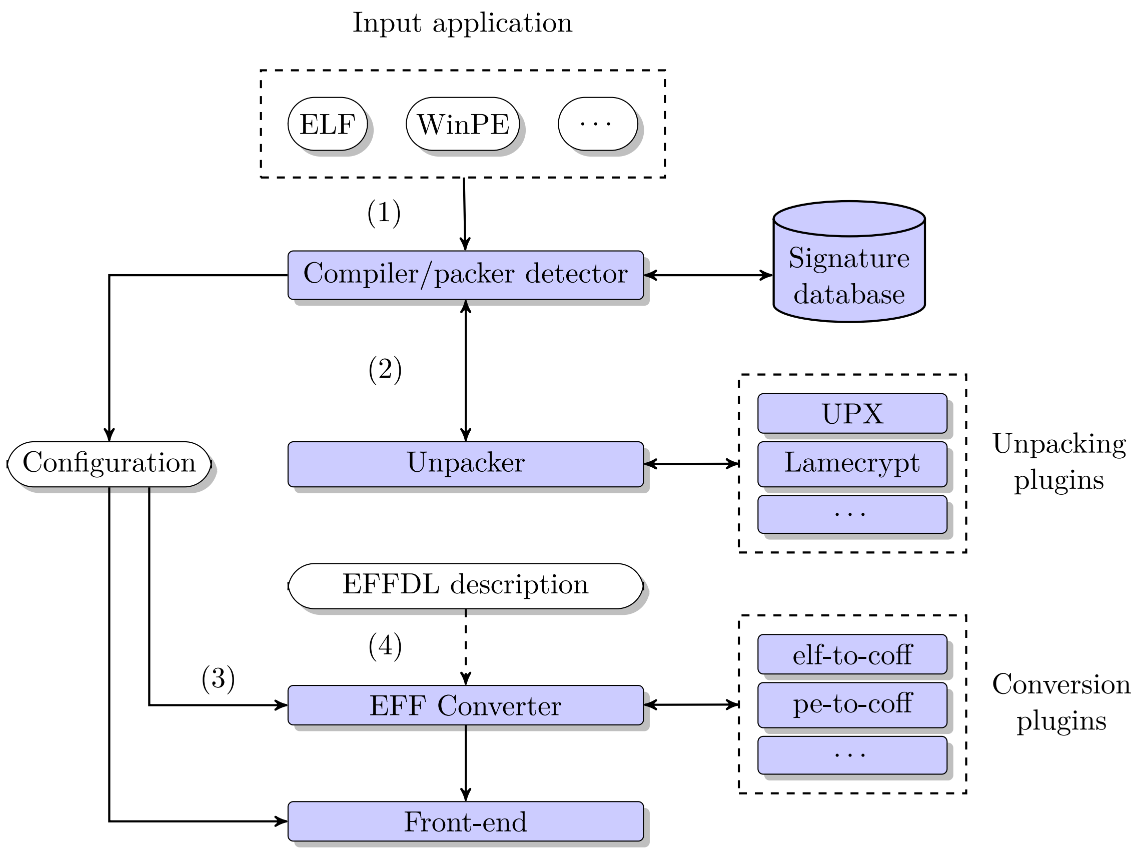


Figure 6.3: Concept of the preprocessing phase.

`bintran` (Binary Transformation). This tool is described in Subsection 6.3.3.

(4) We also propose a novel EFF-description language (EFFDL) that can be used for automatic generation of the EFF parser and converter based on the EFF description in this language. This language is described in the Subsection 6.3.4. It is fair to note that this feature is still under development and it was tested only on the ELF format.

6.3.1 Compiler and Packer Detection

At the beginning of the first decompilation part, the input executable file is analysed and the used EFF is detected. At present, we support all the commonly used formats (e.g. WinPE, UNIX ELF, Mach-O). The next step is the detection of a tool (e.g. compiler, packer, linker) used for executable creation. The information about the originally used compiler is valuable during the decompilation process because each compiler generates quite a unique code in some cases; therefore, such knowledge may increase a quality of the decompilation results.

Our approach of detecting the originally used compiler (or packer) is based on two methods. The first one is done by using a signature-based detection of start-up code. This method is described first. Afterwards, we describe the second method that is based on heuristic detection.

An example of a start-up code can be seen in Figure 6.4. Signature for this code snippet is “`5589E583EC18C7042401000000FF15--------E8`”, where each character represents a nibble of the instruction's encoding. All variable parts must be skipped during matching by a wild-card character “`-`”, e.g. a target address in the `call` instruction.

```
; address    hex dump                    Intel x86 instruction
  0040126c:  55                          push %ebp
  0040126d:  89 e5                       mov  %esp, %ebp
  0040126f:  83 ec 18                    sub  $0x18, %esp
  00401272:  c7 04 24 01 00 00 00        movl $0x1, (%esp)
  00401279:  ff 15 00 00 00 00           call *0x0
  0040127f:  e8                          ...
```

Figure 6.4: Start-up code for MinGW GCC v4.6 on Intel x86 (`crt2.o`).

Our signature format also supports other features for more accurate signature's description, see [34]. Unfortunately, some polymorphic packers (e.g. Morphine encryptor) cannot be described via extended format of signatures because they generate entirely different start-up routine for each packed file. Therefore, we also support the concept of additional heuristics that are primarily focused on polymorphic packers. These heuristics analyse several properties of the executable file (e.g. attributes of sections, offset of EP) and they perform the detection based on a packer-specific behavior. An example of such heuristic is illustrated as a pseudo-code in Figure 6.5.

```
if(file_format == WIN_PE && target_architecture == INTEL_X86 &&
   EP_file_offset >= 0x400 && EP_file_offset <= 0x1400 &&
   sections[0].name == ".text" && sections[1].name == ".data" &&
   sections[2].name == ".idata" && sections[2].size == 0x200)
{
     return "Morphine 1.2";
}
```

Figure 6.5: Heuristic detection of the Morphine encryptor v1.2.

Apart from heuristics for the precise detection of polymorphic packers, we also support simpler heuristics, which are focused only on a name, number, and order of sections. Such heuristics cannot detect the exact version of the used packer, but they are useful if the signature database does not contain an entry for a related tool.

6.3.2 Unpacking

Whenever usage of a packer is detected by the `fileinfo` application, the unpacking part is invoked. Executable-file packing is done for one of these reasons—code compression, code protection, or their combination. Unpacking within the proposed retargetable decompiler is done by our own plugin-based unpacker, which consists of a common unpacking library and several plugins implementing unpacking of particular packers. The common library contains the necessary functions for rapid unpacker creation, such as detection of the original entry point (OEP), fixing import tables, etc. Therefore, a plugin itself is tiny and contains only code specific to a particular packer.

At present, we support unpacking of several popular packers like UPX (Linux and Windows), AcidCrypt, etc. These plugins are able to generate valid executable files with removed protections. In future, we would like to focus on a further extension of this unpacking concept by adapting the generic unpacker as described in Chapter 8.

6.3.3 Executable-File-Format Conversion

At this point, the input executable application is stored in a platform-specific EFF (e.g. ELF, WinPE), which makes its decompilation harder because it is necessary to handle each EFF in a different matter. Therefore, we decided to convert the input applications from these platform-specific EFFs into a uniform internal file format. We chose the COFF-based representation presented in Subsection 4.7.2 for this purpose. By using this uniform format, the decompiler can deal with every application in the same way regardless of its origin.

Our EFF-conversion solution is designed as a plugin-based system, where each plugin implements a conversion of one or more EFFs. The main converter, the host application `bintran`, handles the user interface, manages conversion plugins, and provides the common functionality for plugins (e.g. COFF file manipulation) [34].

The Binary File Descriptor (BFD) library is used for handling ELF, PE, and Mach-O files [2]. During the conversion, the content of an input file is mapped to a BFD internal canonical structure, uniformly characterizing items like sections or symbols. Afterwards, this form is transformed into COFF format via the conversion plugin `bfd-to-coff`, and finally saved in this format by the internal library `objfilelib`, see Figure 6.6. It should be noted, that the inverse conversion (i.e. plugin `coff-to-bfd`) is also possible. Apart from the most commonly used formats (i.e. ELF, WinPE, Mach-O), the converter also supports two additional EFFs – Symbian E32Image and Adroid DEX.

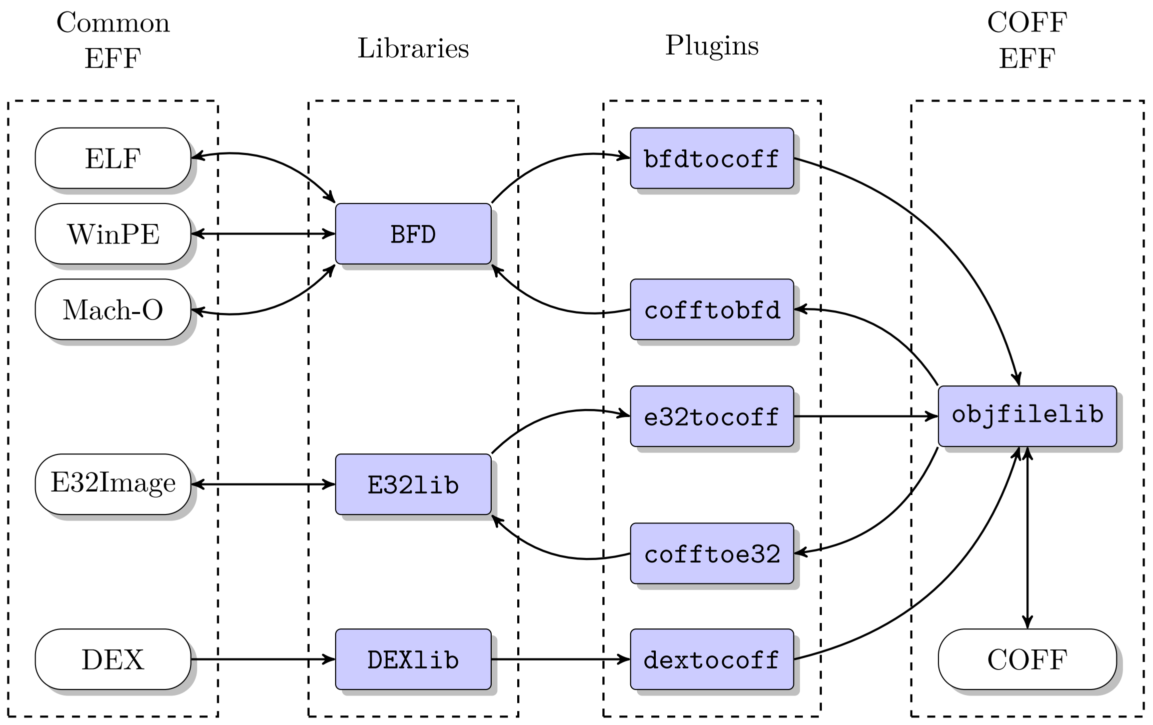


Figure 6.6: Principle of the plugin-based EFF converter `bintran`.

After the conversion, the COFF representation of the input application is processed by the front-end of the retargetable decompiler.

6.3.4 Context-Sensitive Description of Executable-File-Formats

The main drawback of the aforementioned plugin-based EFF converter is the implementation complexity of each newly created plugin because each such plugin has to be written manually. In other words, this part is not as generic (or retargetable) as the other parts of the decompiler.

In order to achieve a truly retargetable decompilation, we should apply the concept similar to one used for description of target processor architectures and automatic generation of the front-end part. This task can be divided into two steps. (1) Description of a target EFF by using a specific EFF-description language. (2) Automatic generation of an EFF converter based on this description. Afterwards, the converter can be used for conversion of applications stored in a particular EFF into an internal code representation used by a decompiler. Once this concept is adopted, the description complexity of the new EFFs should significantly decrease.

However, this is a quite challenging task because the structure of most EFFs (e.g. ELF, WinPE) is relatively complex because its elements are mutually interconnected and the structure is heavily influenced by content of these elements (i.e. they create a context-sensitive behavior). We introduce our own EFF-description language called EFFDL that is capable to describe the context-sensitive elements and it is planned to be used as an alternative to `bintran`. However, this feature is still under development and it was tested only on the ELF format.

Modified Extended Backus-Naur Form (EBNF) is used for a EFFDL grammar's syntax description. Terminal symbols are typeset in `red`. The symbol ∼ is used for concatenation. Sequences (i.e. zero or more repetitions) are denoted by `{}`; optional constructions (i.e. zero or one occurrence) are denoted by `[]`; finally, selections (i.e. a choice between more constructions) are denoted by `|`. The grammar is depicted in Figure 6.7. For information about the particular grammar elements and parsing of this language, see [16].

```
start          ->  root analyser_def { analyser_def } { production }
analyser_def   ->  analyser id ( [ offset [ , offset ] ] ) { { statement ; } }
statement      ->  element [ { semantic_actions } ]
               ->  analyser_id { [ times ] } [ { semantic_actions } ]
element        ->  type id_attribute
               ->  type [ value ]
analyser_id    ->  id_attribute ( [ offset [ , offset] ] )
type           ->  ( int | uint ) ∼ bitwidth_size { [ array_size ] }
attribute      ->  [ < id { , id } ] > ]
production     ->  ( id_attribute { , id_attribute } ) ->
                   ( [ id'_attribute ] { , [ id'_attribute ] } ) [ priority ]
```

Figure 6.7: Simplified grammar of the EFFDL.

Furthermore, we defined a new formal model that is used for EFFDL description – a regulated version of the scattered context grammars (see Definitions 4 and 5), where productions are regulated by the *production-priority function*. This priority function guarantees that productions will be applied whenever it is possible according to their priority, see [20] for details.

Definition 9. A *(propagating) scattered context grammar with priority*, abbreviated as ((P)SCGP), is a quintuple $G = (V, T, P, S, \pi)$, where (V, T, P, S) is a (propagating) scattered context grammar and π is a *function* $\pi : P \to \mathbb{N}$.

6.4 Front-End Phase

The next decompilation phase is done within the front-end part. This part is responsible for translation of input platform-dependent machine-instructions into an independent code representation in the LLVM IR notation. Within this phase, several methods of static code analysis are applied, such as separation of code and data, instruction decoding, CFA, DFA, preliminary reconstruction of HLL constructs (functions, arguments, variables, data types, loops, etc.), and finally emission of front-end outputs (disassembled code and LLVM IR representation of application that is further processed in the remaining decompiler phases). The general structure of this part is displayed in Figure 6.8.

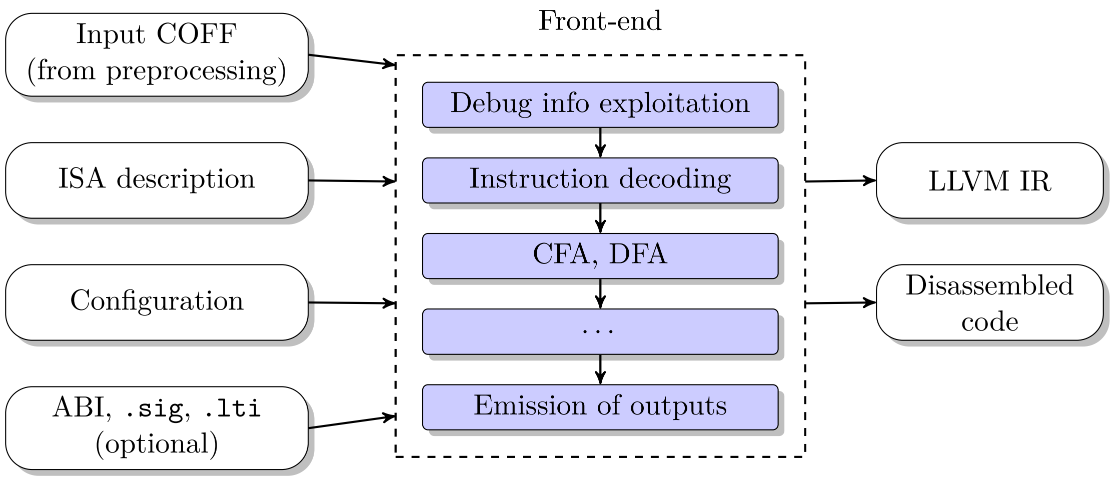


Figure 6.8: General structure of the front-end part.

The decompiled application is stored in the COFF representation produced by the `bintran` application. The last mandatory input of this phase is a description of the instruction set (ISA) as described in Figure 4.1. Finally, the front-end also supports the following optional inputs – ABI description, signatures of statically linked code, and function type information.

The front-end analyses are called in a given order based on their usage and interdependence (e.g. CFA must precede function detection). Some of these methods are called iteratively till their convergence (e.g. no other function is detected). We will describe only the most important methods in this section. These methods are described in order of their application. Furthermore, there are many more analyses used within the front-end part.

6.4.1 Debugging and Symbolic Information Exploitation

As we previously stated, we are dealing with a massive lack of information during the decompilation process. Therefore, we try to to exploit every available piece of information, such as debugging[17] or symbolic information. Symbolic information is very similar to

[17]We support both PDB and DWARF formats within the decompiler, see [17].

debugging information—it is an additional piece of information stored within the executable by a compiler. However, it only contains information about symbols (e.g. function names) and their positions in code sections. On the other hand, presence of symbolic information is more common in real-world applications.

One of the first steps is to check for the debugging/symbolic information presence and process it by the aforementioned libraries, see Section 5.1. Afterwards, it is ready to be used in the decompiler's front-end, which updates its intermediate code representation based on this extracted information.

At present, the decompiler uses all of the common information (e.g. module information, function information, variables, line information). In Figure 6.9, we can see a brief comparison of three different decompilation cases on the executable file containing the factorial function `int fact(int n)`.

```
/*
Module: input.c
Line range: 6 - 10
Address range:
 0x401560-0x401585
*/
int fact(int n)
{
  if (n != 0)
  {
    return n *
      fact(n - 1);
  }
  else
  {
    return 1;
  }
}
```

(a) Debugging information.

```
int fact(int a1)
{
  if (a1 != 0)
  {
    return a1 *
      fact(a1 - 1);
  }
  else
  {
    return 1;
  }
}
```

(b) Symbolic information.

```
int f401560(int a1)
{
  if (a1 != 0)
  {
    return a1 *
      f401560(a1 - 1);
  }
  else
  {
    return 1;
  }
}
```

(c) Stripped file.

Figure 6.9: Decompilation of the factorial function under different circumstances.

6.4.2 Instruction Decoding

The remaining machine code, i.e. the code that was not eliminated as statically-linked standard functions, represents the user-created code that has to be decompiled. The first part of this translation process is the *instruction decoding*.

Because every architecture has its own instruction set, the *instruction decoder* for the particular architecture is automatically created based on its instruction-set. For this task, we utilize the instruction-set description extracted from the ISAC model, see Figure 4.1. It contains all the necessary information for usage in decompilation: binary encoding, assembler syntax, and semantics of each particular instruction.

The decoder is responsible for translating architecture-specific machine-code instructions into an internal code representation as a sequence of low-level LLVM IR instructions (i.e. a basic block with several LLVM IR instructions for each input machine instruction) that is further analysed and transformed by the subsequent analyses.

As we can see, its functionality is similar to a disassembler, except that its output is not an assembly language, but rather the semantics description of each instruction. Furthermore, this part has to deal with platform-specific features. For example, it has to support architectures with different endianness, multiple instruction sets, etc.

6.4.3 Control-Flow Analysis (CFA)

Control-flow analysis is another very important analysis because the other analyses depend on its results. The aim of CFA is to divide the code into basic blocks (see Definition 6) and create a CFG over them (see Definition 7). Both are used in the later analyses (e.g. function detection).

Roughly speaking, CFA is all about analysis of branch instructions during traversal of decoded instruction. Firstly, we need to recognize branch instructions and then mark them according to their purpose (e.g. conditional branch used in `if` statements, function call, return from function). The difficult part is that the purpose of a branch can be changed whenever we retrieve more information (e.g. branch on some address can become a tail call when we find out that this address is an address of a function).

The target address of the branch instruction can be either stored directly as an immediate value and it can be obtained right after its decoding, or the *indirect branch*, which means that an address is stored in some register or memory address during application execution. This may be tricky because decompilation is a static analysis. In such cases, we use our static code interpreter.

6.4.4 Static Code Interpreter

Whenever an analysis needs to investigate a value of a particular instruction operand on a given address (e.g. tracking a register value, stack value, target address of a branch), we use our method called *static code interpreter*, which tries to compute this value by using static interpretation of the code and backtracking the value of the operand over CFG.

6.4.5 Function Detection

Within the function-detection analysis, we use a combination of two detection methods that are based on [9]. The first one uses a top-down approach and the second one uses a bottom-up approach. By using the top-down approach, it is possible to recognize function headers, and by using the bottom-up analysis, we can detect their bodies.

The original principles were tied to particular target architectures, e.g. knowledge of an instruction used for function call and return. However, we enhanced them to be platform independent (retargetable) by processing over LLVM IR representation. Furthermore, we interconnected the top-down and bottom-up approaches and their combination creates a bidirectional function-detection algorithm. A more detailed description can be found in [39].

6.4.6 Data-Flow Analysis (DFA)

The data-flow analysis is used within the front-end part mainly for data-type reconstruction and for function-arguments reconstruction. The former one is beyond the scope of this thesis because of its complexity, while the latter is described in the rest of this subsection.

DFA is based on a *memory-places* analysis originally presented in [42], which was tailor-coded for Intel x86 and x86-64 architectures. Once again, we enhanced this approach to be general enough for our usage, see [39] for details.

Firstly, let R be a set of general-purpose registers and flag registers, and M be a set of all places for storing values in memory and on stack. Then the *memory place* $l \in R \cup M$ can contain a value of a variable stored in this particular locations. The algorithm afterwards inspects the caller–callee pairs in order to detect a usage of memory places for passing the input arguments and output arguments (i.e. return values) between the caller and callee. DFA computes these arguments from instructions that access the stack or registers. Furthermore, memory-places are also used for storage of a function return address.

6.4.7 Generation of LLVM IR

An IR generator is the last part of the front-end. The task of this part is to generate LLVM IR in a text representation (the `.ll` file). An example of LLVM IR code snippet produced by the front-end is presented in Figure 6.10.

```
;804857a  1110101100010011  eb  13
;JMP {19} decode__instr_grpxx_op1_eip32_rel8__instr_jmp_rel8__op1
%u0_804857a = add i8 19, 0 ;used signed value. Unsigned value: 19
%u1_804857a = sext i8 %u0_804857a to i32
%u2_804857a = add i32 134514042, 0  ; Assign current PC
%_e_804857a = add i32 2, 0
%u3_804857a = add i32 %u2_804857a,  %_e_804857a
%u4_804857a = add i32 %u3_804857a,  %u1_804857a
br label %pc_804858f ;4 * %u4_804857a
```

Figure 6.10: Example of a generated LLVM IR code for one machine-code instruction.

6.4.8 Generation of Disassembled Code

The front-end part is also able to emit the disassembled code. In comparison with other disassemblers (e.g. `objdump`), our solution produces more readable code by using information about the reconstructed HLL code, such as hints about branch and call target locations (even for indirect branches), original function names (based on debugging information), names and values of referred string literals as comments, etc.

6.5 Middle-End Phase

In this section, we move to the middle-end and present methods used within this part, which is responsible for optimizing the intermediate representation that is the output of the front-end. A general view concerning this part is shown in Figure 6.11.

The code-optimization done within this part is important because it can significantly reduce the generated-code size by elimination of unnecessary code constructions. Furthermore, it is essential for the subsequent back-end part, which requires a consistent and already-optimized code on its input.

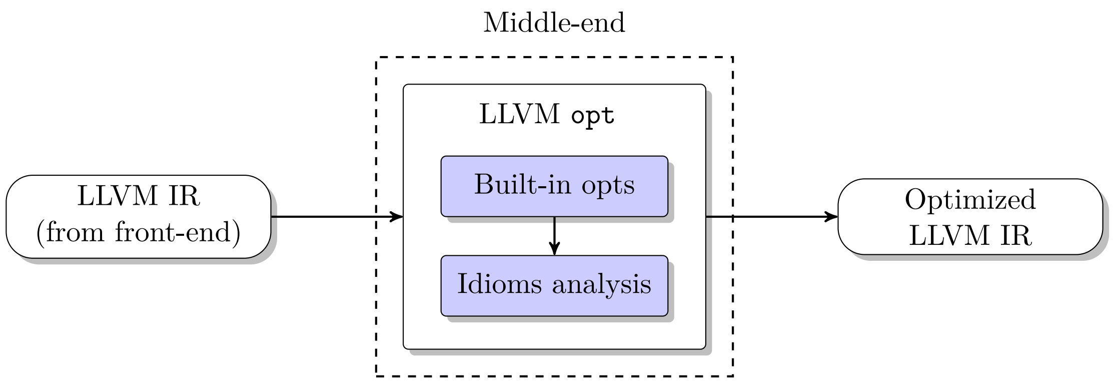


Figure 6.11: A general view on the middle-end part.

As we mentioned in Section 6.2, the middle-end is based on the `opt` tool[18] from the LLVM platform. This tool provides many built-in optimizations and analyses[19]. The `opt` tool is normally used as a LLVM IR optimizer within the LLVM toolchain, but we can use it for optimization of a decompiled code as well. At this point of the decompilation process, the application's representation is the same as during compilation. With some minor modifications, we utilize the `opt` optimization passes that are normally used during compilation with optimization level `-O3` (i.e. even the aggressive optimizations are used).

Furthermore, we describe our own code-transformation pass in Subsection 6.5.1. This pass is responsible for detection and transformation of the instruction idioms within the decompiled code. Its purpose is to detect the hard-to-read code constructions emitted by the original compiler and replace them by a human-readable code.

6.5.1 Instruction-Idiom Analysis

Code de-optimization is one of the necessary transformations used within decompilers. Its task is to properly detect the used optimization and to recover the original HLL code representation from the hard-to-read machine code. One example of this optimization type is the usage of *instruction idioms*. An instruction idiom is a sequence of machine-code instructions representing a small HLL construction (e.g. an arithmetic expression or assignment statement) that is highly-optimized for its execution speed and/or small size [41].

The instructions in such sequences are assembled together by using Boolean algebra, arbitrary-precision arithmetic, floating-point algebra, bitwise operations, etc. Therefore, the meaning of such sequences is usually hard to understand at first sight. A notorious example is the usage of an exclusive or to clear the register content (i.e. `xor reg, reg`) instead of an instruction assigning zero to this register (i.e. `mov reg, 0`).

Instruction Idioms used in Compilers

At present, the modern compilers use dozens of optimization methods for generating fast and small executable files. Different optimizations are used based on the optimization level selected by the user (e.g. `-O0`, `-O1` in GCC compiler).

In Table 6.1, we can see a shortened list of instruction idioms used in common compilers (GCC 4.7.1, Open Watcom 1.9, LLVM/`clang` 3.3, MSVC Compiler 16 and 17, Bor-

[18] http://llvm.org/docs/CommandGuide/opt.html
[19] Complete list: http://llvm.org/docs/Passes.html

land C++ 5.5.1, and Intel C/C++ Compiler XE13). Some of these instruction idioms are widespread among modern compilers. Furthermore, we have found out that actively developed compilers, such as GCC, Visual Studio C++, and Intel C/C++ Compiler, use these optimizations heavily. For example, they generate the `idiv` instruction (fixed signed division) only in rare cases on the Intel x86 architecture. Instead, they generate optimized division by using magic number multiplication.

Table 6.1: Shortened list of instruction idioms found in compilers.

Instruction idiom	**GCC**	**MSVC**	**Intel**	**Watcom**	**Borland**	**LLVM**
Less than zero test	✓	×	✓	×	×	✓
Greater or equal to zero test	✓	×	×	×	×	✓
Bit clear by using `xor`	✓	✓	✓	✓	✓	✓
Bit shift multiplication	✓	✓	✓	✓	✓	✓
Bit shift division	✓	✓	✓	✓	✓	✓
Division by `-2`	✓	×	×	×	×	✓
Expression `-x - 1`	✓	✓	×	×	×	×
Modulo power of two	✓	✓	✓	×	×	✓
Assign `-1` by using `and`	×	✓	✓	×	×	×
Multiplication by an invariant	✓	✓	✓	✓	×	✓
Signed modulo by an invariant	✓	×	✓	×	×	✓
Unsigned modulo by an invariant	✓	✓	✓	×	×	✓
Signed division by an invariant	✓	✓	✓	×	×	✓
Unsigned division by an invariant	✓	✓	✓	×	×	✓
Negation of a float	✓	×	×	×	×	×

Instruction-Idiom Analysis within the Middle-end

In [18], we presented an approach of instruction-idioms transformation within the middle-end part, see Algorithm 1. Every program's basic block is inspected sequentially starting from the beginning. Every instruction within a basic block is treated as a possible root of an abstract syntax tree (AST) containing one particular instruction idiom. If so, the AST is inspected and whenever an instruction idiom is found, it can be easily transformed to its de-optimized form. Since an AST does not depend on the position of instructions in LLVM IR, but rather on the use of instructions, position-dependent problems are solved in this way.

Once a particular idiom is detected, it is substituted into a human-readable form on the level of LLVM IR instructions. An example demonstrating this substitution on the LLVM IR level is shown in Figure 6.12. Figure 6.12a represents the already mentioned `xor` bit-clear instruction idiom. To use register content, a register value has to be loaded into a typed variable `%1`. By using the `xor` instruction, all bits are zeroed and the result (in variable `%2`) can be stored back into the same register. To transform this idiom into its de-optimized form, a proper zero assignment has to be done. This de-optimized LLVM IR code is shown in Figure 6.12b. In this case, the typed variable `%2` holds zero, which can be directly stored in the register. At present, we support detection of all instruction-idioms specified in Table 6.1, among others.

Algorithm 1 Detection of instruction idioms in the middle-end phase.

```
function IdiomInspector(idiom, bb)
    for each instruction in bb do
        FindIdiom(idiom, instruction)                ▷ AST inspection
    end for
end function

for each bb in ProgramBasicBlocks do
    for each idiom in SupportedIdioms do
        IdiomInspector(idiom, bb)
    end for
end for
```

```
%1 = load i32* @regs0
%2 = xor i32 %1, %1
store i32 %2, i32* @regs0
```

(a) Optimized form.

```
store i32 0, i32* @regs0
```

(b) De-optimized form.

Figure 6.12: Example of the bit-clear `xor` idiom transformation (LLVM IR).

6.6 Back-End Phase

The final part of the proposed decompiler is the back-end phase. We do not describe this part in detail as the others because the author had only a minimal contribution in this phase.

The back-end part converts the optimized intermediate representation into the target HLL. Currently, we support two target HLLs: C and a Python-like language. Like the middle-end, the back-end part is also based on the LLVM toolchain. This time, we re-use the `llc` tool[20], which is (in terms of compilation) normally used for transformation of LLVM IR into a target-architecture assembly code. In our case, we only use `llc` methods for LLVM IR handling and the remaining methods are created by us.

The conversion itself is done in several steps. Firstly, the input LLVM IR is read into memory by using the methods available within the `llc` tool. Then, it is converted into another intermediate representation: *back-end intermediate representation*, BIR for short. During this conversion, high-level constructs are identified and reconstructed from the low-level constructs in LLVM IR. Apart from the actual code, we also have to obtain debugging information, which, as described in Section 6.4, is stored in LLVM IR.

After we have BIR with attached debugging information, we run optimizations over it. Even though the input LLVM IR has already been optimized in the middle-end, by converting it into BIR, which is more high-level than LLVM IR, more optimizations are needed. After BIR has been optimized, we rename the program variables to make the code more readable. In this phase, we utilize the debugging information we have obtained earlier. Finally, we use an appropriate generator to emit source code in the selected HLL. Furthermore, the back-end phase is capable of generating a control-flow graph and call graph representation of the decompiled application for its better analysis.

[20]http://llvm.org/docs/CommandGuide/llc.html

Chapter 7

Experimental Results

This chapter contains an evaluation of the proposed methods of dynamic and static retargetable analysis of machine code. In Section 7.1, we focus on performance of two types of translated simulators that were proposed in Section 5.2. Afterwards, in Section 7.2, we discuss the performance achieved by our retargetable source-level debugger described in Section 5.3. Finally, in Section 7.3, we present experimental results related to the retargetable decompiler described in Chapter 6.

The tests were performed on a machine with a following configuration – processor Intel Xeon E5-2630v2 with 16 GB RAM running the Linux-based 64-bit operating system. Our tools were compiled by GCC version 4.9.0 with optimizations enabled (either `-O2` or `-O3`).

MIPS, ARM, PowerPC, Intel x86 architectures were used as the target architectures. All of these architectures are described as instruction-accurate models in the ISAC language. Finally, we briefly describe the test-suite. In the following experiments, we use multiple existing benchmark suites (e.g. SPEC CINT2006 and MiBench), compiler test-suites (e.g. GCC, LLVM), open-source applications (e.g. `sgrep`, `sed`), as well as our own tests.

For compilation of these C/C++ tests, we used two different compiler toolchains – GCC and LLVM. It should be noted that different versions of compilers were used for every particular target platform (i.e. architecture, compiler, and file format). Furthermore, the target file-format was primarily ELF because it is supported for all of these architectures. We also tested by using WinPE executables for ARM and Intel x86 architectures. Finally, all results are the average values of several runs of each test (differences of values from average are usually in tenths of a percent).

The results presented in this chapter are based on the previously published results [18, 19, 28, 29, 39], but they were re-evaluated using a more modern test environment.

7.1 Translated Simulation

Several experiments of translated simulation concept (both static and JIT) were performed. As testing processor architecture we chose MIPS. Benchmarking applications are part of the MiBench test-suite. These applications are used to measure the simulation speeds and simulator creation times. Since benchmark algorithms for self-modifying code are very rare, custom test applications with heavy usage of SMC were created. The analyser and simulators have been complied with the optimization level `-O3`. To avoid confusion, the simulation speed (i.e. performance) is measured in units *million instructions per second*, which are commonly abbreviated as MIPS.

Figure 7.1 shows a comparison of simulation speeds for different simulation types by using MiBench algorithms and custom SMC. We used two different settings of JIT translated simulator, which differ in a threshold value used for a hot-spot detection (simulators are marked as `t=1` and `t=10`).

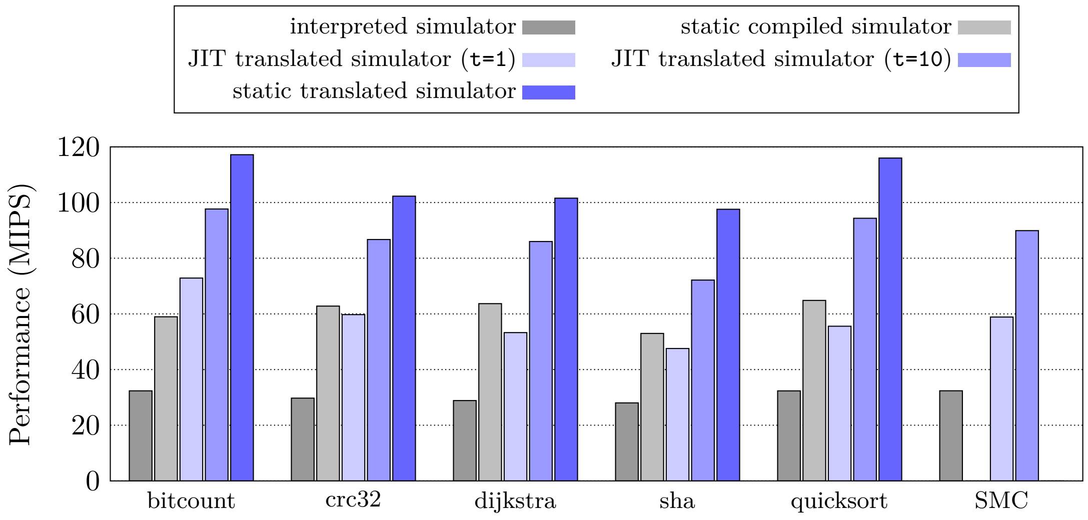


Figure 7.1: Performance comparison of all simulator types (MIPS architecture).

Based on these simulation-speed comparisons, we can make three conclusions. (1) The speed of the static translated simulation is approximately 75% faster than the compiled simulation and more than 3.5x faster than the interpreted simulation. At present, it is the fastest simulation time within the Lissom project, but it cannot simulate SMC code. (2) The speed of the JIT translated simulation varies according to the threshold set by the user and the difference can be up to 90%. Therefore, it is important to try a simulation with different threshold values to find the best one. An automatic selection of this threshold is marked for future research. (3) In our test case, the best threshold was `t=10`, which was almost three times faster than the interpreted simulation, 50% faster than the compiled simulation, and only 20% slower than the static translated simulation.

7.2 Source-Level Debugging

This section contains an evaluation of the retargetable debugger presented in Section 5.3. More precisely, we focus on the speed of the debugged applications, which is one of the most important debugging criteria.

For testing, we used the same applications from the MiBench test-suite as in the previous section, but we aggregated the results as one value this time. These applications were compiled with enabled optimizations (`-O2`) and generation of DWARF debugging information (`-g`). Furthermore, we used the MIPS processor architecture for testing the simulation speeds in four different scenarios of debugging. (1) The first one is used as a reference for measuring the debugging speed drop-off because it represents simulation only, i.e. the debugging information was stripped from the executable file (in order to disable any influence from our DWARF-parsing library) and the particular application is simulated without any instrumentation.

(2) In the second scenario, the debugging information is available, but the debugger simply simulates the application without setting any breakpoints (marked as *0 breakpoints* in the following figures). (3–4) In the remaining two scenarios, we used two different settings of debugging, which differ in the number of used breakpoints (one and five respectively). The address-checking mechanism has been used for the breakpoint implementation, see [19].

We tested an unconditional breakpoints, conditional breakpoints, and even unconditional breakpoints controlled by the number of their execution (marked as *unconditional*, *conditional*, and *controlled* in figures). The breakpoints were set on frequently-executed instructions, such as loops.

Figure 7.2 shows the comparison of simulation speeds for the MIPS architecture.

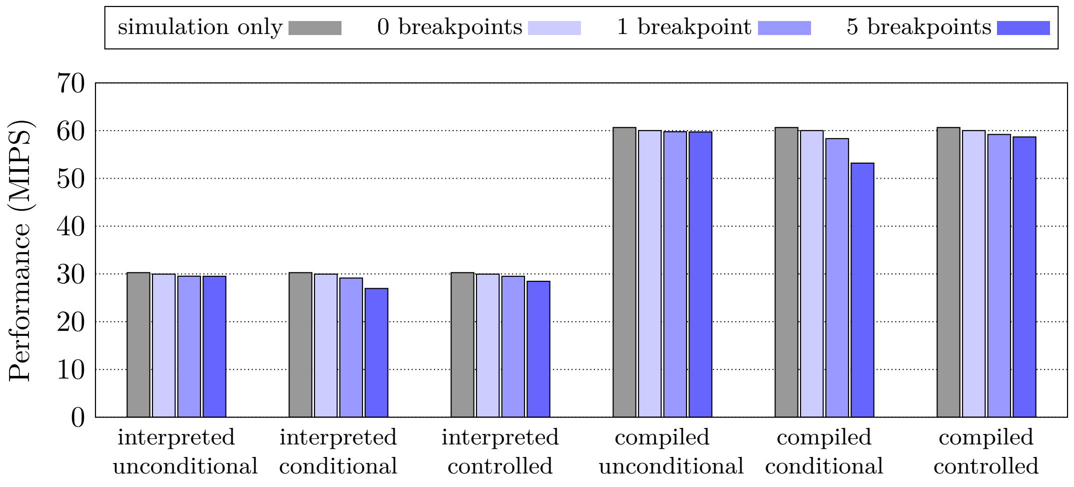


Figure 7.2: Simulator speeds of MIPS processor in different debugging scenarios.

As we can see in these figures, the simulation speed drop-off between scenarios (1) and (2) (i.e. pure simulation and simulation within debugger without any breakpoints) is approximately 1%. Whenever the breakpoins are enabled, in scenarios (3) and (4), we can observe another small performance decrease in comparison to the scenario (2): the difference is about 1% for unconditional breakpoints, 10-14% for conditional breakpoints, and 1-7% for controlled breakpoints. In conclusion, these speed drop-offs are acceptable for a efficient application debugging while using multiple breakpoints (of different types).

7.3 Decompilation

The last section of this chapter is focused on the retargetable-decompiler testing (we use version 1.8 released in September 2014). The primary goal of the aforementioned tools (i.e. simulator and debugger) is to achieve the best performance as a matter of speed. On the other hand, this is not so true in the case of a decompiler because the quality of the decompiled code is far more important in this case. Therefore, we focus on testing of decompiled-code accuracy in this section.

However, measuring the code accuracy is not an easy task because there is no such quality-measuring metric as number of instructions simulated per second in simulation-speed testing. Therefore, we focus on multiple aspects of the decompilation process. First of all, we measure accuracy of the compiler and packer detection within the preprocessing phase. Accurate detection is important for most of the following decompilation analyses (e.g. statically-linked code detection, entry-point analysis, detection of idioms). Afterwards, we present accuracy of the functions and arguments reconstruction. Finally, we focus on testing the instruction-idiom detection and transformation phase. In all of these tests, we compare our results with the best available competitive solutions.

7.3.1 Compiler and Packer Detection

In this part, we focus on evaluation of the previously described methods of compiler and packer detection. The accuracy of our tool (marked as `fileinfo`) is compared with the latest versions of the existing detectors such as PEiD, Exeinfo PE, and DiE. All of these detection tools use the same approach as our solution—detection by using signature matching.

In total, 40 packers (e.g. ASPack, FSG, UPX, PECompact, EXEStealth, MEW) and several of their versions (107 different tools) were used for comparison of the aforementioned detectors. The test set consists of 5,317 executable files in total. We prepared three test cases for evaluation of the proposed solution.

Firstly, we evaluated the detection of a packer's name. This information is critical for the complete decompilation process because if we are unable to detect usage of an executable-file protector, the decompilation results will be highly inaccurate. The results of detection are illustrated in the first data set in Figure 7.3.

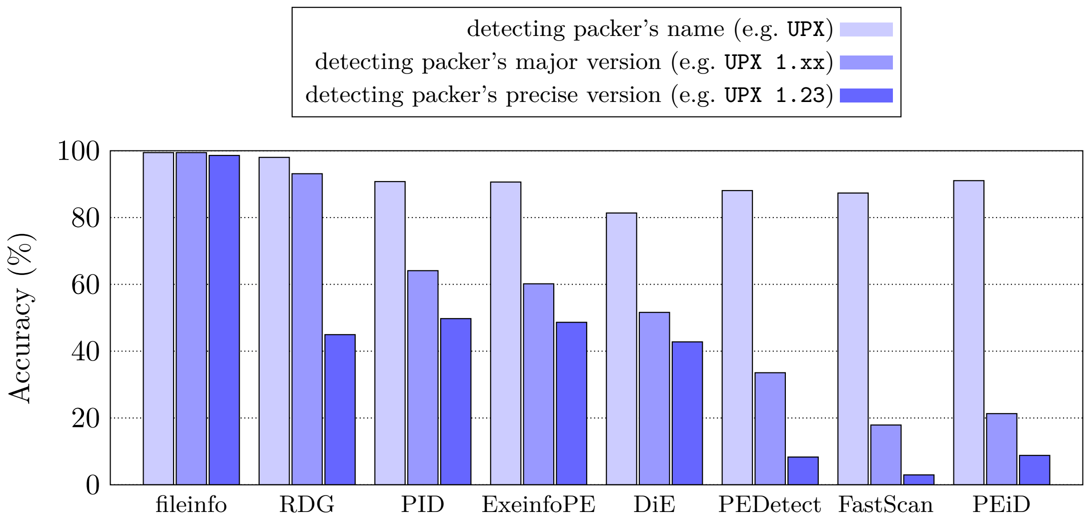


Figure 7.3: Comparison of packer-detection accuracy.

According to the results, our tool has the best ratio of a packer's name detection (over 99%), while the RDG detector was second with ratio 98%. All other solutions achieved comparable results—between 80% and 91%.

Afterwards, we tested the accuracy of a major-version detection. In other words, this test case was focused on a detector's ability to distinguish between two generations of the same tool (e.g. UPX v2 and UPX v3). This feature is useful in the front-end phase during compiler-specific analyses (e.g. loops transformation, jump tables). The results are depicted in the second data set of Figure 7.3. Within this test case, our solution and RDG once again achieved the best results (`fileinfo` scored 99%, RDG scored 93%).

Finally, we tested the ratio of a precise-version detection. This information is crucial for the unpacker because the unpacking algorithms are usually created for one particular packer version and their incorrect usage may lead to decompilation failure. Based on the results depicted in the last data set of Figure 7.3, our detector achieved the best results in this category with 98% accuracy. The results of the other solutions were much lower (50% at most).

7.3.2 Function, Argument, and Return Value Reconstruction

For evaluation of the front-end phase, we measured the accuracy of function, argument, and return value reconstruction. The results of the retargetable decompiler are compared with the modern decompilation "standard"—the Hex-Rays Decompiler (v1.9). The Hex-Rays Decompiler supports the Intel x86, x86-64, and ARM target architectures. Therefore, the direct comparison between these two decompilers can be done only on the ARM and Intel x86 architectures. However, we also tested our solution on the MIPS and PowerPC architectures.

We prepared three test cases for the evaluation of the proposed solution. (1) Firstly, we evaluated the detection and recovery of functions from stripped binaries. (2) Afterwards, to avoid errors found in the previous test case, we tested the accuracy of function-arguments recovery from non-stripped binaries. (3) Finally, we tested the ratio of function return-values detection on the same set of binaries as in the second test case.

Figure 7.4 depicts the overall results of all test cases. The retargetable-decompiler plots contains results of all four currently-supported architectures (MIPS, PowerPC, ARM, Intel x86), while the Hex-Rays Decompiler contain only ARM and Intel x86 results.

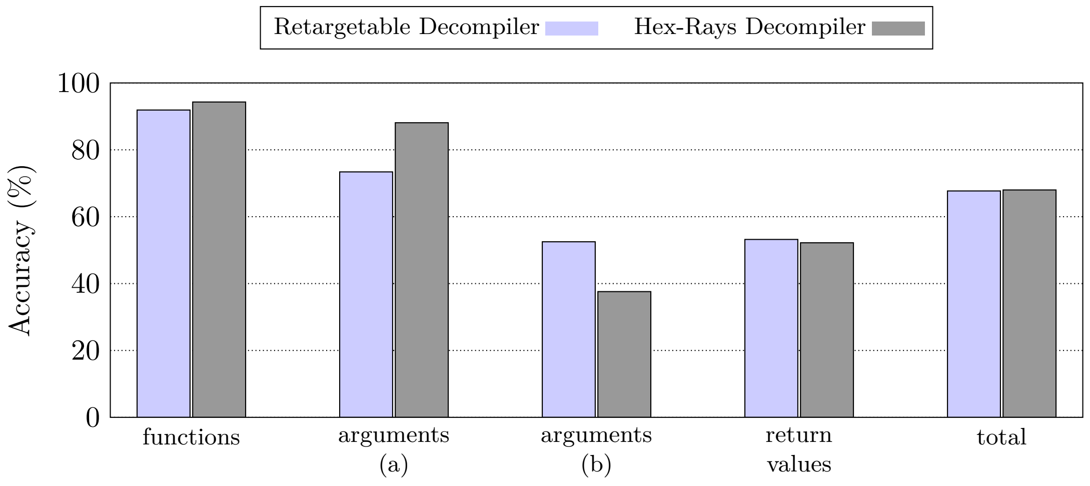


Figure 7.4: Total accuracy of function, argument, and return value reconstruction.

Based on the results, the function detection and return-value detection accuracy is practically the same in both solutions. The Hex-Rays decompiler is more accurate in detection of number of arguments, while the retargetable decompiler is more accurate in detection of their types. Overall, Hex-Rays Decompiler achieved 68.0% accuracy and our solution achieved 67.7% accuracy. This is a very promising result because our solution is fully competitive with the existing commercial off-the-shelf Hex-Rays Decompiler in the matter of function, arguments, and return values recovery. Furthermore, the latest supported architecture, PowerPC, still has some flaws in DFA and without its results, we achieved an overall accuracy of 75.7%.

7.3.3 Instruction-Idiom Detection

Within the last test case of the retargetable decompiler, we focused on the accuracy of instruction-idiom detection and code reconstruction. We compared our solution with the Hex-Rays Decompiler under the same circumstances as described in the previous subsection. A brief comparison of the other decompilers is available in [18].

We created 21 test applications in the C language, where each test is focused on a detection of a different instruction idiom. These C test applications were compiled by the aforementioned compilers into ELF executables for MIPS, PowerPC, ARM, and Intel x86 architectures. We used the ELF file format in each test case, but the same results can be achieved by using the WinPE EFF. All these compilers are based on GNU GCC. The reason for its selection is the fact that it allows retargetable compilation to all target architectures and it also supports most of the idioms specified in Section 6.5.

Different optimization levels were used in each particular test case. Because of different optimization strategies used in compilers, not every combination of source code, compiler, and its optimization level leads to the production of the instruction idiom within the generated executable file. Therefore, we count only the tests that contain instruction idioms.

The decompilation results are depicted in Figure 7.5. We can observe two basic facts based on the results.

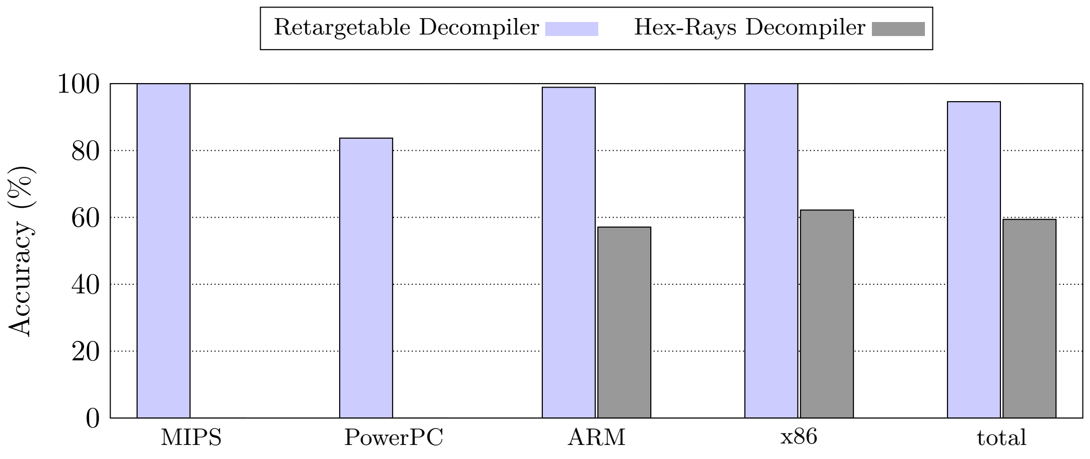


Figure 7.5: Accuracy of instruction-idioms detection and code reconstruction. Note: the total accuracy of Hex-Rays decompiler is calculated based on ARM and Intel x86 only.

(1) The results of the Hex-Rays decompiler on ARM and Intel x86 are very similar (approximately 60%). Its authors covered the most common idioms for both architectures (multiplication via bit shift, division by using magic-number multiplication, etc.). However, the non-traditional idioms are covered only partially or not at all (e.g. integer comparison to zero, floating-point idioms).

(2) Our solution achieved almost perfect results on the MIPS, ARM, and Intel x86 architectures (99.5% in average). Only one test for the ARM architecture failed. Furthermore, our results on PowerPC are not as good as on the other architectures (83.7%). The support of PowerPC decompilation is still in an early phase of development and the PowerPC model in ISAC lacks a support of FPU. Therefore, we cannot handle the floating-point idioms at the moment.

Chapter 8

Conclusion

This thesis was focused on program analysis with special focus on machine-code analysis. This kind of analysis is useful whenever we need to examine the behavior of a particular executable file, e.g. during malware analysis, debugging of optimized code, testing, or re-engineering. As we discussed, there are other existing solutions of machine-code analysis with varying quality and supported features. However, all of them lack a proper support of retargetability, which is a very important feature because of a present day diversity of the used processor architectures. Such non-retargetable solutions support only a limited set of target architectures (file formats, operating systems, etc.) and support of others is either not possible at all, or expensive and time consuming.

The contribution of this thesis is a presentation of several novel approaches of machine-code analysis and their implementation in practice as three tools. We paid special attention to the aforementioned retargetability, which makes our tools usable even on a newly created target architectures. We also focused on two different areas of machine-code analysis — dynamic analysis and static analysis. The first two tools, the translated simulator and source-level debugger, belong to dynamic analysis. The last one, the retargetable decompiler, belongs to static machine-code analysis.

Although their aim is similar, they seek to provide information about behavior of a target application to the user, they differ in the approach of how they achieve this objective. The translated simulator simulates the application's run-time and it can be used for an application's instrumentation. The source-level debugger is also focused on the application's run-time, but it is more oriented to user interaction and investigation. On the other hand, the retargetable decompiler fulfills its objective by static analysis and transformation of a binary code without its execution.

The proposed static translated simulator further improves the previous simulator types and, based on the experimental results, it achieves more than 75% better performance than the previous simulator types developed within the Lissom project. Its drawback is a lack of SMC simulation, but this limitation has been solved by designing the JIT translated simulator that is only 20% slower than its static version. Both of these versions use the debugging information obtained by the newly created library, which supports both PDB and DWARF formats and which is also used within the other two tools.

Furthermore, the retargetable source-level debugger can be primarily used for creating and testing of applications for the newly created architectures. It enhances the existing instruction-level debugger by providing the support of HLL debugging including all the common features, such as different breakpoint types, inspecting variable values, stepping over HLL statements, etc.

Finally, the most important contribution of this thesis is the retargetable decompiler, which is quite unique in comparison with the competitive projects because of its retargetability and high accuracy. Up to now, it supports decompilation of multiple target architectures (MIPS, PIC32, PowerPC, ARM, and Intel x86), EFFs (ELF and WinPE), and it has been tested on several compilers (GCC, LLVM, MSVC, Watcom, Delphi, etc.). We made several tests focused on the accuracy of our solution and according to the experimental results, it can be seen that our concept is fully competitive with other existing tools.

The proposed tools are already used in practice for design and debugging of new ASIPs (within the Lissom project and its spin-off). Furthermore, the retargetable decompiler is used for analysis of real-world malware[21] and user samples via our public web decompilation service located at `http://decompiler.fit.vutbr.cz/decompilation/`.

8.1 Future Work

Despite the achieved results, this topic represents a vast area of research and there are still many open problems. Therefore, we close the thesis by proposing several areas of future research related to the presented tools.

(1) The translated simulator represents a significant step in order to support fast and accurate application simulation. Further improvement of performance can be achieved via a direct binary translation of a simulated application's instructions to a host instructions and their subsequent virtualization. For this task, it may be possible to re-use some of the existing solutions, such as QEMU[22].

Furthermore, the need of debugging information presence may be limiting whenever we would like to simulate applications without it (e.g. release-build applications without debugging information, malware). This problem can be solved (at least partially) by using the retargetable decompiler, which can generate information about basic blocks from its control-flow analysis. In such a scenario, the application will be analysed by the decompiler at first, which will generate an address ranges of all BBs. Afterwards, this information will be used for generation of the translated simulator.

(2) The future research of the retargetable debugger is mainly related to testing on other target architectures because these may reveal some difficulties in controlling the control-flow of the debugged application (e.g. non-traditional function calls or argument passing). Another challenge represents debugging of a C++ code and all of its constructs. The preliminary testing of C++ debugging has already been successfully done, but only with a limited amount of C++ constructs (e.g. no virtual methods, no inherited classes).

Another interesting research topic is an HLL debugging of executable code without debugging information or original HLL source code available. This can be for example used for dynamic analysis of malware. At present, this has to be done on a instruction-level debugging because the sources are obviously unavailable. However, the decompiler can be used once again for this task. Before the main debugging, the application will be fully decompiled and the decompiler will generate some lightweight debugging information usable by the debugger.

[21] `http://now.avg.com/malware-campaign-hits-czech-republic/`,
`http://now.avg.com/turla-rootkit-analysed/`,
`http://now.avg.com/autoit-approach-infecting-browser-code-recycling/`

[22] `http://qemu.org`

(3) There are many more open topics related to future research of the retargetable decompiler. Within its preprocessing phase, it will be very useful to implement a generic unpacker because whenever we are unable to unpack a packed application, the decompilation usually fails in producing accurate code. For this task, it may be possible to employ some of the existing simulators or emulators. It will unpack the application as soon as its run-time unpacking routine is finished and the OEP is hit.

Furthermore, the proposed EFFDL language should be used for modeling of the other EFF types, such as bFLT, U-Boot, etc. The related concept of the context-sensitive parser may also be used in the other areas of computer science, such as compiler construction, see [16].

The decompiler should also be further tested on applications created in other-than-C languages (such as C++, assembler, Delphi, etc), compiled by the non-traditional compilers, or even obfuscated (e.g. malware). These may reveal some non-covered techniques and paradigms. For example, other instruction idioms, function invocations, different data types. The last area is related to another important of future research – reconstruction of more complex data types (e.g. unions, classes), see [22].

Furthermore, it is always possible to add new target architectures (e.g. AArch64, Intel x86-64, SPARC, AVR) or enhance the current ones (e.g. another instruction-set extensions for ARM or Intel x86, FPU for PowerPC) and test the retargetability of the decompiler. The decompiler can also be enhanced by other HLL back-end generators (e.g. C++, Java) and it should be possible to use it for binary-code or source-code migration as we presented in [38].

(4) We already mentioned the interconnection of static and dynamic analysis tools (e.g. usage of a decompiler within simulation or debugging). However, we can go even further via emulating/simulating the decompiled applications in order to obtain some valuable information from their run-time before (or even during) their decompilation. We call this approach *hybrid machine-code analysis* and we have already done some preliminary research in [15] and [12]. As we figured out, the information obtained during emulation (e.g. list of called functions, values of arguments and registers, allocated memory and its usage, and many more) can be quite easily used for producing a more accurate code (e.g. de-obfuscation of code, decryption of strings, reconstruction of composite data types, better handling of switch statements).

This approach is very promising because it can circumvent the limitations of individual static and dynamic analyses.

Bibliography

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, 2nd edition, 2006.

[2] S. Chamberlain. *The Binary File Descriptor Library*. Iuniverse Inc., 2000.

[3] C. Cifuentes. *Reverse Compilation Techniques*. PhD Thesis, School of Computing Science, Queensland University of Technology, Brisbane, QLD, AU, 1994.

[4] K. Dolgova and A. Chernov. *Automatic Type Reconstruction in Disassembled C Programs*. In 15th Working Conference on Reverse Engineering (WCRE'08), pages 202–206, Washington, US-DC, 2008. IEEE Computer Society.

[5] DWARF Debugging Information Committee. *DWARF Debugging Information Format*, 4 edition, 2010. http://www.dwarfstd.org/doc/DWARF4.pdf.

[6] M. J. Eager and Eager Consulting. *Introduction to the DWARF Debugging Format*. Group, 2012. http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf.

[7] M. J. Van Emmerik. *Static Single Assignment for Decompilation*. PhD Thesis, University of Queensland, Brisbane, QLD, AU, 2007.

[8] A. Husár, M. Trmač, J. Hranáč, T. Hruška, K. Masařík, D. Kolář, and Z. Přikryl. *Automatic C Compiler Generation from Architecture Description Language ISAC*. In 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science MEMICS'10, pages 84–91, Brno, CZ, 2010. Masaryk University.

[9] D. Kästner and S. Wilhelm. *Generic Control Flow Reconstruction From Assembly Code*. ACM SIGPLAN Notices, 37(7), 2002.

[10] J. Kinder. *Static Analysis of x86 Executables*. PhD Thesis, Technische Universität Darmstadt, DE, 2010.

[11] D. Kolář and A. Husár. *Output Object File Format for Assembler and Linker*. Internal document, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2012.

[12] J. Končický. *Enhancement of Decompilation by Using Dynamic Code Analysis*. Master's Thesis, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2014. Supervised by: J. Křoustek.

[13] J. Křoustek. *Usage of Decompilation in Processor Architecture Modeling.* In 31th International Autumn Colloquium Advanced Simulation of Systems (ASIS'09), pages 64–67. MARQ, 2009.

[14] J. Křoustek. *Selected Methods of Code Debugging.* Programming Language Theory, Brno University of Technology, Brno, CZ, 2010.

[15] J. Křoustek and D. Kolář. *Approaching Retargetable Static, Dynamic, and Hybrid Executable-Code Analysis.* Acta Informatica Pragensia (AIP), 2(1):18–29, 2013.

[16] J. Křoustek and D. Kolář. *Context Parsing (Not Only) of the Object-File-Format Description Language.* Computer Science and Information Systems (ComSIS), 10(4):1673–1702, 2013.

[17] J. Křoustek, P. Matula, J. Končický, and D. Kolář. *Accurate Retargetable Decompilation Using Additional Debugging Information.* In 6th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE'12), pages 79–84. International Academy, Research, and Industry Association (IARIA), 2012.

[18] J. Křoustek, F. Pokorný, and D. Kolář. *A New Approach to Instruction-Idioms Detection in a Retargetable Decompiler.* Computer Science and Information Systems (ComSIS), 11(4):1337–1359, 2014.

[19] J. Křoustek, Z. Přikryl, D. Kolář, and T. Hruška. *Retargetable Multi-level Debugging in HW/SW Codesign.* In 23rd International Conference on Microelectronics (ICM'11), page 6. Institute of Electrical and Electronics Engineers, 2011.

[20] J. Křoustek, S. Židek, D. Kolář, and A. Meduna. *Scattered Context Grammars with Priority.* International Journal of Advanced Research in Computer Science (IJARCS), 2(4):1–6, 2011.

[21] K. Masařík. *System for Hardware-Software Co-Design.* VUTIUM. Brno University of Technology, Faculty of Information Technology, Brno, CZ, 1st edition, 2008.

[22] P. Matula. *Reconstruction of Data Types for Decompilation.* Master's Thesis, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2013. Supervised by: J. Křoustek.

[23] A. Meduna and J. Techet. *Scattered Context Grammars and their Applications.* WIT Press, Southampton, GB, 2010.

[24] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg, DE, 1999.

[25] G. Nolan. *Decompiling Java.* Apress, Berkeley, US-CA, 2004.

[26] G. Nolan. *Decompiling Android.* Apress, Berkeley, US-CA, 2012.

[27] Z. Přikryl. *Advanced Methods of Microprocessor Simulation.* PhD Thesis, Brno University of Technology, Faculty of Information Technology, 2011.

[28] Z. Přikryl, J. Křoustek, T. Hruška, and D. Kolář. *Fast Just-In-Time Translated Simulation for ASIP Design.* In 14th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDCS'11), pages 279–282. IEEE Computer Society, 2011.

[29] Z. Přikryl, J. Křoustek, T. Hruška, and D. Kolář. *Fast Translated Simulation of ASIPs.* OpenAccess Series in Informatics (OASIcs), 16(1):93–100, 2011.

[30] Z. Přikryl, J. Křoustek, T. Hruška, D. Kolář, K. Masařík, and A. Husár. *Design and Simulation of High Performance Parallel Architectures Using the ISAC Language.* GSTF International Journal on Computing (GTSF IJC), 1(2):97–106, 2011.

[31] Z. Přikryl, K. Masařík, T. Hruška, and A. Husár. *Fast Cycle-Accurate Interpreted Simulation.* In 10th International Workshop on Microprocessor Test and Verification (MTV'09), pages 9–14. IEEE Computer Society, 2009.

[32] J. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture.* John Wiley, New York, US-NY, 1996.

[33] J. Křoustek. *On Decompilation of VLIW Executable Files.* Problems of Programming (ISS), pages 1–8, 2014. Accepted, will be published.

[34] J. Křoustek, P. Matula, D. Kolář, and M. Zavoral. *Advanced Preprocessing of Binary Executable Files and its Usage in Retargetable Decompilation.* International Journal on Advances in Software (IJAS), 7(1):112–122, 2014.

[35] L. Ďurfina, J. Křoustek, P. Matula, and P. Zemek. *Design of Methods for Retargetable Decompiler.* Technical Report, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2013.

[36] L. Ďurfina, J. Křoustek, P. Matula, and P. Zemek. *Implementation of a Retargetable Decompiler.* Technical Report, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2013.

[37] L. Ďurfina, J. Křoustek, P. Matula, and P. Zemek. *A Novel Approach to Online Retargetable Machine-Code Decompilation.* Journal of Network and Innovative Computing (JNIC), 2(1):224–232, 2014.

[38] L. Ďurfina, J. Křoustek, and P. Zemek. *Generic Source Code Migration Using Decompilation.* In 10th Annual Industrial Simulation Conference (ISC'2012), pages 38–42. EUROSIS, 2012.

[39] L. Ďurfina, J. Křoustek, P. Zemek, and B. Kábele. *Detection and Recovery of Functions and Their Arguments in a Retargetable Decompiler.* In 19th Working Conference on Reverse Engineering (WCRE'12), pages 51–60, Kingston, ON, CA, 2012. IEEE Computer Society.

[40] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna. *Design of a Retargetable Decompiler for a Static Platform-Independent Malware Analysis.* International Journal of Security and Its Applications (IJSIA), 5(4):91–106, 2011.

[41] H. S. Warren. *Hacker's Delight.* Addison-Wesley, Boston, US-MA, 2003.

[42] J. Zhang, R. Zhao, and J. Pang. *Parameter and Return-value Analysis of Binary Executables.* In 31st Annual International Computer Software and Applications Conference (COMPSAC'07), volume 1, pages 501–508, Washington, US-DC, 2007. IEEE Computer Society.