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

Retargetable Analysis of Machine Code

Statement of Originality

I hereby declare that this thesis is my own work that has been created under the supervision of doc. Dušan Kolář. Several results were achieved in co-operation with Lukáš Ďurfina, Petr Zemek, Peter Matula, Zdeněk Přikryl, Stanislav Židek, my students, and other members of the Lissom project. Where other sources of information have been used, they have been duly acknowledged.

....
Jakub Křoustek
November 17, 2014

Acknowledgements

I wish to thank Dušan Kolář for the support during his supervision of this work. I also wish to thank all members of the Lissom research project for their help and ceaseless support of my research. Last, but certainly not least, I wish to thank my family and my girlfriend Lenka for their support and patience during my work on this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Truth can only be found in one place: the code. Only the code can truly tell you what it does."

Robert C. Martin

Program analysis is a computer-science methodology, whose task is to analyse the behavior of a given program [62]. 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 [56, 73, 90], program slicing [97], or abstract interpretation [95]. 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

[1]The term *debugging* denotes "methodology of finding, isolating, and removing defects (bugs) from software programs", according to [77].

[2]There are also other forms of code representation (e.g. object code); however, those are rarely used for program analysis [82].

bytecode [63, 64], 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 [63] 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 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, 96]. 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 [13, 20, 80]. 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 [41] 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 Author's Contribution

In particular, the main author's contribution is as follows:

- *ISAC language extensions and architectures modeling* – In order to support retargetability of the designed tools, the ISAC architecture-description language (ADL) [52] is used for description of target architectures. Afterwards, the ISAC models are used during automatic generation of tools. However, it was necessary to make several enhancements of ISAC language constructions. For this reason, the author added constructions for a description of instruction decoders, modeling of multiple instruction sets, support of less common architectures (e.g. VLIW and its instruction encoding and compression) [121]. Furthermore, the author created (or enhanced) several ISAC models of existing architectures (e.g. VEX, MIPS, ARM).

- *Exploitation of debugging information* – Because of a massive lack of information stored in analysed executable files, it is necessary to use every source of available information. One of such sources is debugging information. The author designed an approach for detection, extraction, and representation of debugging information. At present, the two most common formats are supported — DWARF and PDB. He also utilized this approach in the following tools. Debugging information is used within the translated simulator for detection of basic blocks [117]. The retargetable debugger uses this information for source-level debugging [113] and the retargetable decompiler exploits it for generation of a more readable code [109]. Furthermore, the extracted information about functions is also used within the profiler for the C language [72]; however, this tool has been created without the author's contribution.

- *Static Translated Simulator and Just-In-Time Translated Simulator* – The author participated in design, implementation, and testing of these tools for a high-speed simulation of applications for target architectures. This work extends the previous work of Zdeněk Přikryl in the field of interpreted [71] and compiled simulators [70]. The newly designed methods lack a simulation of processor microarchitecture; on the other hand, they provide higher simulation speeds. Two versions of this simulator have been designed — *static* [119] and *just-in-time* translated simulator [118].

- *Retargetable Source-level Debugger* – One of the most common tools for machine-code analysis is a debugger. This tool can basically operate on two levels of abstraction — *instruction-level* and *source-level*. The former one is more low-level and it operates on a level of machine instructions, registers, memory cells, etc. Basically, it is only able to investigate the state of the hardware resources during debugging, but it is

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

mostly unusable for analysis of the original application on its source level. This type of debugger has been already done by other members of the Lissom project. The author created the second level (i.e. support of source-level debugging) on the top of the existing debugger. The source-level debugger supports the retargetable analysis on the source-level with all the previously mentioned features [120]. This extension is used for debugging of applications generated by the Lissom C compiler. The author also contributed with modifications of the compiler toolchain to allow generation of a proper debugging information.

- *Retargetable Decompiler* – The last tool mentioned in this thesis is a retargetable decompiler. It serves for a static, platform-independent analysis of machine code. Its complexity is much higher than in the previous tools. For this reason, the author designed this tool as a complete toolchain containing multiple applications that are interconnected via scripts [102, 103]. We can divide these applications in the following phases — preprocessing, front-end, middle-end, and back-end. The author participated in design and implementation of most of the methods used within these phases, e.g. detection of originally used compiler/packer [108], unpacking [123], executable-file format conversion [105, 110] (preprocessing phase), exploitation of debugging information [109], instruction decoding [132–134], control-flow and data-flow analysis [131, 135], function detection [128–130] (front-end phase), and instruction-idiom reconstruction [111, 112] (middle-end phase). The author also contributed during testing of this tool on real-world malware samples [124, 126, 127]. Finally, the author also defined a new formalism *scattered context grammar with priority function* [114, 115] (and proved its generative power [116]) that can be used for uniform parsing of input binary executable files within the preprocessing phase [107].

  Contrariwise, the author has only a very limited contribution in the design of methods used within the back-end phase and no contribution in their implementation as well as in several other methods used within the front-end phase, e.g. elimination of statically-linked code, data-type recovery, and detection of similarity between two files.

1.2 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. Finally, we present the capabilities of the proposed retargetable decompiler on several examples in Appendix A as well as a case study of malware decompilation by using this tool.

Chapter 2

Preliminaries

"*In theory, there is no difference between theory and practice. In practice, there is.*"

Johannes L. A. van de Snepscheut

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, such as grammars, languages, and their generative power. 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 [27, 55]. 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, 60] for further reference. This chapter is based on [35, 54, 69, 78, 79].

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*). We set $\mathrm{lhs}(p) = x$ and $\mathrm{rhs}(p) = y$, which represents the *left-hand side* and the *right-hand side* of the production p, respectively. If $\mathrm{rhs}(p) = \varepsilon$, p is called *erasing production* (ε*-rule*).

Definition 3. Let $G = (V, T, P, S)$ be a phrase-structure grammar, $y = w_1\alpha w_2$, $z = w_1\beta w_2$, $y, z \in V^*$, $p = \alpha \to \beta \in P$. Then y *directly derives* z in the G according to the production p, $y \Rightarrow_G z$ $[p]$ (or simply $y \Rightarrow_G z$). Let $\Rightarrow_G^+$ and $\Rightarrow_G^*$ denote the *transitive* and the *reflexive-transitive closure* of $\Rightarrow_G$, respectively. To express that G makes the *derivation* from u to v by using the sequence of productions $p_1, p_2, \dots, p_n \in P$, we write $u \Rightarrow_G^* v$ $[p_1 p_2 \dots p_n]$ (or $u \Rightarrow_G^+ v$ $[p_1 p_2 \dots p_n]$ to emphasize that the sequence is non-empty).

Definition 4. The *language* generated by G is denoted by $L(G)$ and defined as $L(G) = \{w : w \in T^*, S \Rightarrow_G^* w\}$. A *recursively-enumerable language* (RE language) is a language generated by a phrase-structure grammar. The family of recursively-enumerable languages is denoted by $\mathcal{L}(\mathrm{RE})$.

Definition 5. A *context-sensitive grammar* (CS grammar) is a phrase-structure grammar $G = (V, T, P, S)$, such that every production $p = x \to y \in P$ satisfies $|x| \le |y|$.

Definition 6. A *context-sensitive language* (CS language) is language generated by a CS grammar. The family of context-sensitive languages is denoted by $\mathcal{L}(\mathrm{CS})$.

Definition 7. 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 8. A *context-free language* (CF language) is a language generated by a context-free grammar. The family of context-free languages is denoted by $\mathcal{L}(\mathrm{CF})$.

Definition 9. A *scattered context grammar* (SCG, see [27]) 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 $(A_1, \dots, A_n) \to (x_1, \dots, x_n)$, where $A_i \in V - T$, $x_i \in V^*$ for all $i : 1 \le i \le n$.

Definition 10. A *propagating scattered context grammar* (PSCG) is a SCG $G = (V, T, P, S)$, in which every $(A_1, \dots, A_n) \to (x_1, \dots, x_n) \in P$ satisfies $x_i \in V^+$ for all $i : 1 \le i \le n$.

Definition 11. Let $G = (V, T, P, S)$ be a (propagating) SCG. If

$$\begin{aligned} y &= u_1 A_1 u_2 \dots u_n A_n u_{n+1}, \\ z &= u_1 x_1 u_2 \dots u_n x_n u_{n+1}, \end{aligned}$$

and $y, z \in V^*$, $p = (A_1, \dots, A_n) \to (x_1, \dots, x_n) \in P$, then y directly derives z in the SCG G according to the production p, $y \Rightarrow_G z$ $[p]$ (or simply $y \Rightarrow_G z$).

Definition 12. The *(propagating) scattered context language* is a language generated by a (propagating) SCG. The family of (propagating) scattered context languages is denoted by $\mathcal{L}((\mathrm{P})\mathrm{SC})$.

We can see that an application of a scattered context production simulates application of several context free productions in parallel. Furthermore, scattered context grammars have another advantage over other grammar types (e.g. CS grammars) — they preserve CF grammar ability to be effectively analysed and parsed [44].

Theorem 1. *Generative power* of the previously defined language families can described as (see [55]):

$$\mathcal{L}(\text{CF}) \subset \mathcal{L}(\text{PSC}) \subseteq \mathcal{L}(\text{CS}) \subset \mathcal{L}(\text{RE}) = \mathcal{L}(\text{SC}).$$

The relation $\mathcal{L}(\text{PSC}) \subseteq \mathcal{L}(\text{CS})$ is still an open problem.

Definition 13. An *extended lazy finite automaton* is a 7-tuple $M = (Q, \Sigma, R, s, S, F, z)$, where (see [35, 69])

- Q is a finite set of states;
- Σ is an input alphabet;
- R is a finite set of transitions in the form $pa \to q$, where $p, q \in Q$ and $a \in \Sigma^*$;
- $s \in Q$ is the start state;
- S is a set of *semantic actions*;
- $F \subseteq Q$ is a set of final states;
- z is a relation $z \subseteq R \times S$.

The relation z assigns the semantic actions to the particular transitions. The semantic action is an indivisible sequence of a C code, which is executed when a particular transition is taken. The further details about this formalism and its usage are described in Chapter 5. Furthermore, definitions of configuration, move, and accepted language are analogical to definitions related to a standard *lazy finite automaton*, see [35].

Definition 14. A *two-way coupled finite automaton* (see [35]) 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 an extended lazy finite automaton, $i = 1, 2$;
- h is a bijective mapping from R_1 to R_2.

  Let h^* be a mapping from R_1^* to R_2^* and

  - $h^*(\epsilon) = \{\epsilon\}$;
  - for every $r_1, r_2, \dots, r_n \in R_1$ applies that $h^*(r_1, r_2, \dots, r_n) = h(r_1)(r_2)\dots(r_n)$ for all $n \geq 1$.

In other words, the automata M_1 and M_2 cooperate via the bijective mapping h from R_1 to R_2 and vice versa. The further details about this formalism and its usage are described in Chapter 5.

2.2 Theory of Compiler Construction

Within this section, we define the basic terminology used within the compiler construction theory. Most of these terms are based on the graph theory; therefore, the basics of this area are reviewed at first. This section is based on [1, 6, 68].

Definition 15. A *directed graph* (or *digraph*) is a pair, $G = (N, E)$, where

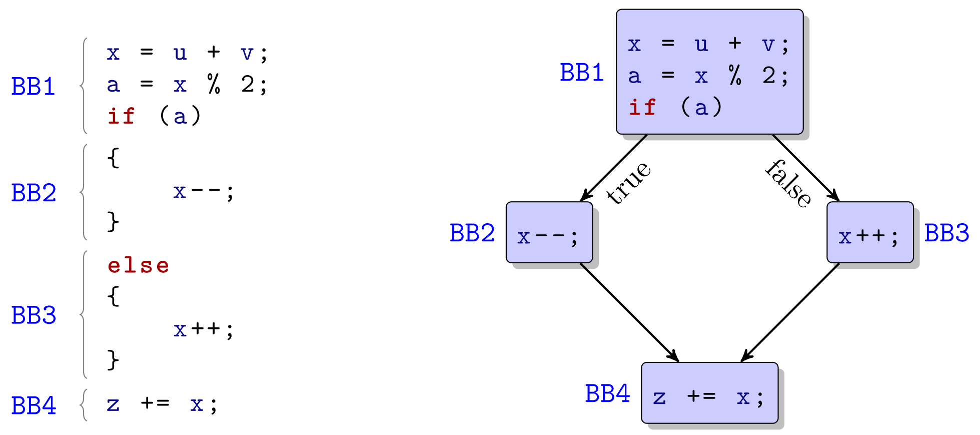
- N is a non-empty finite set of elements called *nodes* (*vertices*);
- $E \subseteq N \times N$ is a finite set of ordered pairs of nodes called *edges* (*arcs*).

If $(u, v) \in E$ then u is *adjacent to* v and v is *adjacent from* u. A *predecessor* of a node v is a node adjacent to v, and a *successor* of v is a node adjacent from v. The *indegree* of a node v is the number of predecessors of v, and the *outdegree* of v is the number of *successors* of v.

A *walk* W in G is a sequence of nodes $v_0 v_1 \dots v_n$ such that $v_i \in N$ and $(v_{i-1}, v_i) \in E$ for $1 \le i \le n$. The length of a walk W is the number of edges in W and it is as denoted $|W|$. A walk with $v_0 = v_n$ is called *cycle*.

Definition 16. *Basic block* (BB) [15] 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.

Example of source-code partitioning into basic blocks is depicted in Figure 2.1a.


```
BB1 { x = u + v;
      a = x % 2;
      if (a)
BB2 { {
          x--;
      }
BB3 { else
      {
          x++;
      }
BB4 { z += x;
```

(a) C source code divided into basic blocks. (b) Control-flow graph of the source code.

Figure 2.1: Illustration of basic blocks and control-flow graph for a simple source code.

Definition 17. *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).

[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).

Nodes not occurring in any walk from v_I to v_E are called *unreachable*. An example of a control-flow graph is illustrated in Figure 2.1b.

Definition 18. *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 of the program reaches all procedures. An example of a call graph is illustrated in Figure 2.2.

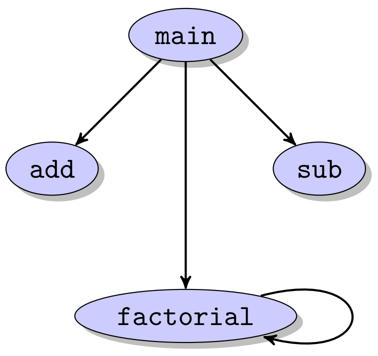
```
int add(int a, int b) {
    return a + b;
}

int sub(int a, int b) {
    return a - b;
}

int factorial(int n) {
    if (n == 0)
        return 1;
    return n * factorial(n - 1);
}

int main() {
    int x = add(4, 5);
    int y = sub(x, 1);
    return factorial(y);
}
```

(a) C source code consisting of four functions.



(b) Call graph for a given source code.

Figure 2.2: Example of a call graph.

Definition 19. The *definition-use graph* (def-use chain, or DU chain) [40] for a program P is a directed graph $G = (N, E)$ that satisfies the following conditions:

1. Each node $n \in N$ represents one statement of P; each statement may *define* or *use* one or more *variables*;

[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.

2. Each edge $(u, v) \in E$ represents a relation between a variable's definition in u and its usage in v during processing of P. The use of the variable (in v) must be reachable from its definition (in u) without any intervening re-definitions.

Definition-use chains are used mainly in compilers to get from a definition of a variable to all locations that could use that defined value.

Definition 20. The *use-definition graph* (use-def chain, or UD chain) is a counterpart of the definition-use graph. Each of its edges $(u, v) \in E$ represents a relation between a statement u with variable's usage and statement v that may have defined the variable last prior to u.

An example of def-use and use-def chains for a variable `x` listed in Figure 2.1 is depicted in Figure 2.3.

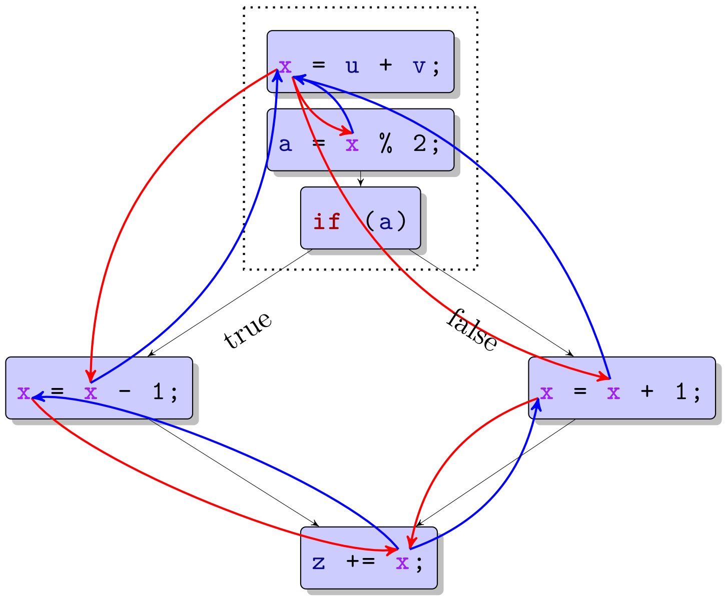


Figure 2.3: Example of def-use chains (red) and use-def chains (blue) for variable `x`.

Definition 21. *Static Single-Assignment Form* (SSA) is a form of code representation, which guarantees that each variable is assigned exactly once. This is the major difference to classical three-address code representation [1]. A short comparison of these two forms is depicted in Figure 2.4. As we can see, the original variable `a` has to be redefined as a new variable (e.g. `a1`, `a2`) each time we reassign the original variable.

```
a = u + v;
b = a - w;
a = b * x;
a = a * y;
```

(a) Three-address code.

```
a  = u  + v;
b  = a  - w;
a1 = b  * x;
a2 = a1 * y;
```

(b) Static single-assignment form.

Figure 2.4: Example of different types of intermediate-code representation.

Storage of intermediate code representation (IR) in the SSA form is common for compilers (e.g. LLVM[8], GCC[9]) and decompilers (e.g. Boomerang). In the SSA form, use-def chains are explicit and each contains a single element. The major advantage of SSA is its positive impact on optimizations and transformations, such as expression propagation, data flow analysis (DFA), etc. For more details about SSA and its usage in decompilation, see [21].

As will be described in Chapter 6, basic blocks, control-flow graphs, and call graphs are fundamental parts of a *control-flow analysis* (CFA) used within the proposed retargetable decompiler. Furthermore, a *data-flow analysis* of this decompiler is based on use-def chains, def-use chains, and SSA form.

[8]`http://llvm.org/`
[9]`http://gcc.gnu.org/`

Chapter 3

Focus of the Thesis

"The x86 really isn't all that complex – it just doesn't make a lot of sense."

Mike Johnson (AMD)

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. Methods for analysis of these files are already well described, see [63, 64].

In the following chapters, we often reference existing architectures and standards. For this reason, we briefly describe the most common architectures in Section 3.1, executable-file formats (EFF) in Section 3.2, and debugging-information standards in Section 3.3.

This chapter is based on [104, 109, 110].

3.1 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. The following classification is used to refer to different instruction sets in the latter chapters.

We can find three main groups of CPU architectures [67]:

- *CISC* – Complex Instruction Set Computer;
- *RISC* – Reduced Instruction Set Computer;
- *VLIW* – Very Long Instruction Word.

CISC, RISC, and VLIW instructions are illustrated in Figure 3.1. A brief description of each architecture type together with examples of real CPUs follow.

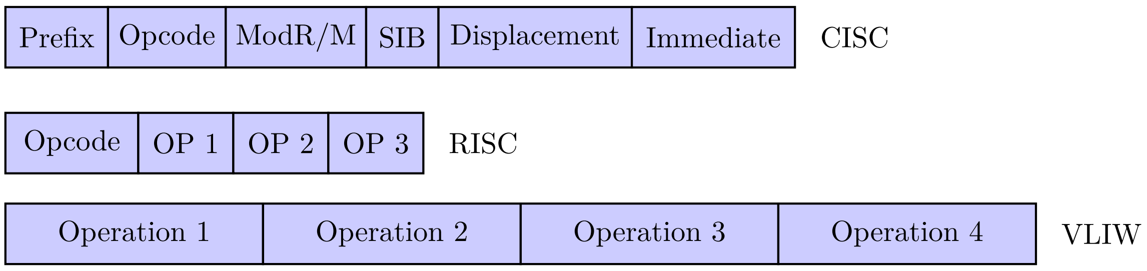


Figure 3.1: Example of CISC, RISC, and VLIW instructions.

3.1.1 CISC Architecture

This architecture type is typical because of its broad instruction set. It is not rare that CISC architecture contains hundreds or even thousands of highly-specialized instructions. This is mainly because the instruction set contains multiple instructions for one particular operation (e.g. move, addition, branch) that differ based on the used operand size (e.g. move byte, move double-word) or type (e.g. move byte from memory, move byte from register). Each such operation has a different *opcode* (i.e. operation code). This approach is programmer-oriented because it makes programming easier (i.e. there is a good chance that there exists a specialized instruction for each HLL construct). CISC compilers have good flexibility in choosing instructions for a given task, but not all of them can fully utilize a processor's instruction set.

The draw-back of CISC is the complexity of hardware architecture. Some of the most complex parts of CISC hardware are instruction fetch and instruction decoding units because they must handle all CISC instructions that have non-uniform length and format. After fetching and decoding, some CISC architectures internally convert complex CISC instructions into simpler RISC-like microcode (see below) that is easier to execute. Furthermore, the CISC architecture usually contains only a limited number of registers (e.g. 8-16).

Examples of CISC instruction-set architectures are Intel x86, Intel 8051, and Motorola 68k. The first one is a standard of present day general-purpose computing and we will refer to it multiple times. It contains four main registers (`eax`, `ebx`, `ecx`, `edx`), four index registers (`esi`, `edi`, `ebp`, `esp`), one program counter (`eip`), and few others. Its instruction set (also referred as IA-32) contains a large number (over 300) of different instruction classes[10] [38].

3.1.2 RISC Architecture

In comparison to CISC, RISC simplifies the processor by only implementing instructions that are frequently used in programs. Such processors contain only a few tens of instructions (e.g. load, add, branch) with a strict meaning of their behavior (e.g. the target address of a branch instruction is always taken from a special register). Therefore, programming such CPUs is a lot harder because every uncommon construction has to be rewritten via multiple RISC instructions. This leads to larger programs than on CISC and the compilers have to be more complex.

On the other hand, all RISC instructions for one particular architecture have the same length (e.g. 16-bit, 32-bit) and the decoding phase is simple. Furthermore, it is not necessary

[10] An *instruction class* corresponds to an operation (e.g. move, branch, add). However, there may be dozens of different opcodes for each class, specifying different sizes or types of operands

to convert instructions into microcode. RISC processors do not use specified registers like on CISC; they use large register files instead (e.g. an array of 32 or even 64 registers).

Examples of RISC instruction-set architectures are MIPS [58], ARM [3], or PowerPC [37]. All of them are 32-bit architectures[11] and all are mentioned in the following text. MIPS and PowerPC contain one register field with 32 32-bit registers for use by integer and logic instructions and a second register field with 32 registers for floating-point instructions. MIPS uses the first integer register `$0` (`$zero`) as a constant zero (i.e. write into this register has no effect and read from it always returns zero). The register field for integer instructions on ARM has a size of 16; the last register `R15` acts as a program pointer. Furthermore, every MIPS and PowerPC instruction is 32-bit long. However, this is not always true on ARM, where we can find multiple instruction sets that are switched during run-time. For example, the ARM-Thumb instruction set contains 16-bit instructions. There are also other instruction sets for ARM, e.g. VFP, SIMD, Neon.

3.1.3 VLIW Architecture

The first reference about the VLIW processor architecture dates back to 1983 [25]. Since this time, VLIW processors are characterized by explicit instruction level parallelism (ILP). Each VLIW instruction specifies a set of operations that are executed in parallel. Each of these operations (also known as *syllables*) is issued and executed simultaneously with others. VLIW operations are minimal units of execution and are similar to RISC instructions [25].

The performance speed-up (against RISC and CISC) is achieved via scheduling of a program execution at compilation time. Therefore, there is no need for run-time control mechanisms and hardware can be relatively simple. On the other hand, all constraints checks must be done by the compiler during compilation. Whenever the compiler is unable to fully utilize all operation slots, it must fill the gap with the `nop` (i.e. No OPeration) operation. This may lead to a rapid performance decrease because the instruction cache will be full of inefficient `nop` instructions. Therefore, all the major VLIW processors use some kind of instruction encoding (i.e. compression). It basically packs each instruction into a so-called *bundle* that is smaller in size because the compression removes the `nop` instructions.

From the micro-architectural point of view, VLIW processors consist of large register files and specialized functional units (adder, multiplier, unit for memory access, etc.) [115]. Those units are managed by operations. Therefore, this architecture contains several different decoders, while it usually contains only one fetch unit for fetching the whole long instruction words.

Most of the VLIW processors are used in digital-signal processing [22], e.g. SHARC by Analog Devices, the C6x family by Texas Instruments (TI), ST2xx family from STMicroelectronics. The most well-known example is Itanium IA-64 by Intel. In the later text, we focus on the VEX architecture. VEX is a 32-bit VLIW (Very Long Instruction Word) processor with four slots designed by Hewlett-Packard [25]. Each slot is unique (i.e. each slot operates with different set of operations) and the maximal instruction length is 128 bits.

[11]There are also 64-bit versions of these architecture — MIPS64, AArch64, and ppc64. However, these are yet not so common as their predecessors and we will consider only the 32-bit version in this thesis.

3.2 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, also called Object-File Formats, or OFF in short) 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 following parts [51]:

- *Header* – Contains essential information about the file, such as:
  - *Magic number* – Sequence of unique bytes for format identification;
  - *Type* – Executable, library, or object code;
  - *Target machine information* – Specification of architecture (e.g. MIPS, ARM, x86), endianness (little, big, or mixed), bitwidth, etc.;
  - *Entry-point (EP) address* – Address of the first machine-code instruction that will be executed[12];
  - *Others* – Size and pointer to each section and symbol table, etc.
- *Object code* – The most important information, which each format holds is code and data. It is stored in the so-called *sections* in most of the formats. The section with the machine code is often named `.text`. Application data (e.g. variables, constants) is usually stored in multiple sections — e.g. `.rodata` for constants (read-only data), `.bss` for uninitialized data, `.data` for others (e.g. global variables);
- *Relocations* – "*Relocation is the process of assigning load addresses to the various parts of the program, adjusting the code and data in the program to reflect the assigned addresses*" [51]. We can find a wide range of relocation types for each target architecture. Some relocations can be resolved during compilation by linker; while the other ones have to be resolved by a loader before a program's execution;
- *Symbols* – Symbols are usually stored in tables and they characterize EFF's local, imported, or exported symbols (variables, functions, etc.);
- *Debugging information* – Generated by compilers for debug support. There exist several debugging information standards, see Section 3.3. The presence of the debugging information is optional.

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 [86] and WinPE [57]. However, other formats are on the rise (e.g. Apple's OS-X Mach-O), see [110]. We briefly describe their structure in the following subsections.

3.2.1 UNIX Executable and Linking Format

The *UNIX ELF* [86] file format is a standard on all modern UNIX-like systems. It replaced the previous `a.out` file format. ELF is independent of a particular target architecture (e.g. MIPS, ARM, x86) and it is very extensive. The leading part of an ELF file is a header

[12]We can find techniques bypassing this concept, such as *thread-local storage (TLS) callback* described in http://www.hexblog.com/?p=9.

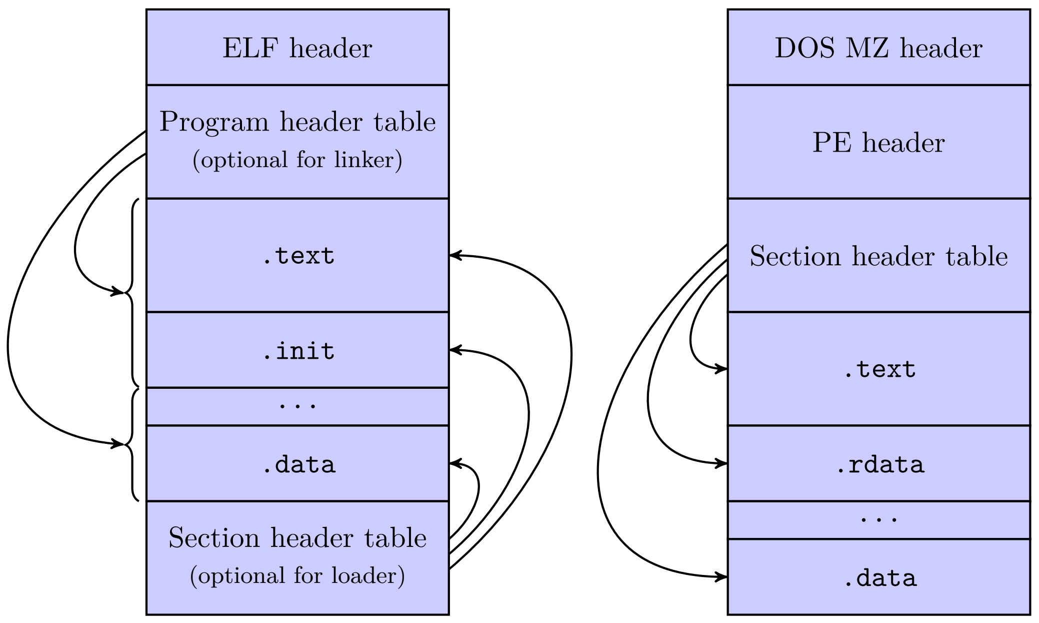
with all the essential information. It also points to the program and section header tables. These tables contain information about particular segments and sections, respectively (e.g. their sizes, offsets within the file). Each section can store different content (e.g. code, data, symbol, hash tables); furthermore, one or more sections may form a segment. ELF also supports dynamic symbol tables, relocation tables, hash tables, etc.

From the linker point of view, an ELF file consists of a group of sections defined in a section-header table. Contrariwise, loader handles an ELF file as a group of segments defined in a program-header table. The very important characteristic of this format is its flexibility. Only the header has a fixed offset within the file, all other elements are optional, as well as their offsets within the file. Therefore, all elements are scattered throughout the file, and the size or content of padding is unspecified. A simplified structure of the ELF format is depicted in Figure 3.2a.

3.2.2 Microsoft Portable Executable and Common Object File Format

The *Microsoft Portable Executable and Common Object File Format* (abbreviated as PE or WinPE) [57] also supports all of the three file types — object files, executables, and libraries. WinPE can be used on all Windows-based systems on architectures Intel x86, x86-64, IA-64, and others.

The structure of the WinPE format is based on the *Common Object File Format* (COFF) [26]. It consists of a number of headers and sections that tell the loader how to map the file into memory. Each section has its own header and often a specific purpose. For example, the `.text` section holds the program code; `.data` section holds initialised global variables, `.edata` and `.idata` sections contain exported and imported symbols, etc. A brief overview of the PE structure is illustrated in Figure 3.2b.



(a) Structure of the ELF format. (b) Structure of the WinPE format.

Figure 3.2: Structure of the ELF and WinPE executable file formats.

3.2.3 Other Common Formats

The **Mach-O** object file format [2] is used in operation systems Darwin, NeXTSTEP, OS X, or iOS from Apple Inc. It is made up of three parts — Mach-O header, followed by a series of load commands, and one or more segments, each of them containing up to 255 sections. Mach-O supports Intel x86, x86-64, PowerPC, and PowerPC64 as the target architectures.

A special feature of this format is its support of multi-architecture binaries, where multiple Mach-O files can be combined in a single multi-architecture file. Such binary file contains code for multiple instruction-set architectures.

The **E32Image** format is used on the Symbian operation systems, usually used in smartphones[13]. It was developed by Symbian Ltd., which currently belongs to Nokia. E32Image is used only on the ARM architecture.

E32Image is a proprietary format, and its specification has never been publicly released; therefore, all the necessary information was gathered using reverse engineering. This format was originally based on WinPE, but since Symbian version 9.1 (in 2005), its authors switched to an ELF-like format. The E32Image file is created from an existing PE or ELF executable file by a special post-linker. The main idea of this format is to provide a basic EFF structure with a low-memory overhead. In comparison with the aforementioned formats, the E32Image format merges and compresses the same types of sections, it does not explicitly encode information about the target architecture (e.g. word size), and it also removes the unnecessary strings (e.g. symbol names).

For comparison of these formats, we also briefly describe one of the most common byte-code formats — **Android DEX** (Dalvik EXecutable). DEX is used by Google in its Android OS running on smart devices. The Android operating system is based on the Linux kernel, which runs the Dalvik virtual machine that interprets DEX applications[14].

A DEX file is built by converting Java bytecode into the Dalvik internal bytecode by a tool called `dx`. In comparison with previous formats, DEX is a very specific file format. Its code is not supposed to be executed by a real processor, but rather interpreted via an abstract virtual machine. This is the reason why DEX does not contain sections, symbols, or relocations. The structure of DEX is more similar to HLL constructions like classes, data types, and prototypes. The interpretation is done by executing methods from these classes.

3.2.4 Comparison of EFFs

A brief comparison of these file formats is depicted in Table 3.1. Only the ELF32Image is a proprietary format, while others are open. The classical structure described at the beginning of this section (e.g. header, section tables, sections) is used by all formats, except DEX. As has been previously said, its structure is very different to other formats; therefore, all other compared criteria are not met by this format. The Mach-O format has only a limited number of sections (max. 256) and the `a.out`[15] format is limited even more (maximally 7 sections with the predefined meaning). The E32Image format is limited to ARM architecture in the same way as DEX to Dalvik VM. Therefore, these two formats together with `a.out` do not support explicit declaration of the target processor architecture. Finally, only the

[13] `http://www.developer.nokia.com/Community/Wiki/E32Image`

[14] `http://www.retrodev.com/android/dexformat.html`

[15] More details can be found in `a.out` FreeBSD Man Page: `http://www.freebsd.org/cgi/man.cgi?query=a.out&sektion=5`.

Mach-O format supports retargetability of executables (also called *fat binaries*), i.e. the same executable file can be run on two different architectures.

Table 3.1: A comparison of common EFFs.

	ELF	**PE**	**Mach-O**	**E32Img**	**DEX**	**COFF**	**a.out**
Open standard	✓	✓	✓	×	✓	✓	✓
Classical EFF structure	✓	✓	✓	✓	×	✓	✓
Unlimited sections	✓	✓	256	✓	×	✓	7
Architecture independent	✓	✓	✓	ARM	×	✓	✓
Retargetable executables	×	×	✓	×	×	×	×
Explicit processor declar.	✓	✓	✓	×	×	✓	×

3.3 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. This section reviews two of the present day most common formats. See [109] for more details about theses formats.

3.3.1 DWARF

DWARF [19] is a debugging-data format originally developed in the mid-1980s at Bell Labs for Unix System V. Even though it is mainly associated with ELF [86], it is independent of EFF, programing language, operating system, or target architecture. The latest, fourth version has been published in 2010 [18] and the next version is planned to be released in 2014. However, its second version (DWARFv2) is still the most commonly used among compilers and debuggers.

DWARF information is stored in special sections of the executable file. Each of these sections contains different kind of debugging data, such as basic file information, source-code line and column numbers, look-up tables, call frames, and macros. The whole program is represented as a tree, whose nodes can have children or siblings. Each node is a *Debugging Information Entry* (DIE) structure that has a tag and a list of attributes. A tag specifies the type of the DIE (e.g. function, data type, variable) and attributes contain all the description details. There are two general types of DIEs:

- *Data DIE* – Those describing data and types such as base types, variables, arrays, structures, classes, etc. Locations of data are defined by location expressions, consisting of a sequence of operations and values, evaluated by a simple stack machine.

- *Code DIE* – Those describing machine code. Functions (they have a return value) and subprograms are treated as two flavours of the same construction. These DIEs typically own many other DIEs that describe them (e.g. arguments, local variables, lexical blocks).

There is also a special type of DIE called a *compilation unit* which is the parent for all DIEs created by a separate compilation of a single source file. It holds information about compilation (e.g. name of the source, language, used compiler) and locations of other DWARF data not described by DIEs (e.g. line number table, macro information).

DWARF is generated by many common compilers (e.g. GCC, LLVM `llc`, Intel `icc`)[16], it is supported by many debuggers, and there are several libraries and utilities that allow its examination or manipulation (e.g. `libdwarf`, `dwarfdump`, `objdump`).

3.3.2 PDB

PDB (the abbreviation is more common than the full name — *Program Database*) is a format developed by the Microsoft Corporation. It is generated only by Microsoft Visual Studio compilers and used by Microsoft debuggers. PDB is mainly used on the ARM and Intel x86 architectures and it is based on the older CodeView format. PDB debugging information for a particular application is stored in a separate file with the `pdb` extension. Each executable binary file in a PE format [57] has its counterpart in a PDB file. Both files are paired by a unique GUID code.

PDB is a proprietary format, and its specification has never been publicly published. Therefore, the analysis of this format can be only done via reverse engineering. The official DIA SDK is provided for processing debugging information[17], but this toolkit is platform dependent – it can be only used under the MS Windows operating system.

3.3.3 Other Formats

Except the two aforementioned formats, there are some other formats that can be used to store debugging information. For example, *Symbol Table Strings* (STABS), where data are stored as special entries in symbol table, and *Common Object File Format* (COFF) [26], that could contain debugging information in special sections much like DWARF. Both were strongly tied to specific EFFs and became obsolete along with them.

[16]This is true mainly for GNU and Linux compilers. For example, Microsoft compilers do not support this format.

[17]`http://msdn.microsoft.com/en-us/library/ee8x173s.aspx`

Chapter 4

Machine-Code Analysis: State of the Art

"Before software can be reusable it first has to be usable."

Ralph Johnson

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). These tools support only a limited number of target architectures and the support of others implies hand-made modifications of the tools' source codes. This approach is easier to implement for one particular architecture (e.g. create a disassembler for MIPS), but its extension for support of other architectures is problematic (e.g. it involves complete rewriting of code).

Afterwards, we give a brief overview of existing projects focused on dynamic analysis (e.g. debuggers, simulators). Once again, several tools are platform-dependent, but in comparison with static-analysis tools there is a significantly higher number of retargetable tools. These retargetable tools usually employ their own specific ADL for a description of the target architectures, i.e. processor models. These models are used for automatic generation of a complete toolchain that consists of multiple analytical tools (e.g. disassembler, simulator, debugger). Limitations of these tools are described as well.

Finally, we describe the Lissom research project in deeper detail because the novel methods presented in Chapters 5 and 6 are closely tied to this project. These methods are based on the Lissom project's ISAC ADL, they use the internal COFF-based file representation for storage of input applications, and they extend the existing tools (e.g. various types of simulators, debugger).

This chapter is based on [107, 109, 113, 118, 123] and [92].

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*[18]. This is a typical case of malware. The information about the used packer and its version can be used for *unpacking* via some of the existing unpackers that are tailor-coded for a particular version of a packer. Otherwise, analysis of packed code will be inaccurate or not possible at all.

At present, there are several tools focused on such preliminary analysis (the most common ones are mentioned in Section 7.3). The knowledge of the originally used tool (e.g. compiler, linker, packer) for executable creation is useful in several security-oriented areas, such as anti-virus or forensics software [83]. The overwhelming majority of existing tools are limited to the Windows Portable Executable (WinPE) format on the Intel x86 architecture and they use signature-based detection. Almost all of these tools are freeware, but not open source.

Formats of signatures used by these tools for pattern matching usually contain a hexadecimal representation of the first few machine-code instructions on the application's *entry point* (EP). EP is an address of the first executed instruction within the application. A sequence of these first few instructions creates the so-called *start-up* or *run-time* routine, which is quite unique for each compiler or packer and it can be used as its footprint. Accuracy of detection depends on the signature format, signatures quality, and used scanning algorithm. Furthermore, identification of sophisticated packers may need more than one signature.

Databases with signatures are either internal (i.e. pre-compiled in code of a detector), or stored in external text files. The second case is more readable and users can easily add new signatures. However, detection based on external signatures is slower because they must be parsed at first. In several cases, detection tools are also distributed together with large, third-party external databases.

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). The largest group consists of hand-coded tools that are focused on binary conversion between two particular EFFs.

In the past, a typical example was the Macintosh Application Environment project, which supported execution of native Apple Macintosh applications on UNIX based workstations. AT&T's FreePort Express was another binary translator, which permitted conversion of SunOS and Solaris executables into Digital UNIX executables. Furthermore, Wabi allowed conversion of executables from Windows 3.x to Solaris [33]. However, none of these projects is developed anymore.

Another important project is the Binary File Descriptor library (BFD) [12]. BFD was developed by the Cygnus Support company, and currently forms a part of the GNU Binutils package. It supports unified, canonical format for manipulation tens of EFFs (e.g. ELF, PE, COFF). BFD is used as a front-end of many existing projects, however, it is not a

[18]In the following text, we use the term *packing* for all the techniques of executable-file creation, such as packing, compression, protection, etc.

retargetable solution because support of each new EFF must be hand-coded. Furthermore, due to BFD's complexity, the interconnection of the target application and BFD is often difficult. More details can be found in [110].

The other projects use their own grammar-based systems for a formal description of binary formats. The architecture description language SLED [74], developed within the New Jersey Machine Code Toolkit [75], is supposed to describe the instruction sets of target processor architectures, i.e. syntax and binary encoding of each instruction. Such a description can be used for the automatic generation of a retargetable linker [24], debugger [76], or other tools. However, this language does not support a description of EFFs. We can find the same limitation in all common ADLs, see [52] for more details.

We can find two formalisms called BFF grammar (Binary File Format Grammar). The first one[19] (DWG BFF) was originally designed for a description of non-executable file formats. More precisely, it was designed and tested only on the AutoCAD DWG format. This grammar is a state-of-the-art concept, which has never been implemented, nor used in any tool. The grammar is limited to the DWG format, but it can be possibly used on other EFFs. Its author claims that the grammar is in the LL(1) form and it can be parsed by the recursive descent approach [1].

The DWG BFF grammar inspired project UQBT[20] and its SRL BFF grammar [89]. An example of this grammar is illustrated in Figure 4.1. Despite the limitations of the original DWG BFF grammar, it was simplified and used within this project for generation of the Simple Retargetable Loader (SRL). Although, it is claimed that this concept can be used for automatic generation of other retargetable tools, the grammar constructions are limited only for a simple loader. According to [89], the SRL was tested as an ELF loader for the existing decompiler `dcc` [13].

```
DEFINITION FORMAT
    header
    program_header_table
    sections
    section_header_table
END FORMAT

DEFINITION header
    h_ident SIZE 16
    h_type SIZE 16
    h_machine SIZE 16
    h_version SIZE 32
    ...
END header
```

Figure 4.1: Example of a BFF-grammar description of the ELF format.

Both BFF grammars have several limitations. For example, they are unable to properly model optional elements of EFFs, such as the missing section-header table in ELF; relocation information is not taken into account, etc. The most significant drawback of both grammars is the lack of semantic actions [1] (e.g. semantic checks, validation of EFF content, user-defined actions). Therefore, the grammars are only capable to describe syntactical structure

[19] http://www.iwriteiam.nl/Ha_BFF.html
[20] http://itee.uq.edu.au/~cristina/uqbt.html

of the input EFF, but modeling of context-sensitive properties is left to the user.

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. We can find dozens of existing disassemblers because their development is not so complex. 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.).

Disassemblers utilize one of the following algorithms for instruction decoding [48, 80]:

- *Linear Sweep* – This is a straightforward approach that simply translates all bytes in the code section of the input file in a linear sequence. The approach does not take into account any interleaved alignment (e.g. empty bytes between end of one function and beginning of another function) or data bytes stored in code sections (e.g. optimized store of constants directly in a code section – this is often done by GCC for ARM/Thumb). This leads to a fast disassembly, but with incorrect results in some situations. This is a bigger issue on CISC architectures where almost every byte sequence can be decoded as a valid instruction (e.g. Intel x86). This approach is for example used in the GNU `objdump`.

- *Recursive Traversal* – Recursive traversal tries to improve the linear sweep algorithm via decoding only the meaningful bytes (i.e. containing machine-code). This is done via a different traversal strategy that no longer inspects code linearly byte-after-byte, but it detects changes in control-flow and the instruction decoder continues with decoding on new target addresses. For example, if the branch instruction is detected, the processing continues on the target address of this branch instruction. The advantage of this approach is that it no longer produces incorrect instructions (e.g. a wrongly detected beginning of an instruction); however, it may skip large portions of code. This for example happens on architectures with indirect branches, i.e. the target of a branch is stored in the register and it is computed during run-time. However, a disassembler is a static-analysis tool and it does not have any run-time information. Therefore, complete functions are skipped because there are no direct branch (or call) instructions pointing to them. The same limitation applies to code segments related to HLL `switch` statements. These segments of code usually utilize the so-called *jump-tables* that are also evaluated only during run-time. Therefore, its `cases` are also unreachable in this approach. Furthermore, this approach has to know the semantics of each instruction to distinguish instructions that change the control-flow.

- *Hybrid Traversal* – This approach combines the benefits of both previous approaches. The recursive traversal is used at first for decoding of instructions reachable via direct branches/calls. In the second phase, the linear sweep is used for decoding of the remaining code segments. This can also be iteratively combined until decoding of all code segments, as demonstrated in [48].

Furthermore, these algorithms are also utilized in other tools that manipulate and analyse machine code. In decompilers, the instruction decoders work in a very similar way as disassemblers except they do not translate decoded machine instructions into assembly language, but they produce intermediate representation of these instructions.

At present, the most common disassemblers are:

- `objdump` – This tool is a part of the GNU Binutils package[21] and it uses the GNU Binary File Descriptor (BFD) library to do a low-level manipulation of input code. This project is ported on most of the common architectures and it also supports multiple EFFs[22]. It supports fast analysis of input file, but the results of disassembly are often inaccurate because of used linear sweep approach.

- *IDA Disassembler* – IDA is a commercial tool[23] with a rich set of functions (e.g. automatic annotation, CFG generation, built-in debugger) and a graphical user interface. It supports more than 60 target architectures and 30 EFFs. According to our information, IDA is not automatically retargetable and support of additional architectures has to be done manually. Furthermore, it utilizes the recursive traversal approach.

- *ODA* – This is a free online disassembler with an interactive web-based interface[24]. It supports a wide range of target architectures and EFFs.

- *Others* – BORG[25], Lida[26], objconv[27], etc.

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 description of existing machine-code decompilers in this section is deeper than the previous tools because it is more important to this thesis. On the other hand, this description is not intended to be an exhaustive list of all existing decompilers; we present only several milestones related to the machine-code decompilation. A more detailed description can be found in [13,21,49].

Pioneers of Machine-Code Decompilation

Machine-code decompilation has a surprisingly long history – it stretches back more than 50 years. Halstead reports that the Donnelly-Neliac (*D-Neliac*) decompiler was producing Neliac (an Algol-like language) code from machine code in 1960 [29]. At this time, decompilers were based on pattern matching and they left more difficult cases to the programmer to perform them manually.

Barbe's *PILER System* was the first attempt to build a generic decompiler. The system was designed to be able to process the machine code of several different target architectures, and generate code for several different HLLs. However, only one input phase (the GE/Honeywell 600 machine) and only two output phases (Fortran and COBOL) were completed [7].

[21] `https://www.gnu.org/software/binutils/`
[22] However, the ELF format is its primary target format.
[23] `https://www.hex-rays.com/products/ida/`
[24] `http://www.onlinedisassembler.com/odaweb/`
[25] `http://www.caesum.com/`
[26] `http://lida.sourceforge.net`
[27] `https://github.com/vertis/objconv`

Exec-2-C was an experimental project by Austin Code Works, but it has never been completed. Intel 80286/DOS executables were disassembled, converted to an internal format, and finally converted into a low-quality C code (no loop or conditional statements, it contained registers and inlined assembler, etc.)[28].

The *University of Queensland Binary Translator* (UQBT) also emits C source code. The output is not intended to be readable, and it is very difficult to read it in practice. However, the output is compilable, so UQBT could be used for optimizing programs for a particular platform, or cross-platform porting.

The *dcc decompiler* was developed by Cristina Cifuentes as her PhD thesis at the Queensland University of Technology in 1994 [13]. Its structure is similar to a classical compiler; it contains a front-end, middle-end, and back-end. The front-end is a platform-dependent module that reads a machine code and transforms it into an intermediate representation of the program. The middle-end (as known as the Universal Decompiling Machine or UDM) is a module performing the core of the decompiling analysis: data-flow and control-flow analysis. Finally, the back-end generates the C language code for the input program[29]. This decompiler only supports Intel i80286 architecture with combination of MS-DOS input executables. It is also capable of generating a call graph, control-flow graph, disassembled code, and several others [13].

Boomerang

Boomerang[30] is an open source project established in 2002 that was strongly inspired by UQBT and dcc. The original author is Mike Van Emmerik. 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. The aforementioned SLED language was used for a compact description of instruction syntax and coding. However, this language does not support a description of instruction semantics. In the words of its authors: "*To make the toolkit simple and general, we avoid describing the semantics of instructions, because too often semantic information is both hard to get right and of use only to a single application.*" [75]. Therefore, this language itself cannot be used for generation of tools like compilers or decompilers. Because of that, the authors of the Boomerang decompiler have to use SLED together with the RTL-based semantics description language SSL [14]. According to Boomerang's source code and author's notes, the usage of SLED/SLL was slow and error-prone for more complex processor architectures, such as Intel x86. Moreover, the final solution is not truly retargetable because several target-platform related parts are hand-coded.

During the first phase of the analysis, called decoding, the code is translated from machine code to an intermediate representation. Each instruction is decoded and it is expanded into the respective SSL semantics. The decoding is done via the aforementioned recursive-traversal approach. Once a code fragment is completely decoded and its control-flow graph is built, it is transformed into a static single assignment form (SSA, see Definition 21) [21].

One of its limitations is a missing recognition of statically linked code. This causes complications for decompiling programs that were compiled with the option `-static` and,

[28] http://www.program-transformation.org/Transform/HistoryOfDecompilation2
[29] http://itee.uq.edu.au/~cristina/dcc.html
[30] http://boomerang.sourceforge.net/

therefore, all system libraries, like `libc`, are linked to the program's code. The decompiled code is then disarranged because the code of a library can be larger than the code of the program itself. The output is generated in the C language, but there is no effort to generate code with correct syntax. According to the official site[31], the development was discontinued in 2006.

REC Studio

REC Studio—Reverse Engineering Compiler[32] 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 uses more powerful analysis techniques such as partial SSA and supports 32-bit and 64-bit executables. Furthermore, REC has loaders for more EFFs: WinPE, ELF, COFF, and Mach-O. Its author claims that debugging information is also supported and the decompiler can process the DWARF format and partially the PDB format.

The author also noted on his web page that the disassemblers used in REC were taken from various sources. Due to this fact, we estimate that it is not a retargetable decompiler and support of other architectures may be difficult as well as a maintenance of code from different origins. Furthermore, the software is available for Windows, Linux (Ubuntu), and Mac OS X. However, this software was quite unstable on several architectures (e.g. Windows) during our testing, and it often crashes during decompilation (this applies to the latest version 4, released in 2014).

Hex-Rays Decompiler

The *Hex-Rays Decompiler*[33] 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[34]. It also supports both major EFFs—UNIX ELF and WinPE. The output is generated as a highly readable C code; however, the output is not designed for re-compilation, only for more rapid comprehension of what the program is doing.

This software is commercial and distributed without sources. Furthermore, the details about its internals are not publicly known. The first version of the x86 decompiler was released in 2007, support of the ARM decompilation was added in 2010, and x64 in 2014. The current version is 1.9. Its author, Ilfak Guilfanov, claims that this is the first decompiler able to process real-world executables.

The plugin enhances the existing disassembler with another view over the input executable and adds several new features. The decompilation itself is very fast and oriented on function detection and recovery.

It supports most of the common features like distinguishing loop types (`for`, `while`, presence of `breaks`, etc.), creation of compound conditions, usage of debugging information, highly accurate recovery of functions, arguments, and return values, etc. There is also a software-development kit (SDK), which gives access to the decompiler's internals and one

[31]http://boomerang.sourceforge.net/

[32]http://www.backerstreet.com/rec/rec.htm

[33]www.hex-rays.com/products/decompiler/

[34]There are several limitations in this decompiler, e.g. not supported FPU and VFP/SIMD/Neon instruction sets in ARM. For details see https://www.hex-rays.com/products/decompiler/manual/limit.shtml.

can easily create new plugins or scripts (using Ruby or Python). It also has a useful GUI, which helps with easy understanding of the decompilation process and better readability of the generated code. The interface is interactive; therefore, it is possible to fine-tune the results (e.g. specification of data structures, function arguments).

Decompile-it.com

The author of this project is Naftali Schwartz. This project is tightly linked to the Valgrind[35] framework. Therefore, it probably has the same advantages and disadvantages as Valgrind. Valgrind is a framework for building dynamic analysis and instrumentation tools (e.g. memory error detector, cache and branch-prediction profiler, thread error detector). It supports several UNIX-based target operating systems (e.g. Linux, Android, Darwin) and target architectures (e.g. MIPS, ARM, x86).

Based on our tests done in early 2013, this tool is limited to the decompilation of x86/Linux executables and it seems that debugging information is mandatory. The official info claimed that the following problems are at least partially solved—reconstruction of composite types and unions, detection and transformation of instruction idioms, and recovery of switch statements, unrolled loops, and inlined functions. According to the author's note[36], the MIPS and ARM architectures are also supported as well as C++ generated executables.

Up to the day of this thesis creation, there is no published article about this project, therefore, its description is very brief. The project has not been released yet and the only possibility of how to try this tool was a web interface on its homepage[37]. However, this site expired in 2014 and there has been no further news about development of this decompiler.

SmartDec

The original name of this decompiler is *TyDec* and it was firstly presented in 2009 [87]. Later, it was renamed to *SmartDec*[38]. At present, there is a downloadable beta version of this tool. There also exists a plug-in into IDA disassembler with a similar approach as Hex-Rays Decompiler. SmartDec is also a proprietary software distributed without sources.

The work on the decompiler was based on research related to automatic type reconstruction in disassembled C programs [16]. This is the reason why SmartDec only allowed assembly code as its input in previous versions. Support of input executable files was added in December 2012. However, the quality of output generated by this tool is still quite low and it needs further development, the details are discussed in [92].

This decompiler is focused on decompilation of executables produced by C++ compilers. It supports specific C++ constructs, such as virtual functions, classes, class hierarchies, i.e. inheritance relations between classes, constructors, destructors, types of pointers to polymorphic classes, non-virtual member functions, layout and types of class members, calls to virtual functions, and exception raising and handling statements.

[35]http://valgrind.org/
[36]http://sourceforge.net/p/valgrind/mailman/message/29903738/
[37]http://decompile-it.com/
[38]http://decompilation.info/

Other Approaches

We can also find other approaches of machine-code analysis. *DeDe* is an example of a machine-code decompiler focused on one particular source language—Delphi. It achieves the best results with reconstruction of resources (e.g. forms, strings), but decompilation of code is insufficient because it only produces well-commented assembler code. Therefore, the code is not re-compilable in Delphi.

The *Jakstab project* [42] is a static analysis framework written in Java, which currently supports the x86 architecture and 32-bit WinPE and ELF executables. Its purpose is not to decompile file, but translate it into a low-level IR language. Afterwards, it performs several data and control-flow analyses over this representation that can be used for "smart" disassembly, function detection [43], etc. It is designed to be adaptable to multiple hardware platforms by using customized instruction decoding and processor specifications similar to the Boomerang decompiler. The whole system was developed as a part of Johannes Kinder's PhD thesis [41].

Decompilers Comparison

In Table 4.1, we summarize the base features of the decompilers and knowledge obtained from our tests [92]. In this table, we can see that the most popular file formats are WinPE and ELF, as well as the Intel x86 architecture. The C language is the most common output language. Other features are supported less often. Furthermore, none of the actively developed decompilers is available with its source code and most of these tools tend to be proprietary software.

According to our previous testing [92], Hex-Rays Decompiler is capable of producing the best outputs from this set of decompilers. Boomerang takes the second place, and the REC Decompiler is in the third place; the last position is SmartDec.

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. Furthermore, it is often used for testing applications together with newly designed architectures in the field of electronic design automation (EDA).

A proper debugger is used for finding, isolating and removing bugs that are detected during the application run-time. For this purpose, the analyst needs complex context information about the application state and an overview of the target environment, see [77].

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 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). Analysts should have a good interface allowing an easy breakpoint setting, inspection of a particular pipeline stage etc. It can be implemented either by the command line interface (e.g. `gdb`) or via the graphical user interface (e.g. `ddd`).

Table 4.1: Overview of features and tests of existing machine-code decompilers. The features marked with an asterisk (*) are claimed by the tools' authors, but are not included in any publicly available release. The criteria for measuring quality of output is based of our survey done in [92].

	dcc	**Boomerang**	**REC Studio**	**Hex-Rays**	**decompile-it.com**	**SmartDec**
Supported architectures	x86	x86 SPARC PPC	x86 SPARC MIPS	x86(-64) ARM	x86* MIPS* ARM*	x86(-64)
Supported EFFs	DOS-MZ	ELF WinPE	ELF WinPE COFF Mach-O	ELF WinPE Mach-O	ELF	ELF WinPE
Input	binary	binary	binary	binary	binary	binary+asm
Output language	C	C	C-like	C	C	C/C++
Distribution	source	source	binary	binary	web-service	binary
License	GPL	BSD+GPL	freeware	commercial	unknown	unknown
DWARF dbg support	×	×	✓	✓	✓	×
PDB dbg support	×	×	partial	✓	×	×
Interactive interface	×	✓	✓	✓	×	×
Statically-linked code detection	✓	✓	×	✓	×	×
Detection of instruction idioms	×	×	×	✓	×	×
Retargetability	×	partial	×	×	×	×
Quality of output	deprecated	middle	middle	high	not available	low

The debugged application can be written in an assembly language or in any HLL, such as C language. Debugging of applications written in the assembly language is done at the *instruction level*, while debugging of applications written in an HLL is done at the *source level* (also known as the statement-level). The *cycle-accurate* level refers to debugging of a micro-architecture. On this level, the analyst can easily see the instruction flow through the pipeline registers.

Each of the debugging levels can be useful in a different situation. The on-chip debugging features allow the debugging on a real hardware. It is very useful either for the testing of the hardware or for the debugging of the target application (the speed of the real processor is usually higher than the speed of the simulator). In the rest of this thesis, we will focus on source-level debugging because the main contribution lies in this field.

The multi-level architecture-specific debugging is a problem that has already been solved and we can find several existing debuggers with such features. Implementation of this tool is difficult, but straightforward task [77]. At present, the most common non-retargetable debuggers are:

- *Intel Debugger* (`idb`) – It was deprecated in 2013 and replaced by `gdb`. It supported debugging applications written in C, C++, and Fortran compiled for Intel x86-64.
- *OllyDbg* – This debugger is mainly used for machine-code debugging (i.e. instruction-level), but it also supports source-level debugging whenever the PDB debugging information is present. This tool is often used for malware analysis. At present, it supports only Intel x86 architecture, but the x86-64 version is under development[39].
- *Microsoft Visual Studio Debugger* – This debugger and its GUI is shipped along with Microsoft Visual Studio. It supports debugging of user created applications in several languages such as C, C#, ASP.NET, etc. The supported architectures are Intel x86(-64) and .NET. It does not allow debugging of kernel-mode code and it only supports Microsoft Windows.
- *WinDbg* – WinDbg is another debugger developed by Microsoft. In comparison with Visual Studio Debugger, it is more powerful and it also supports debugging of drivers and kernel-mode code. However, its GUI is not so rich as the other one.

Retargetable debugging is a more challenging task. We can find several projects focused on this task, such as GNU Debugger `gdb`[40], LLVM debugger `lldb`[41], or `ldb` [76]. These projects use the concept where the debugger's core is the same for all architectures (e.g. 350,000 lines of C/C++ code for `gdb` version 7.2) and the specifics of each particular target architecture is manually described in separate files (hundreds of code lines for each one of them). For example, the GNU Debugger currently supports over twenty manually implemented target architectures.

This is not effectively applicable to some areas of debugger usage, such as processor design and testing. Such processor architecture is changing frequently during its design and the debugger recompilation is slow because of the core complexity. Furthermore, the `ldb` project supports only one programming language (ANSI C), only one compiler (`lcc`), and its format of debugging information is not supported by any other compiler or debugger.

[39]http://www.ollydbg.de/odbg64.html

[40]http://www.gnu.org/software/gdb/

[41]It is a very promising project with ambitions to be competitive to `gdb`. However, it is still in an early stage of development. It supports x86-64 and ARM at the moment. Homepage: http://lldb.llvm.org/

The present trend of designing embedded systems is to use ADLs, see [5]. The processor is described in the selected ADL and the complete toolchain is automatically generated based on this description. The toolchain consists of the tools for processor programming, such as the C compiler or assembler, and tools for processor simulation, such as different types of simulators or the profiler (these are described in the following section). This approach significantly reduces the design time and its cost. No matter how effective these compilers and other tools are, the newly created applications still need to be debugged because they may contain bugs. Therefore, it is necessary to generate a retargetable multi-level debugger too.

We can find several projects focused on a rapid processor design and debugging. All of these projects exploit their own ADLs for the toolchain generation. However, only a few projects allow the proper multi-level debugging. The projects can be divided into three groups according to the type of the debugging features:

1. The first group consists of projects without the sophisticated debugging tools, e.g. the xADL project [9] developed at the Vienna University of Technology. The debugging capabilities of this project are limited to the basic program instrumentation and the proper debugging is not supported.

2. The second project group uses the automatically generated architecture descriptions for the `gdb` core. However, these projects involve previously described limitations (e.g. limited number of supported languages and tools, slow re-generation of debugger during changes in architecture). Among these projects falls the ArchC[42] and SimnML [23].

3. The last group of projects uses their own implementation of the debugging tools (e.g. LisaDB). The EXPRESSION [28] and LISA [32] languages create the background of these projects. Both of them allow a description of the systems on the cycle-accurate and instruction-accurate levels. Unfortunately, there is almost no public information about these proprietary solutions.

4.6 Tools for Simulation, Emulation, and Instrumentation

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 [69, 77]. Roughly speaking, *simulation* is a model-based approach that behaves similarly to guest, but it is implemented in an entirely different way. A real-world example is a flight simulator – it simulates a flight, but it is unable to physically transport someone from one point to another. On the other hand, *emulation* is focused on reproduction of the same exact external behavior like the emulated system. It is effectively a complete replication of another system, right down to being binary compatible with the emulated system's inputs and outputs, but operating in

[42] http://archc.sourceforge.net/

a different environment to the environment of the original emulated system. Therefore, a proper emulator is capable of becoming a substitute for the original system.

Simulators and emulators are primarily focused on execution of applications, but their support of machine-code analysis is very limited. For this reason, they are often enhanced by instrumentation features. Instrumentation is a process of monitoring and measuring different aspects of program during its execution, e.g. creating a list of inter-procedural calls, monitoring I/O operations, tracking of the performance. Instrumentation can be done on a real system as well as within an emulator or simulator. The most common tools are listed in the rest of this section.

- *QEMU* – The QEMU project[43] is focused on emulation of complete target architectures. At present, the QEMU project supports various target architectures (x86, x86-64, MIPS, SPARC, ARM, PowerPC, and others) and guest and host systems (Linux, Windows, Mac OS X). QEMU is distributed under the GNU GPL license. It uses dynamic translation of instructions (from guest's instruction set to host's instruction set), which implies much better performance than simple emulation.

- *Bochs* – Another interesting project is the Bochs emulator[44]. In comparison with QEMU, Bochs is focused only on x86(-64) guest architecture, but in greater detail because it supports almost all x86-based CPUs. Bochs is capable of running most guest OSs including Linux, Windows, or DOS.

- *Intel Pin* – The two aforementioned emulators are not focused on code analysis and they lack any such features. In other words, emulators can help with code analysis only in cooperation with other (instrumentation) tools. One of the most common instrumentation tools is Intel Pin[45]. Pin is proprietary software (i.e. no sources available) developed by Intel that is supplied free of charge for non-commercial use. It is limited to Intel x86(-64) and Intel Xeon Phi architectures and it supports Windows, Linux, and Mac OS X. The previous attempt to use Pin on ARM architecture has been dropped by Intel [31] as well as support of Intel Itanium [65].

  Pin is a framework for creating your own instrumentation tools (so-called *PinTools*). It provides a rich API that can be quite easily utilized to create various types of tools with different features, such as inspection of code (instructions, basic blocks, calls traces, etc.), profiling usage of resources (memory, registers, cache, etc.), or monitoring of thread and processes. However, the analysed application has to be run directly on the host system because Pin does not provide any simulation or emulation features. Therefore, instrumentation of malware has to be done in a sand-box[46] or in emulated or virtualized machine[47].

[43]http://wiki.qemu.org/

[44]http://bochs.sourceforge.net/

[45]http://www.intel.com/software/pintool

[46]*Sandboxing* is a security mechanism used primarily in AntiVirus software that isolates execution of untested (potentially malicious) applications from the rest of the system. The side-effects of execution (e.g. modification of files or registers) are stored in scratch space operated by sandbox. If the application is classified as malware via analysis of its behavior, there will be no harm to the host system.

[47]Roughly speaking, *virtualization* is closely related to emulation. Emulation can create a completely different guest architecture running on host, where the guest's instruction set will be emulated via instruction set of the host. This is a robust solution, but it may be very slow because of such translation. On the other hand, virtualization only supports guests with instruction sets compatible with the host's set. The guest's instructions are executed with minimal slow-down. This is, for example, useful for running guest OS in

As we can see from the list of supported architectures in these tools, emulators are mostly used for emulation of existing architectures, which already have a robust design. On the other hand, simulators are mostly used for testing of newly designed architectures (especially ASIPs) during all phases of their creation. In this case, the design of particular target architectures is simulated together with an application for this architecture. Therefore, the simulator is used for testing and validation of the design and it can uncover hidden bugs. Design of both architecture and application changes frequently and it is necessary to recreate a complete toolchain, including a simulator, every time. Therefore, it is necessary to support retargetability in these tools.

We can find three basic types of simulators; each having different advantages and disadvantages and usable in the different phases of a processor design. A more detailed description of these simulator types is given in Section 5.2. Additional details can be found in [69]. These types are:

- *Interpreted simulator* – It is the basic type of simulator. The run of an interpreted simulator is based on the following concept. It fetches an instruction, then it decodes the instruction and finally it executes this instruction. Therefore, the simulator is not dependent on the simulated application and it, for example, supports self-modifying code (SMC) out of the box. On the other hand, it may constantly fetch and decode the same instructions (e.g. instructions within a loop), which makes it the slowest simulation type.

- *Compiled simulator* – This is an improved version of the interpreted simulator. Firstly it analyses the target application, and then the simulator itself is created. It is faster than the interpreted simulator, but each particular simulator is created for one particular application. Therefore, each change in such an application involves regeneration of a compiled simulator.

- *Translated simulator* – This is an even more improved version of the compiled simulator that is used only for a target software development. Therefore, the processor micro-architecture is not simulated. It is also generated based on a particular application and furthermore, it uses debugging information about basic-blocks within the target application. On the other hand, the translated simulator can achieve much higher simulation speeds than the previous two types.

The former two types are commonly used in the aforementioned projects ArchC and LISA. However, none of these projects supports a concept of translated simulation. Translated simulation is only supported in the xADL project. In this project, the LLVM platform is used for simulator creation; therefore, the creation of a simulator takes a long time since the LLVM back-end has to be compiled [84]. The overview[48] of instruction-accurate simulators within these projects is depicted in Table 4.2.

4.7 Lissom Project

In this section, we describe the Lissom research project[49] that creates a background for the newly designed methods proposed in this thesis. The Lissom project is located at

different type of OS (e.g. running Linux under Windows). There are more differences, but these are out of the scope of this thesis.

[48]It is the state at the beginning of this thesis.

[49]http://www.fit.vutbr.cz/research/groups/lissom/index.html

Table 4.2: Overview of instruction-accurate simulator types within the ADL-based projects.

	Simulator type		
	interpreted	compiled	translated
ArchC	✓	✓	×
nML	✓	✓	×
xADL	✓	✓	✓
LISA	✓	✓	×
Lissom	✓	✓	×

Brno University of Technology, Faculty of Information Technology, 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 of each processor. This section is based on [117, 121]. The original motivation of this project is summarized in the following paragraph cited from [117].

"*Embedded systems have become the essential part in all areas of our life (e.g. car industry, informatics, automation). Each such processor has usually dedicated functionality and it is highly optimized for its task. However, there are many trade-offs among which part of functionality should be implemented directly in the processor and which part should be implemented in software. That is where the methodology of design space exploration (DSE) comes into play. It is a process of searching the optimal design of the processor. In order to support effective DSE, it is necessary to provide a good integrated desktop environment (IDE) for the processor design. Moreover, this IDE should provide automatic toolchain generation based on the processor description in a particular ADL. This toolchain consists of the tools for processor programming (e.g. assembler, C compiler), and of the tools for processor simulation (e.g. simulator, profiler). The Lissom project provides all these features.*"

During the start of a work on this thesis, the development of new ASIP chips was the primary target of the Lissom project. The ISAC ADL has been designed for description of embedded processors and the toolchain was automatically generated based on the ISAC description of a particular processor. The original toolchain contained a C compiler, assembler, linker, simple disassembler (without any reconstruction of control-flow, etc.), two types of simulators (interpreted and compiled), the assembler-level debugger, and a prototype of hardware-description generator.

Moreover, we realized that we can re-use this concept for description of existing architectures, such as Intel x86, MIPS, ARM, PowerPC and many others. The generated toolchain can be used as a replacement of existing insufficient tools (e.g. obsolete or ineffective) or whenever a particular tool is missing (e.g. simulator, C compiler). 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 the Lissom project that create a background for the newly designed methods. The ISAC ADL is briefly presented in Subsection 4.7.1. Afterwards, we describe the internal EFF in Subsection 4.7.2. Finally, we present existing tools in Subsection 4.7.3. For more details about the Lissom project, see [34, 52, 69].

4.7.1 ISAC Language

A processor can be either described by using hardware description language (HDL) or architecture description language (ADL), see [59]. Generally, ADLs are better for fast DSE and rapid processor prototyping, since ADL hides hardware details. Those details can be unknown at the beginning of the processor design or the designer does not want to take care of them.

The ISAC (Instruction Set Language C) language [34,52] 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[50].

The processor model in ISAC consists of two basic parts. In the first part, processor resources, such as registers or memories, are described by using the `RESOURCES` construction. In the second part, the processor instruction set and processor micro-architecture can be described by using the `OPERATION` construction.

The second part forms four basic models of the processor: the *instruction-set* model, *timing* model, model of *instruction analysers hierarchy*, and *behavioral* model. Each model specifies different features of the processor and it describes the processor from a different point of view.

The instruction-set model is formed by the `ASSEMBLER` and `CODING` sections. The `ASSEMBLER` section describes the textual form of an instruction (i.e. assembly language). Similarly, the `CODING` section describes the binary form of the instruction (i.e. machine code).

The `ACTIVATION` section is used for the timing model description. It denotes what and when will be done during the target application execution, such as pipeline stalls, etc. The `STRUCTURE` section is used for the model of hierarchy of instruction analysers. It specifies which decoder will be activated during the target application execution (e.g. pre-decode and decode phase of instruction execution in the pipeline).

The behavioral model is specified by sections `EXPRESSION` and `BEHAVIOR`, where the ANSI C language is used (i.e. ANSI C describes the behavior of instructions and processor micro-architecture). Note that the `EXPRESSION` section has the same meaning as the `return` statement in a C function (i.e. it is used for returning a value if a particular operation is used during instruction decoding). These two sections describe the behavior of an instruction or its part and they can also describe the behavior of the functional units in the processor micro-architecture (i.e. operations without the `ASSEMBLER` and `CODING` sections).

Note: the instruction-set description in sections `CODING` and `BEHAVIOR` is the most important for all the tools proposed in the following chapters (e.g. translated simulator, source-level debugger, retargetable decompiler).

The operations can be grouped according to some criteria, such as similar functionality (e.g. operations describing arithmetic instructions). The `GROUP` construction is used for grouping of that operations or other groups. An operation can use other operations or groups using the `INSTANCE` statement (e.g. an operation describing move instruction uses another operation describing immediate operand). Furthermore, there is a mandatory operation called `main`. This special operation is used for synchronization (i.e. clock-cycle generation).

A snippet of an ISAC instruction-level model is shown in Figure 4.2.

[50]The instruction-set description is more important for this thesis because it is fundamental for description of machine code. The micro-architecture description can be used for enhancing the proposed methods of machine-code analysis and it is marked as a future research.

```
/** Description of resources **/
RESOURCES {
    PC REGISTER bit[32] pc;      // Program counter
    REGISTER bit[32] gpregs[32];// Register file
    RAM bit[32] memory {         // Memory, etc.
        ENDIANESS = little,
        SIZE = 0x20000,
        FLAGS = {R, W, X}        // Read, Write, eXecute
    };
}

/* Textual and binary description of registers */
OPERATION reg REPRESENTS gpregs
{ ... }

/** Instruction-set description **/
/* Description of operation codes */
OPERATION opc_add {
    ASSEMBLER  {"ADD"};          // Assembler syntax
    CODING     {0b100000};       // Binary coding
    EXPRESSION {0x20;};          // Returns value of the operation
}

OPERATION opc_sub{
    ASSEMBLER  {"SUB"};
    CODING     {0b100010};
    EXPRESSION {0x22;};
}

/* Grouping of operation codes */
GROUP opc = opc_add, opc_sub;

/* Description of instructions */
OPERATION instruction_set {
    INSTANCE reg ALIAS {rd, rs, rt}; // Usage of registers
    INSTANCE opc;                // Usage of other ops. and groups

    ASSEMBLER { opc rd "," rs "," rt };
    CODING { 0b000000 rs rt rd 0b00000 opc };
    BEHAVIOR {                   // Instruction's behavior
        switch (opc) {
            case 0x20:           // Instruction ADD
                regs[rd] = regs[rs] + regs[rt];
                break;
            case 0x22:           // Instruction SUB
                regs[rd] = regs[rs] - regs[rt];
                break;
        }
    };
}
```

Figure 4.2: Simplified ISAC ADL model of the MIPS architecture.

In this example, we describe a MIPS 32-bit architecture, which contains a register file of size 32. The simplified instruction set contains one instruction for addition (`ADD`) and one for subtraction (`SUB`). Both of them read source operands from registers (`rs` and `rt`) and stores the results in another register (`rd`). The micro-architecture is not specified in this example for simplification.

It should be noted that the description of the instruction set is strictly separated from the description of micro-architecture (the `CODINGROOT` and `STRUCTURE` sections). This approach brings powerful modeling possibilities because there can be more decoders, which decode an instruction in a particular order (e.g. Intel's 8051), while the instruction is encoded in a different order.

One of the author's contributions is the extension of the ISAC language for modelling parallel architectures, with primary focus on the VLIW architecture. Information about this topic is described in deeper detail in [121, 122].

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). We will refer to this format simply as 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 [45].

The user-created applications for the ASIP processors (either in C or assembly-language) are translated by compiler, assembler, and linker into a COFF file. The same representation is used as an input to other tools, such as simulators, disassembler, debugger, etc.

The structure of Lissom COFF is similar to the classical COFF format. Basically, it has one header, followed by section headers, sections, and symbolic information (symbols, relocations, and debugging information). The section's content is characterized by section flags. The format of COFF is textual; therefore, it is possible to study its content without any additional tools, as we can see in Figure 4.3. In this example, we can see an application created for a 32-bit little-endian architecture. It is an executable file with a predefined entry-point address. The COFF file contains 30 sections and one symbol table. The first section is called `.text` and it contains 10,536 bytes of code.

4.7.3 Toolchain

In the remaining text of this chapter, we describe the Lissom toolchain generator and several of its tools. We focus on tools that are referred to in the following chapters and we skip tools that are not so important for this thesis (e.g. GUI, hardware realization, assembler).

During the toolchain generation, the user-described ISAC model is validated and transformed into an XML representation at first. The model in this representation is further processed by two tools – *toolchain generator* and *semantics extractor*, see Figure 4.4. The toolchain generator produces tools like C compiler, debugger, simulator, etc. Each generated tool is built from two types of components: (1) the pre-compiled platform-independent components that are the same for each target architecture (e.g. user interface in debugger) and (2) automatically generated source-codes realizing platform-specific tasks (e.g. instruc-

```
AgT62kG9y7    // Magic string
32            // Word bit-size
4             // Bytes per word
0             // Byte order, 0-little, 1-big
X             // Flags (eXecutable, etc.)
1             // Is the entry point set?
143654972     // Byte address of the entry point
30            // Section count
1             // Symbol table count
...           // Information about sections
.text         // Section header name
0             // Section byte alignment
1             // Is address absolute?
143654972     // Section address
T             // Section flags (Text, Data, BSS, etc.)
10536         // Section data size in bytes
0             // Count of relocations
20            // First line of section data
0             // First line of relocation data
              // Section data follows
00111111110000001111110000000101 // content of section .text
00000000000000000000000000000000
```

Figure 4.3: Simplified example of the Lissom COFF format (several attributes are not listed).

tion decoder in simulator). The generated source-codes are compiled and linked together with the fixed parts afterwards.

Before we describe these generated tools, we must mention the semantics extractor that 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[51], for details see [36]. It transforms ANSI C code from the `BEHAVIOR` section of the instruction into a sequence of LLVM IR-like instructions[52], which properly describe its behavior. Furthermore, information about assembler syntax and binary encoding is extracted from sections `ASSEMBLER` and `CODING`. An example of such extracted information for instruction `ADD` from Figure 4.2 is depicted in Figure 4.5.

The file with extracted semantics for every instruction in an instruction set is used for an automatic generation of the back-end part of the Lissom *C/C++ compiler* [36]. The front-end of this compiler is platform independent and it translates the user C/C++ applications into an intermediate code representation. This part is done via the LLVM Clang[53]. Furthermore, the middle-end phase optimizes this intermediate representation by a wide range of platform-independent optimizations, see [1]. This part is done via the LLVM `opt` tool. Finally, in the back-end part, the optimized intermediate representation is further optimized via platform-specific optimizations and emitted as instructions for the

[51]Extraction is possible only from instruction-accurate ISAC models at the moment. The cycle-accurate models are not supported by semantics extractor and C compiler. Therefore, it is necessary to create both models when you e.g. need micro-architectural simulation and C compiler at the same time.

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

[53]http://clang.llvm.org/

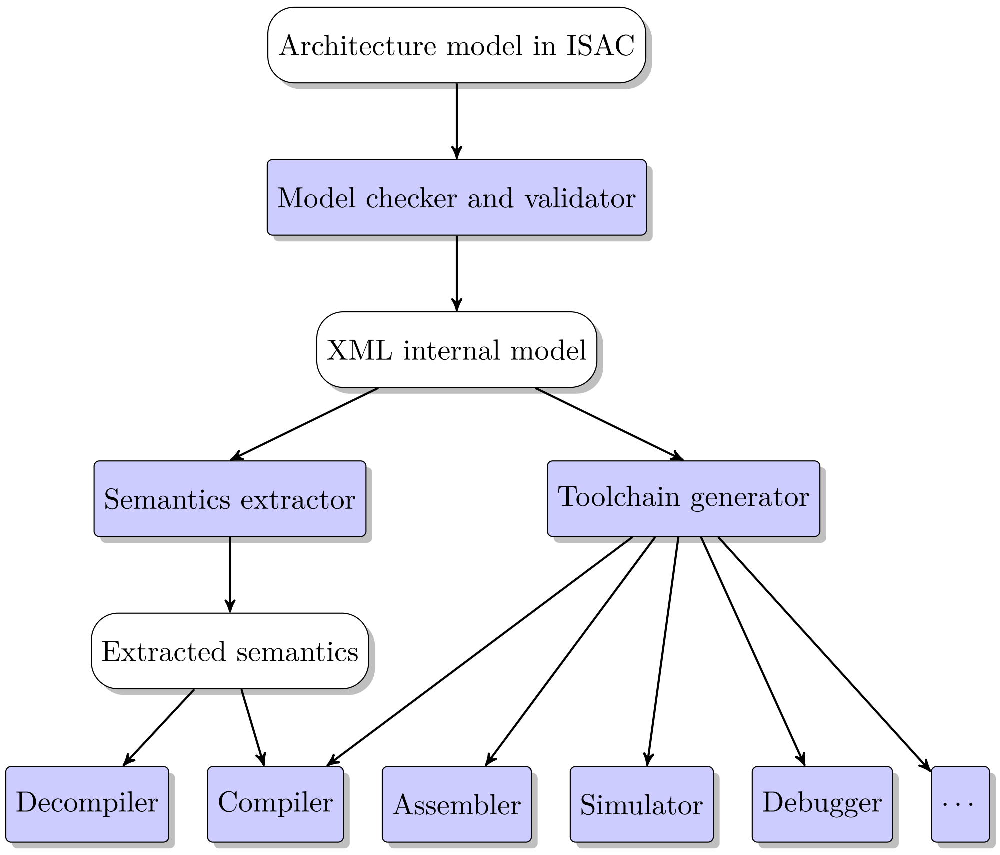


Figure 4.4: Overview of Lissom toolchain and its generation.

target architecture. In other words, the front-end is the only platform-dependant part of the Lissom compiler. The back-end part uses these semantics descriptions during a pattern-based searching for the most suitable instruction for each particular intermediate-code sequence. As a trivial example, the description in Figure 4.5 can fit for the `+` operator for integer numbers in C. The back-end is also based on the LLVM compiler infrastructure [84].

This compiler supports emission of debugging information in the DWARF format. The author of this thesis has a small contribution in this part by implementing particular generators of DWARF information, e.g. mapping of DWARF registers to physical registers, emission of the line number program (`.debug_line`), which maps code locations to source code locations and vice versa. The details about processing and usage of this debugging information will be described in the following chapter.

At the beginning of this thesis, applications compiled via the Lissom compiler could be simulated in two versions of simulator – *interpreted* and *compiled*. As we mentioned in Section 4.6, 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. However, this is achieved via a dependency on the target application, which means that there has to be a generator of compiled-simulator for each particular target application.

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,

```
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 4.5: Example of an extracted semantics for MIPS instruction `ADD` from Figure 4.2.

the simulator is used for their execution and debugging. Any type of simulator can also be used for this task. At the beginning of this thesis, the Lissom debugger supported two levels of debugging – instruction level and micro-architectural 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. One of the other similar limitations was the support of mapping from machine-code to assembly code, but not to a source-code. Therefore, stepping and tracing was done only on a level of assembly instructions. Other details about this simulator are mentioned later in Subsection 5.3.1. Whenever the assembly code was unavailable (e.g. debugging of standard C library), it was possible to use the Lissom disassembler. However, this disassembler lacks any control-flow analysis (i.e. it uses the linear sweep approach) and the results may be inaccurate (especially on CISC architectures where it is harder to determine the correct start of instruction).

Furthermore, it is also possible to profile the application as well as the design of the architecture by using multiple levels of the profiling tool, see [72].

Chapter 5

Retargetable Dynamic Analysis

"If debugging is the process of removing bugs, then programming must be the process of putting them in."

Edsger W. Dijkstra

This chapter is based on [109, 113, 118–120] and it describes the author's contribution within the area of dynamic machine-code analysis.

As we mentioned in Sections 4.5 and 4.6, 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 (e.g. detection of performance bottlenecks, generating statistics of instruction usage), detection of malicious behavior (e.g. emulation, sand-boxing, API-call monitoring), 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. Their task is to detect the presence of debugging information within the executable file or in a separate file, extract the low-level representation of the carried information, and transform it into a unified high-level representation that is easy-to-use by the subsequent methods. We propose two new libraries for this task — the first one for the DWARF format, the second one for the PDB format (see Section 3.3 for details about these formats). The DWARF library 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 [72]. The PDB-processing library is currently used only in the retargetable decompiler, but it might be used by others as well. We describe both of them in this chapter because they are tightly bound together.

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 (e.g. interpreted simulator and compiled simulator, see [69]), the newly designed methods lack a simulation of processor micro-architecture; on the other hand, they provide higher simulation speeds as is demonstrated in Chapter 7. By using these tools, it is possible to simulate a given executable file and analyse its run-time behavior (i.e. dynamic analysis).

Finally, in Section 5.3, we present an extension of the retargetable debugger. This tool can operate on two levels of abstraction — instruction-level and source-level. The former one is more low-level and it operates on the level of machine instructions, registers, memory cells, etc. Basically, it is only able to investigate the state of the hardware resources during debugging, but it is mostly unusable for analysis of the original application on its source-level (e.g. obtaining values of variables, detection of currently executed function, generation of stack back-trace). This type of debugger has already been done by other members of the Lissom project. The contribution of this thesis lays in the second level (i.e. source-level debugging) that is built on the top of the existing debugger. This source-level debugger supports dynamic analysis on the source-level with all the previously mentioned features. This extension is used for debugging of applications generated by the Lissom C compiler, see Subsection 4.7.3 for details.

5.1 Obtaining and Processing Debugging Information

In Subsections 3.3.1 and 3.3.2, we have briefly described 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 [50] and [109].

5.1.1 DWARF Parsing

As we have already noted in Subsection 3.3.1, 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:

- `.debug_info` – the main debugging information about the application.
- `.debug_abbrev` – list of abbreviations used in section `.debug_info` (this feature is used for reduction of debugging-information size);
- `.debug_line` – line number information, i.e. mapping of HLL code lines to machine-code addresses and vice versa;
- `.debug_ranges` – mapping between symbols and memory addresses (e.g. position of variables, constants, function arguments);
- for the description of other sections, see [19].

We can illustrate DWARF information representation on a minimal C program listed in Figure 5.1. We can compile this source code into a statically-linked MIPS ELF executable by using GCC compiler with options `-O0 -g -static`. The size of the resulting ELF file is 282 kB. Furthermore, it contains nine DWARF sections of a total size 219 kB, see Figure 5.2.

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

Figure 5.1: A simple *Hello-world* source code in C.

```
ELF Header
        Class:      ELF32
        Encoding:   Little endian
        Type:       Executable file
        Machine:    MIPS R3000
        Version:    Current
        Entry:      0x890003c

Section Headers:
[ Nr ] Type       Addr      Size      ES  Flg  Lk  Inf  Al  Name
[   1] PROGBITS   0890a280  00000acc  00  WA   00  000  08  .data
[   2] PROGBITS   0890003c  00009a2c  00  AX   00  000  04  .text
...
[  20] DEBUG      00000000  00000980  00       00  000  08  .debug_aranges
[  21] DEBUG      00000000  00000e32  00       00  000  01  .debug_pubnames
[  22] DEBUG      00000000  0001a939  00       00  000  01  .debug_info
[  23] DEBUG      00000000  0000647f  00       00  000  01  .debug_abbrev
[  24] DEBUG      00000000  00005f30  00       00  000  01  .debug_line
[  25] DEBUG      00000000  00001090  00       00  000  04  .debug_frame
[  26] DEBUG      00000000  00004209  01       00  000  01  .debug_str
[  27] DEBUG      00000000  00008d45  00       00  000  01  .debug_loc
[  28] DEBUG      00000000  000008d8  00       00  000  01  .debug_ranges
...
Key to Flags: W (write), A (alloc), X (execute)
```

Figure 5.2: Shortened list of sections obtained from the MIPS-ELF file generated from the source code in Figure 5.1.

As we can see, the debugging information is more than three times larger than the rest of the application. Furthermore, the debugging information is stored in a compressed binary form since the second version of the DWARF standard. Otherwise, the size of DWARF sections would be even larger. The compression is done via so-called abbreviations (section `.debug_abbrev`), usage of data type (U)LEB128 for storing numerical values on a minimal number of bits, usage of bytecode sub-programs for storing information tables, etc.

The content of these debug sections is stored in a binary form defined in DWARF standard [18]. It contains several types of DWARF low-level description elements (e.g. `DW_AT`, DW_TAG, DW_OP, DW_CFA) that have to be assembled together in order to obtain a high-level information, such as function name or type of variable. For example, in a shortened listing in Figure 5.3, we can see DWARF elements describing information about the source-code origin (e.g. file name, compiler name, programming language), description of the main() function (e.g. start and end machine-code address, position in the source file), and definition of the return type of this function (int). We can also notice the aforementioned DWARF abbreviation. In this case, both function definition and definition of its data type are abbreviated (43a and 464). The second abbreviation is used within the first one, which creates a hierarchical structure of DWARF description elements.

```
Compilation Unit @ 3f1:
    Length:         140
    Version:        2
    Abbrev Offset:  348
    Pointer Size:   4
<0><3fc>: Abbrev Number: 1 (DW_TAG_compile_unit)
    DW_AT_producer    : (string, offset: 0x26a): GNU C 4.3.5 -g
    DW_AT_language    : 1 (ANSI C)
    DW_AT_name        : (indirect string, offset: 0x2f9): hello.c
    DW_AT_comp_dir    : /
    DW_AT_low_pc      : 0x8900368
    DW_AT_high_pc     : 0x89003a4
    DW_AT_stmt_list   : 0xe8

// ...
// Definition of a function return-value type (abbreviation 43a).

<1><43a>: Abbrev Number: 4 (DW_TAG_base_type)
    DW_AT_byte_size   : 4
    DW_AT_encoding    : 5   (signed)
    DW_AT_name        : int

// ...
// Definition of the main() function (abbreviation 464).

<1><464>: Abbrev Number: 5 (DW_TAG_subprogram)
    DW_AT_external    : 1
    DW_AT_name        : (indirect string, offset: 0x285): main
    DW_AT_decl_file   : 1
    DW_AT_decl_line   : 3
    DW_AT_type        : <43a>
    DW_AT_low_pc      : 0x8900368
    DW_AT_high_pc     : 0x89003a4
    DW_AT_MIPS_fde    : 0x60
    DW_AT_frame_base  : 0x18f (location list)
```

Figure 5.3: Simplified DWARF debugging info for a program listed in Figure 5.1.

Parsing of this DWARF information is non-trivial because the DWARF standard [18] 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 (debugger, decompiler, etc.).

Parsing of debugging information from an object file to structures defined in the DWARF specification is done by the `libdwarf` library[54]. As we noticed, the original `libdwarf` library is using the `libelf`[55] library to access sections of interest in ELF binaries. However, the input of most of our tools is in the uniform COFF-based EFF[56]. In order to parse information from such files, we exploited `libdwarf` object-access capabilities and provided a new interface by using our internal object manipulation library. The overview of this library is depicted in Figure 5.4.

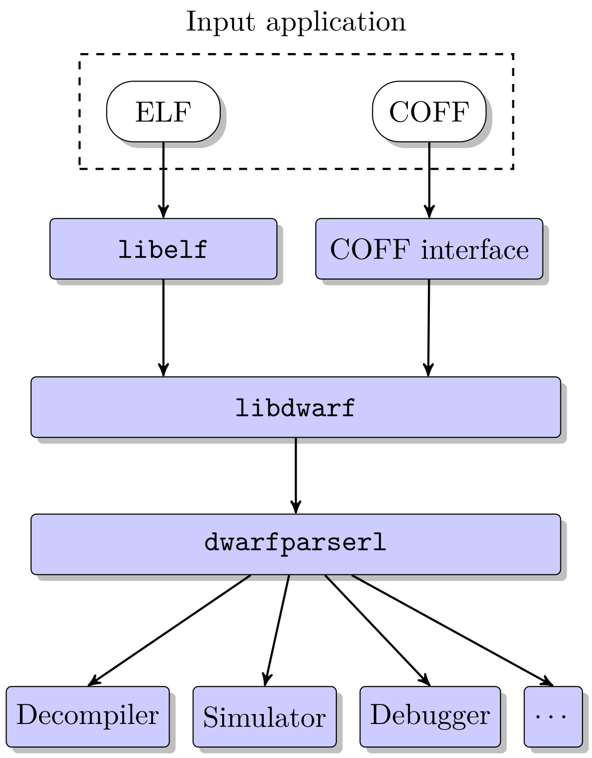


Figure 5.4: Structure of the `dwarfparserl` library.

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. Each of these objects contains complete information about itself and about all the other objects it owns (e.g. local variables in a function). In this way, the information that was in a low-level representation scattered among multiple DIEs is grouped together

[54] http://www.prevanders.net/dwarf.html
[55] http://www.mr511.de/software/
[56] Such COFF file can be created either via the Lissom C compiler or via a generic EFF converter described in Chapter 6.

allowing a very natural and easy way of processing complex DWARF structures. It should be mentioned that `dwarfparserl` is not limited to a particular DWARF version or target architecture. Its brief overview can be found in Figure 5.5.

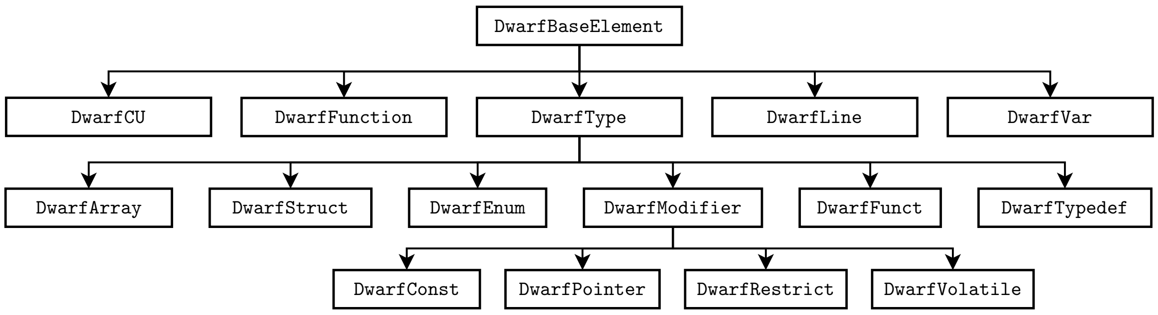


Figure 5.5: Features of the `dwarfparserl` library.

It supports high-level representation of the following properties derived from the `Dwarf BaseElement`:

- `DwarCU` – DWARF compilation unit (i.e. information about each module);
- `DwarFunction` – information about a particular function (name, location, type, etc.);
- `DwarLine` – information about source-code line (and column) mapping;
- `DwarVar` – information about local and global variables;
- `DwarType` – information about the following data types (e.g. variables, return values, arguments):
  - `DwarArray` – array data type;
  - `DwarFunct` – function data type;
  - `DwarEnum` – enumeration;
  - `DwarStruct` – structure data type;
  - `DwarTypedef` – user defined type;
  - `DwarModifier` – the following type modifiers:
    * `DwarPointer` – pointer;
    * `DwarConst` – `const`;
    * `DwarRestrict` – `restrict`;
    * `DwarVolatile` – `volatile`.

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. At the moment, processing of PDB files is important for us only during decompilation of files generated by Microsoft compilers. Our other tools do not utilize information from this format at present because they use DWARF information generated from the Lissom compiler, see Subsection 4.7.3.

The only available official toolkit for PDB parsing—DIA SDK—is unusable for our aims because of the limitations discussed in Subsection 3.3.2. Therefore, it was necessary to reverse-engineer the PDB format and analyse its internal structures firstly. For this purpose, we reused an existing unofficial PDB analyser[57]. Afterwards, we were able to create our own PDB-parsing library.

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. Furthermore, each module has its own stream with information about functions, local and global variables, and line number information within the module, see Figure 5.6.

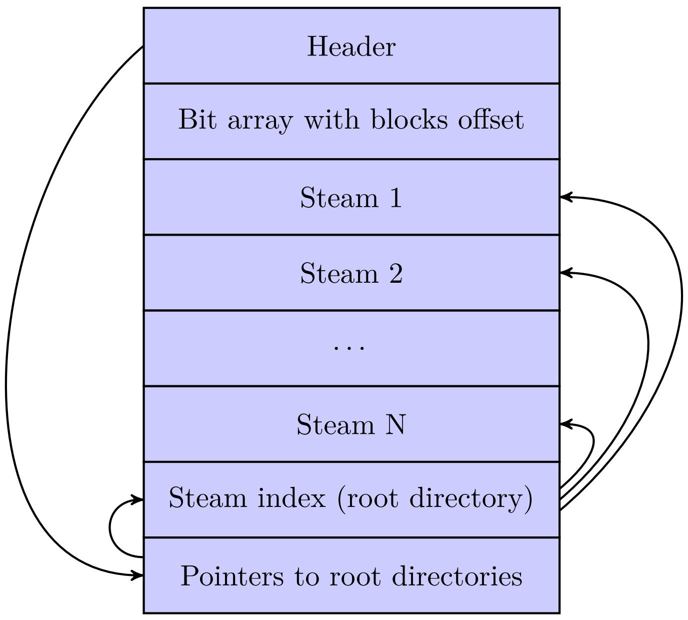

Header
Bit array with blocks offset
Steam 1
Steam 2
...
Steam N
Steam index (root directory)
Pointers to root directories

Figure 5.6: A simplified structure of a PDB file.

This is an advantage of PDB because it allows incremental builds that only attach new information at the end of file without a need to rebuild existing parts. The PDB header contains information about application version, size of allocation block, number of streams, and a pointer to stream root directory. At present, we are aware of two versions of PDB – 2.00 and 7.00 (this version has been used since WindowsXP until now), but their structure is very similar.

The most important streams are:

- *Stream 1* – it is used for identification of the file. It contains time-stamp and unique GUID number, which matches the application's executable file;
- *Stream 2* (TPI) – it contains definitions of user data types;
- *Stream 3* (DBI) – this stream contains a list of all modules and libraries used during linking;

[57] http://code.google.com/p/pdbparser

- *Stream 4* – this stream contains multiple symbol tables (differs based on a scope) and it supports search in these tables;

- *Other streams* – in the remaining streams, we can find information about functions, modules, used compiler and programming language, local and global variables, etc. It should be noted that there are still several streams not covered by our analysis.

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. We can use the root directory for this task, which is located at the end of the PDB file. This directory stores size and the indexes of used data blocks for each particular stream. (2) The main stream 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. For example, we can extract and process an address, length, return type, arguments, local variables, and a line number information (i.e. mapping between machine-code and HLL code location) for each function described in the PDB file.

The meaning of many data entries depends on a context (i.e. the previous data entries). Moreover, the information stored in a particular stream is often interconnected with another stream. For example, each data type has an index to its definition—base types (e.g. `int`, `float`) have a predefined index, but the user types are defined in a type information stream.

By using the `pdbparserl`, we are able to obtain information similar[58] to the DWARF format as depicted in Figure 5.5.

At present, we are able to extract and utilize most of the PDB streams, but a few remaining streams are still unknown for us. As well as the DWARF parser, the PDB parser is not limited to a particular target architecture (e.g. ARM, Intel x86).

5.2 Translated Simulator

The methodology of processor design (i.e. design space exploration) has significantly changed in the last two decades [61]. Firstly, the methodology based on an iterative process was very popular. In this approach, the processor designers create the informal processor design description. Afterwards, they implement all the necessary tools for the processor programming and simulation, the so-called *toolchain* (i.e. assembler, simulator, debugger). Then, they are able to start testing their processor design. However, this process is very time and money consuming because the designers have to change the design and re-implement all tools based on results from the testing phase (e.g. if a bug was found). Furthermore, additional testing must be performed among the tool-chain itself because of consistency assurance among the tools.

A much more effective way of design space exploration is by using ADLs as we mentioned in Chapter 4. The designer describes the processor in a selected ADL and the complete toolchain is automatically generated based on the processor description of this particular ADL. This leads to the shorter design phases and the final design is created in a fraction of the original time. Furthermore, automatically generated tools are usually based on formal models which often imply better chances for processor design verification.

[58]Some of the DWARF information, such as column number mapping or compilation unit information, is not available in PDB.

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.

The simulator can also produce the profiling information, which can (together with debugging results) aid design space exploration and the processor and target application optimization. For further reading about this topic, see [72] where the existing profiling tool is discussed.

As we briefly mentioned in Section 4.6, 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* [71]. 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. For example, instructions within a loop are fetched and decoded several times in the simulated application, although they were not changed. On the other hand, the simulator itself is not dependent on the simulated application, and furthermore, the self-modifying code is supported out of the box. The interpreted simulator creation is also relatively fast.

If the developer wants to increase the speed of a simulation, he or she can use the second type of simulator, the *compiled simulator* [70]. 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 (also known as *static compiled simulator*) is dependent on the simulated application and the self-modifying code is not supported. Nevertheless, the speed of the simulator can be several times faster than the interpreted simulator.

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. Thanks to the first part (hot-spots location) the speed of a just-in-time compiled simulator is slower than the speed of a static compiled simulator. Still, it can be several times faster than the speed of an interpreted simulator. The compiled simulator creation can take more time than the creation of an interpreted simulator (especially a just-in-time compiled simulator).

The interpreted and compiled simulator can have two levels of accuracy. They can be either instruction-accurate or cycle-accurate. During the instruction-accurate simulation the basic step of the simulation is a single instruction (i.e. one instruction is executed at the time). This simulation type has a very good performance, but it is quite far from the hardware representation, since the whole micro-architecture is not simulated at all. On the other hand, during the cycle-accurate simulation, the basic step is a single clock cycle. This level is very close to the hardware because the micro-architecture is simulated (e.g. pipeline, buses). However, the simulation itself is slower.

From the debugging point of view, the simulation speed is crucial for fast design space exploration; on the other hand, the simulation of a micro-architecture is mostly ineffective in this type of usage, because the micro-architecture is being created after the processor design is stable enough.

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 16) in the target application is used[59] 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 later simulator type, JIT translated simulator, is similar to static translated simulator, but it can also be used for simulation of special applications, such as applications with self-modifying code or applications for systems with external memories. Both simulator types are automatically generated based on the instruction-accurate ISAC processor models in a short time, which is another advantage. They are used for a fast simulation of target applications and for obtaining information for subsequent optimization phases.

In the rest of this section, we present a concept and implementation of these two new simulator types. Firstly, we describe the static translated simulator and the JIT translated simulator is presented afterwards. Both of these simulator types exploit the aforementioned debugging information (e.g. information about basic blocks within target application generated by an HLL compiler) contained within the debugged applications for its smoother running. As we will discuss in Chapter 7, both of them can achieve a much higher simulation speed than the previous simulator types used within the Lissom project; furthermore, they are fully competitive to other projects (xADL).

The following notation is used in this section. A *target C compiler* is the retargetable C compiler developed within the Lissom project [36]. 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[60]. 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-

[59]For this reason, the translated simulators are also referred as *basic-block accurate*.
[60]The ISAC processor description has to be on the instruction-accurate level.

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 [69] for details.

The analyser itself is based on the following formalisms: *enhanced lazy finite automata* (Definition 13) and *two-way coupled finite automata* (Definition 14). 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 automaton for $i = 1, 2$, and h is a bijective mapping from R_1 to R_2, see Definition 14. 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 `BEHAVIOR` and `EXPRESSION` sections, which are taken from the ISAC processor description (i.e. from the behavioral description of particular instructions). The content of the `BEHAVIOR` section is changed in a way that the constants, which are represented by attributes, are replaced by their evaluation (values are obtained from the automaton M_1 during translation). Furthermore, each statement is encapsulated, so the C code is only printed into a file, not executed.

We can illustrate this concept on a simplified ISAC model depicted in Figure 5.7. This model contains two operations. The first operation (`imm8`) describes an 8-bit immediate operand using an 8-bit attribute. The second operation (`move_acc`) describes the instruction “move to accumulator” and it uses result of the previous operation (i.e. it uses result from the `EXPRESSION` section, which is the value of an immediate operand). More information can be found in [52].

```
// ...
// Operation with one attribute attr for an 8-bit operand
OPERATION imm8 {
    ASSEMBLER { attr=#U };
    CODING { attr=0bx[8] };
    EXPRESSION { attr; };
}

// Operation describing assignment into accumulator
OPERATION move_acc {
    INSTANCE imm8;
    ASSEMBLER { "move_acc" imm8 };
    CODING { 0b0101 imm8 };
    BEHAVIOR { acc = imm8; };
}
```

Figure 5.7: Example of an ISAC model with two operations.

In this example, the content of the `EXPRESSION` section of the `imm8` operation (i.e. C-code `attr;`) is transformed into a statement `fprintf(fp, "imm8 = %d;\n", attr);`, where `fp` represents a handle to a file containing the generated C code of the analyser. Furthermore, the content of the `BEHAVIOR` section of the `move_acc` operation (i.e. C-code `acc = imm8;`) is transformed into another statement `fprintf(fp, "acc = imm8;\n");`. A simplified version of the two-ways coupled finite automaton for this analyser is illustrated in Figure 5.8.

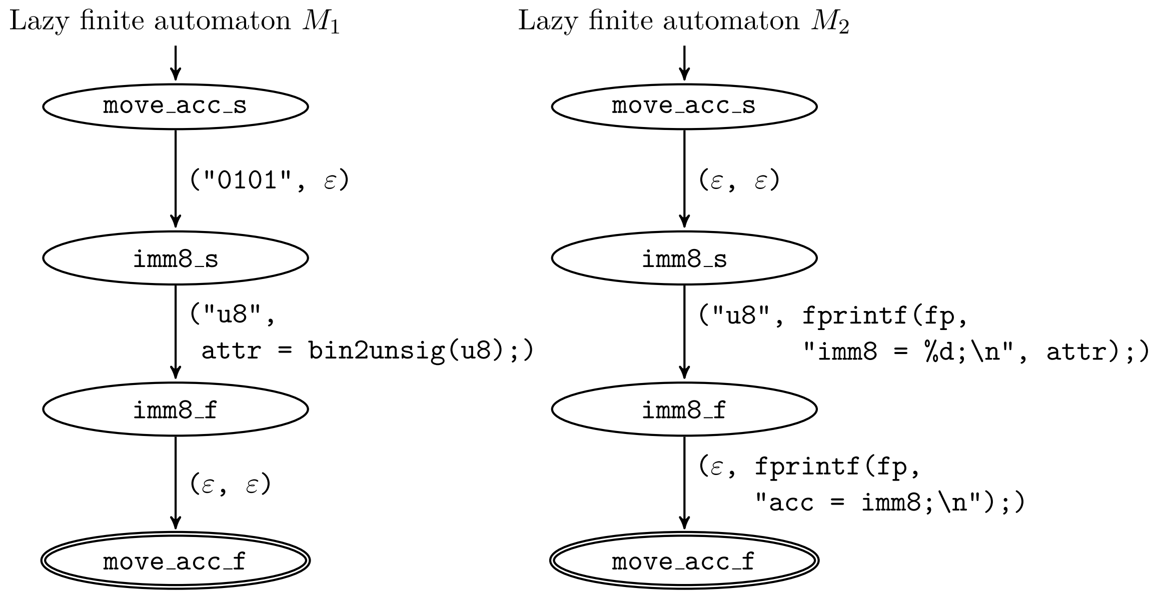

Lazy finite automaton M_1
move_acc_s
("0101", ε)
imm8_s
("u8", attr = bin2unsig(u8);)
imm8_f
(ε, ε)
move_acc_f
Lazy finite automaton M_2
move_acc_s
(ε, ε)
imm8_s
("u8", fprintf(fp, "imm8 = %d;\n", attr);)
imm8_f
(ε, fprintf(fp, "acc = imm8;\n");)
move_acc_f

Figure 5.8: The simplified two-way coupled finite automaton for analyser specified in Figure 5.7.

Each edge of this automaton has a label containg two items – the input symbol and the semantic action. For example, the input symbol `u8` denotes an 8-bit immediate operand. 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)[61]. 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 as illustrated in Figure 5.8.

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.

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

(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, an improved approach is used (it is described in the following text).

As an example, we can assume two instructions in a sequence in the target application, where the second instruction uses the computed values of the first one (e.g. value of some variable). In the straight-way approach, the host compiler has to to move the output from the host-processor registers to the higher level of memory (e.g. cache or even RAM) because its `case` is ended by a `break` statement. However, if there is no `break` or `case` statement between those target instructions and the host compiler can leave the output in the host processor registers, which will improve the simulation performance. This is the reason, why the basic-block addresses are necessary. Furthermore, the analyser generates a `case` construction only when the address of an actual instruction is equal to a start address of a basic block. Analogically, the `break` statement is generated only if the address of the currently processed instruction is equal to an end address of a basic block.

For this task, we can effectively use the DWARF debugging information generated by the target C compiler. This information is stored within the simulated application. 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 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 5.9. 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).

The instructions of the first segment are translated to a C code depicted in Figure 5.10. Behavior of instructions from this segment is described in function `function_0_1023()`. Its `switch` statement is controlled by the value of the program counter and the first `case` represents the first basic block starting at address `0`. As we can see, there is no `break` statement for address 0 (i.e. it is not an ending address) and there is no `case` for the address 4 (i.e. it is not a starting address). Therefore, the code from `BEHAVIOR` section of the synchronization operation `main` is inlined there.

Furthermore, the whole target application is represented by several functions (such as `function_0_1023()`) that are stored in a table. The index (key) to this table is computed by the bitwise shift to the right of the instruction-address value.

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 trans-

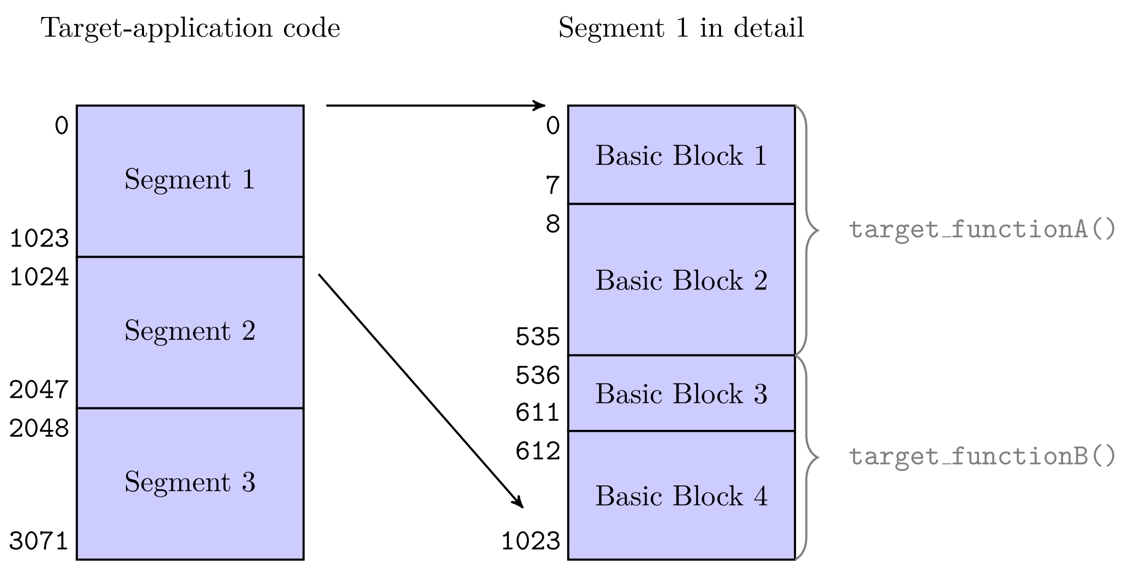


Figure 5.9: Example of an address-space partitioning in the translated simulator.

```
int function_0_1023(pc_t pc)
{
    switch(pc)
    {
        case 0:
            // The first basic block.
            // Behavior of the first instruction.
            // ...

            // Synchronization.
            main_behavior();

            // Behavior of the second instruction.
            // ...
            break;

        case 8:
            // The second basic block.
            // ...

        default:
            return FAILURE;
    }
    return SUCCESS;
}
```

Figure 5.10: Example of a translated-simulator code for Segment 1 from Figure 5.9.

formation 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. if the segment size is 1024, then the number of bits is equal to $log_2(1024) = 10$). This principle is similar to the approach used within the compiled simulator [70]. 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.

The functions are called from the main simulation loop. At the beginning of the main loop, there is also initiation via the aforementioned `BEHAVIOR` section of the `main` operation in the processor model. This function takes care of a program counter incrementation, etc.

(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 run of the static translated simulator can be seen in the Figure 5.11. The `fnc_table` variable is the table with functions. The table key is the variable `spc`, which holds the shifted address of an instruction. The variable `sgt_changed` is set, if the target program is changed within the particular segment.

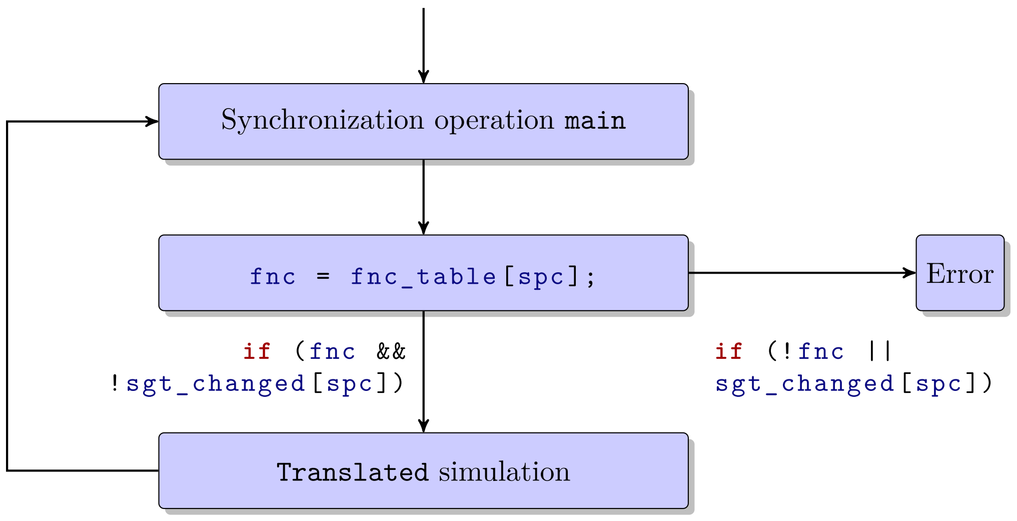

Synchronization operation main
fnc = fnc_table[spc];
Error
if (fnc && !sgt_changed[spc])
if (!fnc || sgt_changed[spc])
Translated simulation

Figure 5.11: Principle of the static translated simulator.

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 [70]. 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 [3]). 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 [13]).

Figure 5.12 depicts the run-time of the JIT translated simulator. The variables `spc`, `fnc_table`, and `sgt_changed` have the same meaning as in the static translated simulator. Furthermore, the variable `sgt_treshold` is set whenever a particular segment is hit and should be recompiled. The dashed arrows denote a parallel execution (i.e. the simulation is not stopped during the segment recompilation).

In conclusion, the JIT translated simulator is similar to the static translated simulator (i.e. it is very fast and it needs additional info from compiler), but it differs in several aspects. The JIT translated simulator can be used instantly, because it is independent of the target application. The target application is recompiled piecewise based on the found hot-spots. Therefore, only parts of the (possibly huge) application are compiled. Furthermore, the simulation of applications with SMC is possible. In this case, the simulator loses information about the basic blocks, but it is still able to continue in the compiled-simulation mode. This unique solution ensures very fast simulation of such applications.

The combination of all simulator types makes the just-in-time translated simulator very useful for ASIP design and target application development. It can also serve as a simulation platform for the source-level debugger.

Furthermore, the target application can be optimized by using the profiling information obtained during simulation[62], such as the source-code coverage, instruction-set coverage, or other statistics (e.g. the most executed functions, the functions causing the most cache misses). The simulator is also capable of generating a call-graph capturing the function calls. All of the mentioned statistics help the developer in the optimization of the target application, see [72] for more details.

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 [30]. The source-level debugger is usually implemented as an extension to an existing instruction-level

[62]The author has no contribution in this profiling mode.

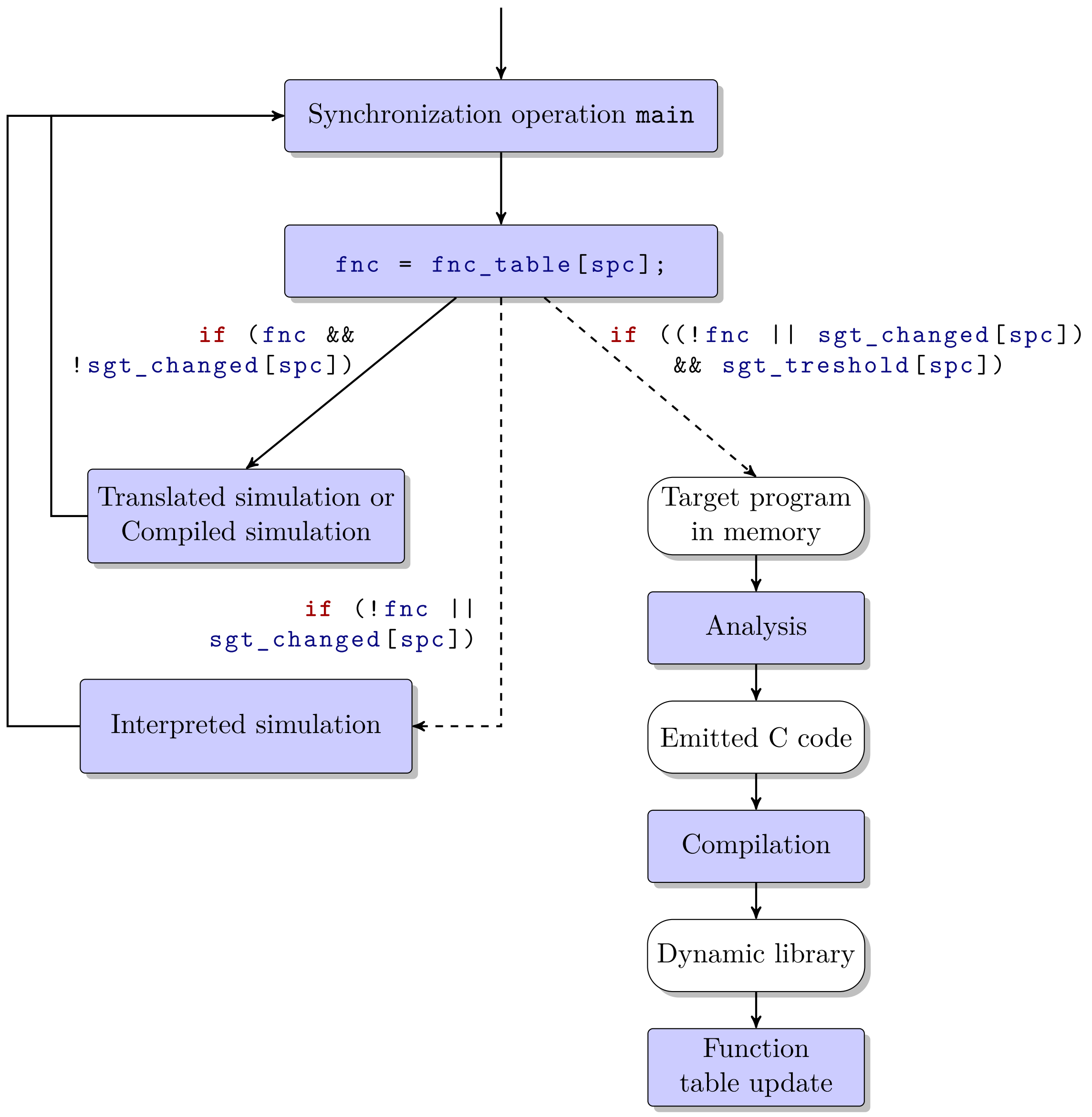

Synchronization operation main
fnc = fnc_table[spc];
if (fnc && !sgt_changed[spc])
if ((!fnc || sgt_changed[spc]) && sgt_treshold[spc])
Translated simulation or Compiled simulation
if (!fnc || sgt_changed[spc])
Interpreted simulation
Target program in memory
Analysis
Emitted C code
Compilation
Dynamic library
Function table update

Figure 5.12: Principle of the JIT translated simulator.

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 general, code debugging is not an easy task. Therefore, the user needs a complex context-aware information about the state of debuggee, to be able to change this state, and to have an overview of target environment. The source-level debugger should have at least the following features (see [77] for more details):

- *Detection of source-code position* – this feature implements detection and visualization of actual position within the statements of the original source code. The basic variant detects lines; advanced debuggers can detect even columns or particular expressions within the line;

- *Program control* – the program execution has to be controlled via breakpoints;
- *Stack backtrace* – the stack backtrace tells the user how the program got into a particular code position (e.g. sequence of function calls, their arguments);
- *Inspecting data and variables* — this feature implements actual evaluation of variables used in the original HLL source code;
- *Processor state* – it is necessary to provide additional information related to the instruction-level debugging (e.g. content of memory, inspection of registers);
- *Interaction with the user* – the user interface (UI) can be done either through a command-line interface (CLI) or via a graphical user interface (GUI).

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 briefly describe the existing debugger of the Lissom project as well as its interconnection with the simulation platform, which represents a background of the proposed source-level debugger. Afterwards, in Subsection 5.3.2, we mention the process of debugging-information generation within the Lissom C compiler. Implementation of the source-level debugger is presented in Subsection 5.3.3.

5.3.1 Lissom Three-Layer Architecture

As we already mentioned in Subsection 4.7.3, the Lissom debugger originally supported the instruction level and micro-architectural level only. Normally, the debugged application is running in one of the supported simulator types (i.e. interpreted, compiled, translated). Furthermore, the debugger also supports debugging of the target application on real hardware. However, this feature refers to the *on-chip* debugging, which is beyond the scope of this thesis (the author has no contribution on it). For more details about this topic, see [113] instead. In the rest of this section, we assume that the debugged application is simulated.

As well as the other tools, the debugger is retargetable and it is automatically generated based on the ISAC model. Actually, the generated debugger is generated as a control part of the generated simulator and the debugger controls execution of the debuggee. In other words, the debugger and simulator together form one application. Furthermore, it is important to mention the architecture of the simulation/debugging platform that is used within the Lissom project. This architecture[63] is illustrated in Figure 5.13.

As we can see, this architecture consists of three layers. The top-most *presentation layer* serves the user for presenting the simulation/debugging state and for interaction. It is implemented either as GUI (Eclipse[64] perspective with a support of visualized debugging) or command-line interface (e.g. debugging in a plain GDB-like style). In both modes, the user is able to control the debugged application as well as to obtain information about its run-time state. Furthermore, the presentation layer communicates with the *middle layer* via an internal TCP/IP-based protocol. As is evident from its name, the middle layer serves as a mediator between user (i.e. presentation layer) and the simulator/debugger (i.e. simulation

[63]It should be noted that the complete architecture is more complex and it is also used for interconnection with other tools than the simulator.

[64]https://www.eclipse.org/

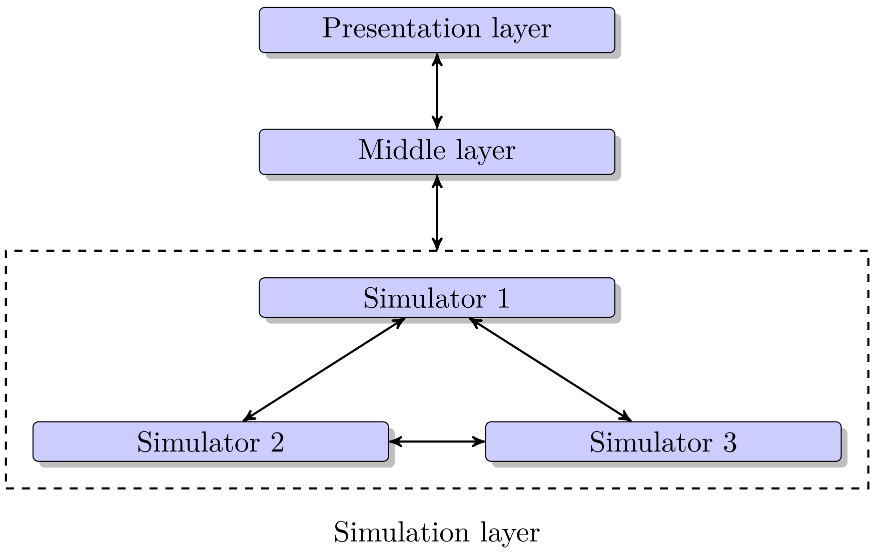


Figure 5.13: Simplified architecture of the Lissom simulation platform.

layer). For example, it can pass a command from user to debugger (e.g. setting a breakpoint) as well as information about the register value that is passed in the opposite direction.

Finally, the *simulation layer* may contain one or more (in a case of multiprocessor simulation that is also supported) simulators that are operated by the commands from the middle layer. Furthermore, the architecture is intended to be distributable. Therefore, the simulation can run on a different machine than the user interface of the presentation layer.

5.3.2 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. As we discussed in Section 3.3, we can find multiple debugging formats with different features. 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 extend this compiler toolchain (e.g. C compiler front-end and back-end, assembler, and linker) for proper DWARF information generation and handling. For example, we had to disable emission of the LLVM-specific attributes that are not included in the official DWARF standard [18] (those with prefix `DW_AT_APPLE`[65]) and that are not supported by the `libdwarf` library (see Section 5.1).

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. For this reason, it was necessary to add the support of several DWARF directives within the assembler tool. An example of such directives is presented in Figure 5.14.

In this example, we can see an assembly-code snippet related to the `main()` function from the source code listed in Figure 5.1. If we focus on its directives (i.e. the lines beginning

[65]LLVM is an integral part of Apple's development tools and it is supported by Apple.

```
main:
    .file   1 "hello.c"
    .loc    1 3 0                   ; hello.c:3:0
    pushl   %ebp
    movl    %esp, %ebp
    subl    $24, %esp
    leal    .Lstr, %eax
    movl    $0, -4(%ebp)
    .loc    1 4 2 prologue_end      ; hello.c:4:2
    movl    %eax, (%esp)
    calll   printf
    movl    $0, %ecx
    .loc    1 5 2                   ; hello.c:5:2
    movl    %eax, -8(%ebp)
    movl    %ecx, %eax
    addl    $24, %esp
    popl    %ebp
    retl
```

Figure 5.14: A snippet of an assembler code with DWARF directives obtained from the source code in Figure 5.1 (`clang -S -g`).

with the dot symbol), we can see the `.file` and `.loc` directives. Those are used for creating content of the `.debug_line` section that is essential for mapping between source lines and machine-code instructions. The former type specifies a name of the file with source code. The second one contains information about the relation of the machine instruction on the next line with the original source statement. It contains a triplet of arguments that specifies a file name (a number related to the `.file` directive), line number, and column number[66]. For example the "`.loc 1 5 2`" directive refers to a statement on the second column of the fifth line in the `hello.c` source code.

The assembler has to gather all those directives and store them into the aforementioned `.debug_line` section[67]. This line information can be represented as a table where rows are addresses of machine-code instructions and columns contain information about a position in the original source code. Furthermore, the line information also contains details about the context of instructions within the original source code (e.g. mapping to statements, specification of function prologues and epilogues, position in basic blocks). Simple line-information example of program for ARM architecture is illustrated in Figure 5.15.

```
;address  file  line  column  statement  BB  epilogue  prologue ISA
;---------------------------------------------------------------------
 0x0      f1.c  42    2       yes        no  no        yes      ARM
 0x9      f1.c  44    2       yes        no  no        no       ARM
 0x2c     f2.c  10    4       yes        yes no        yes      Thumb
```

Figure 5.15: An example of a DWARF line information.

[66]The column number is normally set by the LLVM compiler in order to support a more precise debugging. On the other hand, this information is usually ignored by the GCC compiler.

[67]For illustration, this information is also exploited by the translated simulators (see Section 5.2).

However, it is necessary to compress this table because it may be quite large in a case of large files. The compression is implemented by using the state machine, whose *instructions* are described in the DWARF standard [18]. The design of a program for this machine is another of the authors contribution.

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. Therefore, we created a special COFF section `.ccoff_dwarf_reg_map`, which contains this mapping.

The debug information is now stored in the same way as in the ELF file format, i.e. as binary data in named sections [19]. 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.

5.3.3 Implementation

The initial phase of source-level debugging is a process of obtaining and processing the debugging information. In the past, we tried the third-party libraries such as SymtabAPI[68]. However, those libraries are tied to the ELF EFF while processing DWARF, which is not applicable for us because our applications are stored in the COFF format together with the DWARF sections in it. Therefore, 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 [50] and [113].

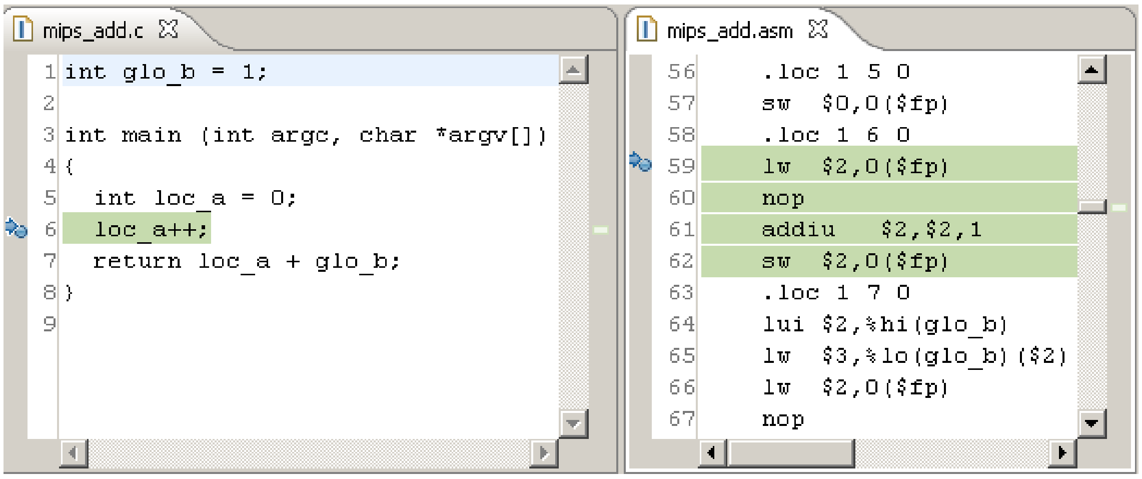
Line-Information Mapping

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 5.16. This is useful for switching between source-level and instruction-level debugging. Within the CLI, the actual code line is printed from the source file to the command line.

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

[68] `http://www.dyninst.org/symtab`



```
int glo_b = 1;

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

```
56    .loc 1 5 0
57    sw   $0,0($fp)
58    .loc 1 6 0
59    lw   $2,0($fp)
60    nop
61    addiu    $2,$2,1
62    sw   $2,0($fp)
63    .loc 1 7 0
64    lui $2,%hi(glo_b)
65    lw   $3,%lo(glo_b)($2)
66    lw   $2,0($fp)
67    nop
```

Figure 5.16: Screenshot of the source-level debugger (lines mapping).

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.

We will describe this feature using an example: the user wants to inspect a value of a local variable in some particular function. This can be done only if the application is stopped (e.g. a breakpoint is hit). Afterwards, a request for variable inspection is sent from the presentation layer over the middle layer to a particular target debugger. There the debugger obtains the current program-counter value, which serves (together with the variable's name) as an index to the DWARF information container. Each such record is mapped to a PC address range and it contains a DWARF expression, which serves for description of the variable location. Such an expression may use multiple values of DWARF registers that have to be evaluated in order to obtain a valid location. Therefore, we have to use the register-mapping information from the aforementioned section `.ccoff_dwarf_reg_map` to get the name of the real processor register for each DWARF register used within an expression and evaluate it. After that, we are able to retrieve the variable's value from the simulated system based on the computed location. However, we have to obtain the DWARF information about the variable's data type at first because it affects the size of data we have to obtain from the simulator as well as its representation once it is retrieved (e.g. signedness, floating point vs. integer number). Finally, the result is sent back to the presentation layer to be displayed to the user.

In the same way, the presentation layer supports evaluation of user expressions that may contain HLL variables. For this task, we reuse the expression solver already available in the original instruction-level debugger.

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[69]. The physical breakpoint can be implemented in several ways in real architectures:

- *Special instruction* – one opcode in an instruction set is dedicated to a special instruction that invokes debug exception (e.g. `int3` in Intel x86, `break` in MIPS);
- *Undefined opcode* – this technique is similar to the previous one, unless the debug exception is invoked by any undefined opcode. This type is for example used on the PowerPC architecture;
- *Hardware breakpoint* – hardware breakpoints are implemented via special debug registers, which contain desired addresses of breakpoints. In each cycle, values of these registers are compared with the program counter and the exception is invoked when matched. Intel x86 (debug registers DR0-DR3) is an example of this type.

However, it may be difficult to implement any of these options in the retargetable debugger because of architecture differences (e.g. some real-life architectures do not have any free opcode for physical breakpoint implementation). Therefore, we have used another approach from the original instruction-level debugger. In this approach, we control a breakpoint hit on the simulation-platform level by comparing the program counter with a list of breakpoint addresses. This solution is independent of a target platform and, based on the experimental results mentioned in Section 7.2, it is about as fast as the the *undefined opcode* approach that we also manually implemented for the testing architectures.

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 proper breakpoint manipulation is crucial for user abstraction. In other words, the user must not have a clue about the internals behind this concept. The algorithms for breakpoint mapping and managing are very similar to algorithms listed in [77].

Furthermore, several types of breakpoints are supported – conditional breakpoints, unconditional breakpoints, controlled breakpoints (i.e. controlled by a number of their execution), watchpoints, etc. However, those types are almost the same (the only difference is their mapping to the HLL constructs instead of instruction-level constructs) as in the instruction-level debugger and the author has no contribution in this field.

Furthermore, it 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). Based on this information and by placing the internal breakpoints in functions, we are able to detect a function call (even a recursion). Afterwards, it is up to the user whether to step in a function or step over it. 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.

[69]For further details about *physical* and *logical* breakpoints see [77].

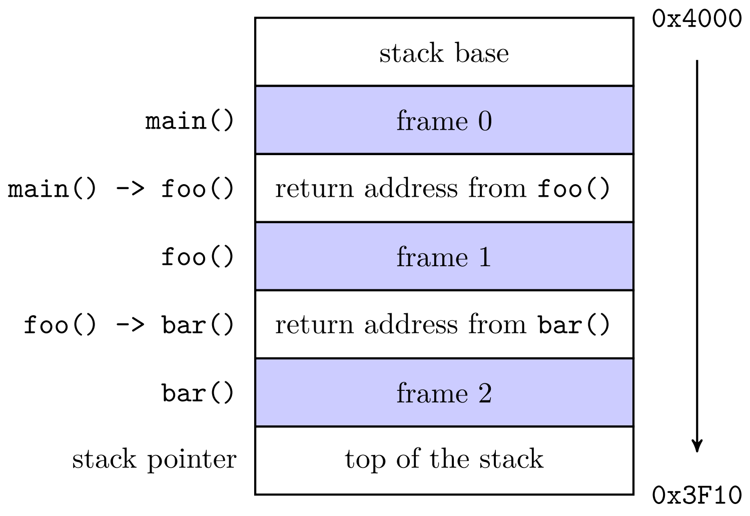
Each item represents one function call together with its arguments and the original calling address. This information is gathered from the call stack (see Figure 5.17) by the unwinding algorithm [77]. Note the stack backtrace structure is platform dependent as well as the application binary interface (ABI) used for calling conventions. Its description must be obtained from the debugging information and from the processor description in the ADL.

```c
int main()
{
    foo(/*...*/);
    // ...
}

int foo(/*...*/)
{
    bar(/*...*/);
    // ...
}

int bar(/*...*/)
{
    // ...
}
```

(a) A simple C application.



(b) Content of the stack.

Figure 5.17: Stack backtrace illustration within the source-level debugger (stack grows downward).

Chapter 6

Retargetable Static Analysis

"Life would be so much easier if we could just look at the source code."

Dave Olson

This chapter is based on publications [107–110, 112, 116, 126, 127, 129, 134] and reports [93, 94]. 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 [103], 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 decompiler consists of four main parts – the preprocessing phase, front-end phase, middle-end phase, and the back-end phase. The complete description of all parts and used methods is beyond the range of this thesis and it can be found in the aforementioned publications. Therefore, we will only give a brief overview with a special focus on those parts related to the author's contribution. Specifically, we will focus on these methods: the preprocessing phase (compiler detection, unpacking, EFF analysis and conversion, etc.), front-end phase (exploitation of debugging information, instruction decoding, control-flow reconstruction, data-flow analysis, etc.), and middle-end phase (instruction-idiom detection and transformation). Contrariwise, the author has a minimal contribution in the design and implementation of methods used within the back-end phase, as well as in several other methods used within the front-end phase, such as data-type reconstruction [53], elimination of statically-linked code [91], or detection of similarity between two files.

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. An overview of these outputs illustrated on examples is presented in Section 6.7. Finally, we present capabilities of this tool on several examples in Appendix A as well as a case study of malware decompilation by using this tool.

At present, a homepage of the implemented retargetable decompiler is available on the following site:

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

Furthermore, the proposed decompiler can be tested via our on-line decompilation service (see [124, 127] for details):

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

6.1 Motivation

Machine-code decompilation[70] is a reverse-engineering discipline focused on reverse compilation [13,20,21]. 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.

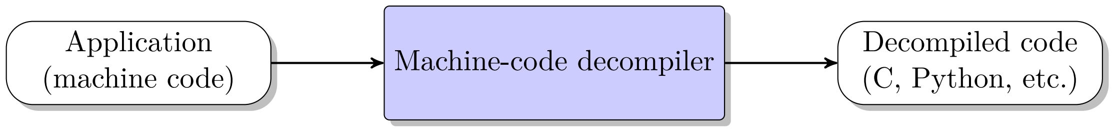


Figure 6.1: Illustration of a decompilation process.

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. For illustration, the compiler has a complete description about the input application specified in a particular programming language. Therefore, it is able to fully understand all of its parts, such as functions, loops, usage of external functions, etc. By using this information, the compiler generates a machine-code representation of this application into an executable file.

However, none of the aforementioned information is normally stored in the executable file because it is not necessary for an application's run-time[71]. The input machine code is often heavily optimized by one of the modern compilers (e.g. GCC, LLVM, 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. Once again, information about application's origin is usually not directly stored and it has to be detected by heuristics.

Furthermore, the process of decompilation is an ambiguous problem equivalent to the *halting problem* for a Turing machine [13]. This applies for example to a problem of separating code and data from the input binary program. There exist several heuristics and

[70]As has been stated in the introduction, we will only focus on the machine-code decompilation in the following text. The other types of decompilation, such as bytecode decompilation, are much easier to achieve and those approaches are already well described in literature [63, 64].

[71]The only exception is a so-called *debug build* in which the compiler also generates debugging information, see Subsection 5.1.1. As we will see, this source of information is very valuable during decompilation process, but the decompiler cannot rely on it because it is available seldom.

algorithms to deal with this problem, but it makes it only partially computable — not in all cases.

6.1.1 Retargetable Decompilation

Despite these problems, there exist a few modern decompilers that are able (at least partially) to deal with such problems, see their description in Section 4.4. These decompilers are tailor-coded for particular target processor architecture(s). However, more attention is paid to retargetable decompilation in recent years. Its goal is to create a tool capable to decompile applications independent of their origin into a uniform code representation. Retargetable decompilation represents an even more difficult task because it must handle all the architecture, operating system, and programming language specific features. Such modification of existing machine-code decompilers may be either not feasible or it will be a very time-consuming task. A basic illustration of this concept is depicted in Figure 6.2.

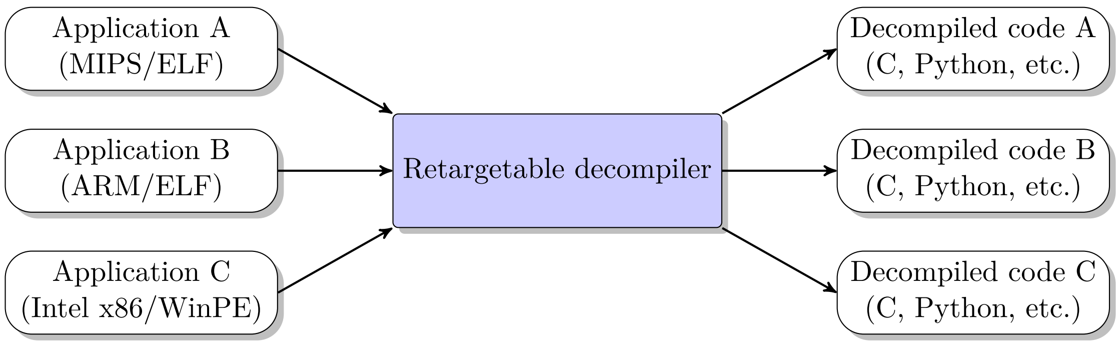


Figure 6.2: Illustration of a retargetable-decompilation process.

The primary advantage of retargetable code decompilation is a fact that the user needs only a minimal knowledge of the target architecture and he or she is able to analyse the target program in a uniform way (i.e. on the level of HLL code). Therefore, there is no need to be familiar with the instruction set of target architecture, its EFF, or other architecture-specific features (e.g. ABI, memory hierarchy, etc.). Further code analysis can be performed on a uniform HLL representation known by the user.

6.1.2 Application of Decompilation

The machine-code decompilation can be applied in different areas of computer science, such as computer and information security, software engineering, etc. Its application differs based on the type of the decompilation output and its usage. For example, the decompiler can automatically generate a complete code representation, which may be used for source-code recovery or code migration (see description in the following paragraph). However, in case of a complex application, the resulting code may be large and difficult to manually analyse. Therefore, the decompiler should be able to generate only smaller code fragments that are easier to analyse for example during malware analysis.

Finally, it is always a great feature of each decompiler whenever it allows user interaction. In this case, it behaves like a code browser while allowing user to affect decompilation results. At this point, it should be stated that decompilation is almost never 100% accurate in comparison to the original code. Therefore, it is up to a user to either manually enhance the resulting code or tweak the settings of the decompiler in order to produce a better code. Some of the most common use cases of decompilation are depicted in Figure 6.3.

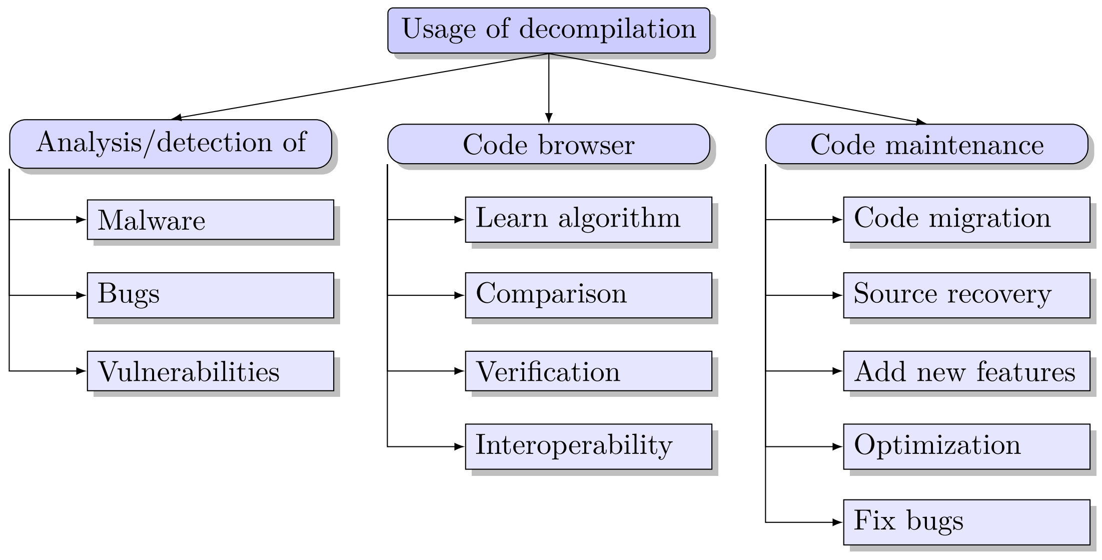


Figure 6.3: Use cases of a decompilation process.

- *Source-code recovery* – Recovery of lost source-codes is one of the most frequently-mentioned usage of decompilation in literature [13, 20]. Fortunately[72], there are not so many users losing their sources and this case is mostly rare at present.
- *Malware analysis* – Contrariwise, the usage of decompilation during malware analysis is more common. The decompiled output, such as C code, CG (Definition 18), or CFG (Definition 17), can give a clear behavioral overview of the (possible) malicious application and the analyst may quickly detect the malicious parts. For example, the infection may be injected into a (large) legitimate application and the manual analysis of the complete file can be very time consuming. On the other hand, analysis of a call-graph can quickly reveal a separation of the injected code from the rest of the application and the analyst can focus just on this part. The same applies to analysis of a statically-linked code, where most of the application's code is represented by standard-library functions (e.g. `printf()`). Studying of these linked functions is merely wasting time[73].
- *Understand the application* – Sometimes it is necessary to understand a functionality of a particular library or application that is created by someone else and distributed without a source-code and proper documentation. For example, the interoperability is a process of discovering the principles of some system in order to enable its cooperation with another program that was originally not intended by the authors of this system. Decompilation of such code can help to quickly discover the piece of code responsible for a desired functionality. In such cases, it's up to a user to respect the intellectual property rights of the code owner because reverse engineering of these applications may be restricted in a licence agreement.

[72] And thanks to existing distributed-revision-control systems and repository providers such as GitHub.
[73] This does not apply to the calls of these functions.

- *Bugs and vulnerabilities detection* – In a reference to Dijkstra's famous quote mentioned in Chapter 5, it is highly unlikely that any real-world non-trivial application does not contain any bug. This may be problematic if a bug or vulnerability is found in a third party software distributed without sources. Reporting or fixing (in a situation when the software is no longer supported by its author) these bugs may imply instruction-level debugging or disassembled-code studying. However, the decompiler can be used for this task in advance via source-level debugging of the decompiled code. This may be a tricky to achieve (because of the missing debugging information) and a user interaction may be necessary.

- *Application comparison* – A machine-code decompiler can be also used for detection of plagiarism and/or forensics analysis of intellectual-property thefts, e.g. enforcing software patents, patent infringements. This happens in situations when an application from one vendor copies principles or even code parts from software that belongs to another vendor. In this situation it is possible to compare decompiled HLL outputs of both applications. Comparison on any lower level (e.g. binary code, disassembled code) is usually not usable (e.g. because of different compilation options producing different binary code, or because of user obfuscation). However, decompilation may filter-out these little differences and it can help to reveal a similarity on the level of call-graphs or CFGs.

- *Re-engineering* – Re-engineering of legacy software to operate on new computing platforms is another common use of decompilation. This process is commonly known as *code migration* and we can find two types of code migration. (1) Within the *binary-code migration*, the user tries to migrate a legacy executable application to a modern system either through conversion into a source-code or directly into a machine-code of this modern system (i.e. *binary translation*). This is a case of aged systems, when the original source-code is no longer available (e.g. its author left the company together with sources). (2) The second type represents a situation when the sources are available, but written in a legacy programming language, and the user wants to migrate them into a modern HLL. This situation is different from the other use-cases mentioned in this section because the machine-code can be circumvented in this process. However, the machine-code decompiler can be still used in this situation, as we presented in [125]. The original code may be compiled into an executable code (or even directly into its intermediate code) with all debugging information available and decompiled as usually.

- *Code optimization and enhancement* – Similarly to the previous scenario, it is possible to use a decompiler to optimize or even enhance the legacy application. For example, it is possible to add new features once the application is decompiled.

- For a description of the other use cases, see [20, 21].

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 [112, 129, 134]. 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 6.4.

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 (the details can be found in [94]).

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. Therefore, we decided to reuse an existing compiler infrastructure for these phases in order to accelerate the decompiler's development. We analysed multiple solutions (e.g. LLVM, GCC, ROSE, COINS, SUIF, Cetus, Zephir) and we decided to choose the LLVM Compiler Infrastructure Project [84]. The reasons are described in detail in [100]. The most important ones are:

- LLVM is far more widespread, used more in both research and industry, and the community around it is bigger than the communities around the other platforms.
- LLVM is designed to be as modular and extensible as possible, it provides many useful programs, like `opt`, `llc`, `clang`, and `lli`, and its LLVM IR can describe many platform-dependent constructs, including support for integers of an arbitrary bit width.
- We can also benefit from the fact that LLVM IR is SSA-based (Definition 21). This form of representation is much more effective and useful for decompiler optimizations than the classical 3-address representation. For example the expression propagation optimization can heavily benefit from the SSA form (see [1]). This optimization is one of the most important ones used in decompiler because it allows us to transform long sequences of individual instruction semantics into more complex HLL statements. For more details about usage of the SSA form in reverse engineering, see [21].
- LLVM provides an online demo[74] that can be used to see how C constructs map into LLVM IR.
- The internals of the LLVM project are well documented[75].

Therefore, we use the LLVM IR format as our internal IR across the front-end, middle-end, and back-end phases.

Now we can continue with a description of the decompilation 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.

[74] `https://kripken.github.io/llvm.js/demo.html`, formerly available at `http://llvm.org/demo/index.cgi`

[75] `http://llvm.org/docs/`

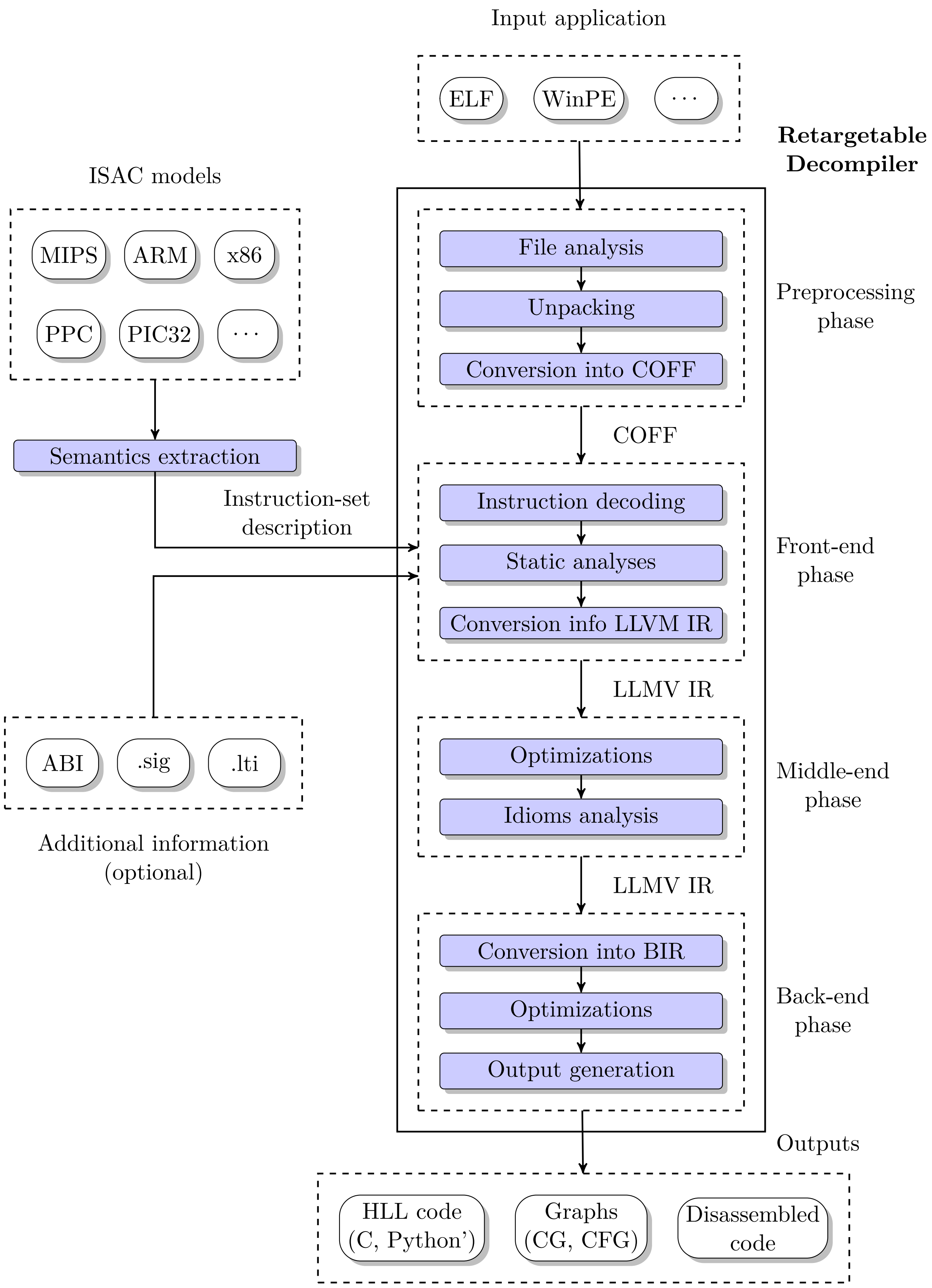


Figure 6.4: Simplified structure of the retargetable decompiler.

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 configured based on the target architecture model in ADL.

Probably the only other attempt to use ADL for creating a retargetable decompiler has been done within the Boomerang decompiler by using the SLED ADL [75] in cooperation with the SSL language [14], as we already noted in Section 4.4. This open-source decompiler is not developed any more and according to its source-code and author's notes, the usage of SLED/SLL was slow and error-prone for more complex processor architectures such as Intel x86.

For our decompiler, we have chosen the ISAC ADL described in Section 4.7. We can find several advantages of ISAC over other ADLs for the purpose of generating a decompiler. (1) This language belongs to mixed architecture description languages, which means that it is general enough to describe both the instruction set and processor resources. (2) The behavior of each instruction is specified as an ANSI C code directly within the ISAC model. This is more readable and easier for the semantics extraction than the description in two different models (i.e. SLED and SSL). An incorrect mapping between instruction encoding and RTL semantics description can also produce errors.

The ISAC model is transformed by a semantics extractor [36], 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. The extracted information obtained from the extractor contains description of semantics, binary encoding, and assembly syntax of each instruction in the instruction set. An example has been depicted in Figure 4.5. 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. These methods are responsible for eliminating statically-linked code, detection of functions and arguments, detection of function calls, reconstruction of data types and many more. 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 (e.g. optimizations of loops, constant propagation, CFG simplifications) 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. However, the LLVM IR is a quite low-level form for HLL representation. Therefore, it is converted into another internal IR called BIR (Back-end IR) first. This form supports the HLL constructions such as functions, loops, conditions, etc. It is much easier to perform the HLL reconstruction over the BIR than over LLVM IR.

Currently, we support C and a Python-like language as the target languages. The latter language is very similar to Python, except a few differences discussed in Section 6.6. The Python output is intended to be used for manual analysis of the decompiled code because it provides the best readability (e.g. no type casting). Therefore, the custom additions to the Python language does not represent an issue because we do not run the decompiled code. Nevertheless, when a compilable output is required, the C-language back-end can be used. For this type, we also focus on syntax correctness of the output (in the ISO/IEC C99 standard). This back-end may be also used when comparing our decompiler with existing solutions (listed in Section 4.4), which mainly support just C. It is also possible to create additional back-ends (e.g. emitting code in the Pascal language).

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[76] 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 6.5. (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 that is based on the XML format. 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.

[76]We will use the term *application* in the this chapter; however, the decompiler also supports decompilation of dynamic-link libraries (i.e. `.dll` files) of shared objects (i.e. `.so` files). Decompilation of static libraries (i.e. `.a` files) is under development at the time of writing this thesis.

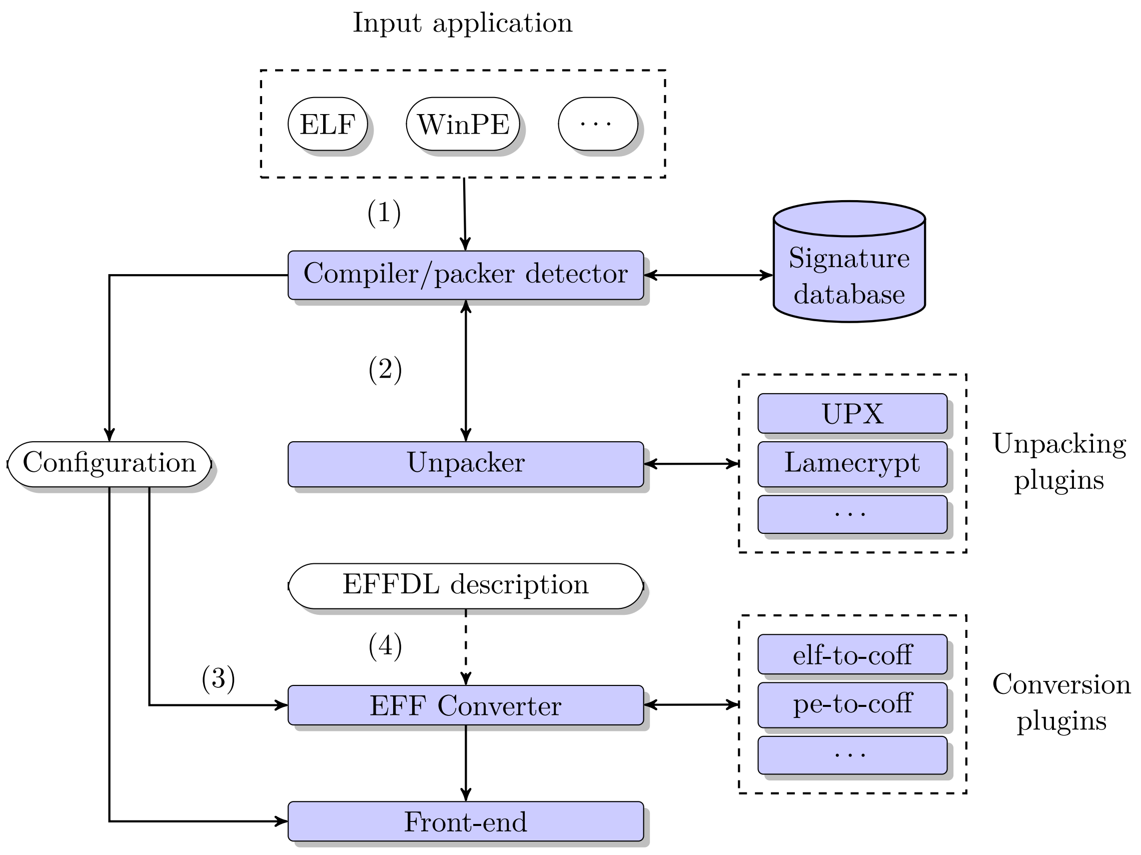


Figure 6.5: Concept of the preprocessing phase.

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

(4) These plugin-based tools have one drawback; whenever we need to describe a new unpacking engine (in a case of unpacker), or a new EFF (in a case of `bintran`), we need to do that by manually implementing a new plugin. However, this is not a truly retargetable approach as is the rest of the decompiler. In the case of unpacker, we can also use one of the existing generic unpackers[77] and it may be quite easy to integrate it into our toolchain (according to our preliminary tests with a proprietary solution of one of our partners). However, as we stated in Section 4.2, there is no such suitable solution that can be applied as a replacement of the `bintran` application. Therefore, we proposed 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.

[77]It is basically a debugger or emulator that executes the packed application and it breaks its execution whenever the application's internal unpacking routine is performed. At this point, the unpacked application's image is dumped from memory to a file. Afterwards, some additional actions has to be done in order to produce a valid executable file (e.g. fixing import table, setting entry-point address).

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). Information about the target processor architecture is extracted from the EFF header (e.g. `e_machine` entry in ELF EFF [86]) and it is used together with other essential information in further steps.

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. We use information about the originally used compiler in multiple front-end analyses, such as detection of the `main()` function, reconstruction of functions and their arguments, detection of statically-linked code, etc. Another typical usage of information about the original compiler is a detection and transformation (i.e. de-optimization) of the instruction idioms that are deeply discussed in Section 6.5. Instruction idiom represents an easy-to-read statement of the HLL code that is transformed by a compiler into one or more machine-code instructions, which behavior is not obvious at the first sight. See [96] for an exhaustive list of the existing idioms.

Motivation

We illustrate this situation using an example depicted as a C language code in Figure 6.6. This program uses an arithmetical expression "`-(a >= 0)`", which is evaluated as `0` whenever the variable `a` is smaller than zero; otherwise, the result is evaluated as `-1`. Note: the following examples are independent of the used optimization level within the presented compilers. All the following compilers generate 32-bit Linux ELF executable files for Intel x86 architecture [38] and the assembly code listings were retrieved via the `objdump` utility.

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    int a;

    scanf("%d", &a);
    // Prints - "0" if the input is smaller than 0
    //        - "-1" otherwise
    printf("%d\n", -(a >= 0));

    return 0;
}
```

Figure 6.6: Instruction-idiom example – source code in C.

Several compilers substitute code described in Figure 6.6 by instruction idioms. Moreover, different compilers generate different idioms. Therefore, it is necessary to distinguish between them. For example, assembly code[78] generated by GCC version 4.0.4 is depicted

[78]Assembly code listed in this chapter is given in the AT&T syntax, see https://en.wikibooks.org/wiki/X86_Assembly/GAS_Syntax.

in Figure 6.7. As we can see, the used idiom is non-trivial and its readability is far from the original expression.

```
; address    hex dump     Intel x86 instruction
;--------------------------------------------
; scanf
; Variable 'a' is stored in %eax
  80483e2:   f7 d0        not   %eax
  80483e4:   c1 e8 1f     shr   $31,%eax
  80483e7:   f7 d8        neg   %eax
; Print result stored in %eax
; printf
```

Figure 6.7: Instruction-idiom example – assembly code generated by GCC 4.0.4 from the source code listed in Figure 6.6.

For comparison, we compiled the same source from Figure 6.6 by the Clang compiler. Output of this compiler is illustrated in Figure 6.8. As we can see, Clang uses an idiom, which is twice as long as the previous one and it is assembled by a different set of instructions. Therefore, it is not possible to implement one generic decompilation analysis. Such a solution will be inaccurate and slow (i.e. detection of all existing idioms no matter on the originally used compiler).

```
; address    hex dump         Intel x86 instruction
;------------------------------------------------
; scanf
; Variable 'a' is stored on stack at -16(%ebp)
  8013bf:    83 7d f0 00      cmpl    $0,-16(%ebp)
  8013c3:    0f 9d c2         setge   %dl
  8013c6:    80 e2 01         and     $1,%dl
  8013c9:    0f b6 f2         movzbl  %dl,%esi
  8013cc:    bf 00 00 00 00   mov     $0,%edi
  8013d1:    29 f7            sub     %esi,%edi
  8013d3:    ;...
  8013d6:    89 7c 24 04      mov     %edi,4(%esp)
; Print result stored on stack at 4(%esp)
; printf
```

Figure 6.8: Instruction-idiom example – assembly code generated by Clang 3.1 from the source code listed in Figure 6.6.

Decompilation of instruction idioms (or other similar constructions) produces correct code; however, without any compiler-specific analysis, this code is hard to read by a human because it is more similar to a machine-code representation than to the original HLL code. Compiler-specific analyses are focused on these issues (e.g. they detect and transform idioms back to a well-readable representation), but the knowledge of the originally used compiler and its version is mandatory.

Figure 6.9 depicts decompilation results for the GCC-compiled code listed in Figure 6.7 (i.e. code generated by GCC 4.0.4). In this figure, we simulate a situation when the information about the used compiler is not available. As we can see, the expression contains bitwise shift and `xor` operators instead of the originally used comparison operator. This makes the decompiled code hard to read.

```c
#include <stdint.h>
#include <stdio.h>

int main(int argc, char **argv)
{
    int v1 = 0;
    scanf("%d", &v1);
    printf("%d\n", -(v1 >> 31 ^ 1));
    return 0;
}
```

Figure 6.9: Decompiled source code from program listed in Figure 6.7. In this case, the decompiler lacks any compiler-specific analysis and the result is hard to read.

Furthermore, it is important to detect the compiler version too. In Figure 6.10, we illustrate that two different versions of the same compiler may generate a different code for the same expression. We use GCC version 3.4.6 and the C code from the Figure 6.6.

```
; address    hex dump              Intel x86 instruction
;-----------------------------------------------------
; scanf
; Variable 'a' is stored on stack at -4(%ebp)
  80483f3:   83 7d fc 00           cmpl $0,-4(%ebp)
  80483f7:   78 09                 js   8048402
  80483f9:   c7 45 f8 ff ff...     movl $-1,-8(%ebp)
  8048400:   eb 07                 jmp  8048409
  8048402:   c7 45 f8 00 00...     movl $0,-8(%ebp)
  8048409:
; Print result stored on stack at -8(%ebp)
; printf
```

Figure 6.10: Instruction-idiom example – assembly code generated by GCC 3.4.6 from the source code listed in Figure 6.6.

In this assembly code snippet, we can see that no instruction idiom was used. The code simply compares the value of a variable with zero and sets the result in a human-readable form. It is clear that the difference between the code generated by the older (Figure 6.10) and the newer version (Figure 6.7) of this compiler is significant. Therefore, we can close this illustration stating that information about the used compiler and its version is important for decompilation.

Signature-based Detection

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 that is similar to one mentioned in Section 4.1. This method is described first. Afterwards, we describe the second method that is based on heuristic detection.

An example of such a start-up code can be seen in Figure 6.11. 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. This signature format is quite similar to formats used by other detectors listed in Section 4.1.

```
; 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 6.11: Start-up code for MinGW GCC v4.6 on Intel x86 (from the object file `crt2.o`).

Our signature format also supports two new features—description of nibble sequences with zero or more occurrences and description of unconditional short jumps. An example of the former one is “(90)”, denoting an optional sequence of `nop`[79] instructions for the Intel x86 architecture. Example for the second one is “#EB”, denoting an unconditional short jump for the same architecture, which size is specified in the next byte; everything between the jump and its destination is skipped. In Figure 6.12, we can find a code snippet covered by a signature “#EB(90)40”.

```
; address    hex dump     Intel x86 instruction
;----------------------------------------------
  00401000: eb 02         jmp short <00401004>
  00401002: xx xx         ; don't care
  00401004: 90            nop
  00401005: 90            nop
  00401006: 40            inc %eax
```

Figure 6.12: Example of an advanced signature format for compiler detection (Intel x86).

These features are useful especially for polymorphic packers [81] producing a large number of different start-up codes (e.g. Obsidium packer). Describing one version of such packer usually needs dozens of classical signatures. However, this number can be significantly reduced by using the aforementioned features.

Signatures within our internal database were created with focus on the detection of the packer’s version. This information is valuable for decompilation because two different versions of the same packer may produce diverse code constructions. The database also contains signatures for non-WinPE platforms; therefore, it is not limited like most of other tools. Finally, new signatures can be automatically created by using our internal tool **getsig** (Get Signature). At least two executable-files generated by the same version of the same tool are mandatory for generation of a new signature. However, presence of multiple files is recommended in order to find all variable nibbles in the start-up code.

[79]The opcode of the `nop` instruction is encoded as `0x90h` on Intel x86 architecture [38].

Heuristic-based Detection

Unfortunately, some polymorphic packers (e.g. Morphine encryptor) cannot be described via extended format of signatures. These packers generate an entirely different start-up routine for each packed file. For instance, an example of three different start-up routines generated by the Morphine encryptor is depicted in Figure 6.13. If we use these three samples to create a new signature, we get a signature with no significant nibbles.

```
1C1C26083EB00F6DFF6DF535BFC7C03C1EC005 ; sample A
7408525566C1C4105D5A51510AC95959F9FC60 ; sample B
510FB6C9770525FFFFFFFF8E2F35983FA2D8B ; sample C

-------------------------------------- ; generated signature
```

Figure 6.13: Example of signatures for the Morphine encryptor.

As we can see, the resulting signature contains only wild-card characters, which is useless for detection. Therefore, in order to support precise detection, we 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, information stored within file header, offset of EP in executable file) and they perform the detection based on a packer-specific behavior. An example of such heuristic is illustrated as a pseudo-code in Figure 6.14—it takes into account the file format, target architecture, offset of EP in file, and information about file sections. These heuristics are implemented directly as a C++ code.

```
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 6.14: 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. Some of these heuristics are depicted in Table 6.1. For a complete list, see [123].

6.3.2 Unpacking

Whenever usage of a packer is detected by the `fileinfo` application, the unpacking part is invoked.

Table 6.1: Overview of heuristics focused on the name and order of sections.

	heuristic
Upack	`sections[0].name == ".Upack"`
ASPack	`sections[last - 1].name == ".aspack" && sections[last].name == ".adata"`
NFO	`numberOfSectionsWithName("NFO") == numberOfSections`
UPX	`numberOfSections == 3 && sections[0].name == "UPX0" && sections[1].name == "UPX1" && (sections[2].name == "UPX2" || sections[2].name == ".rsrc")`
Petite	`numberOfSectionsWithName(".petite") == 1`
MEW v10	`numberOfSections == 2 && section[0].name == ".data" && section[1].name == ".decode"`
NsPack v3.x	`for(i = 0; i < numberOfSections; ++i) sections[i].name == string(".nsp" + numToStr(i))`

Executable-file packing is done for one of these reasons—code compression, code protection, or their combination. The idea of code compression is to minimize the size of distributed files. Roughly speaking, it is done by compressing the file's content (i.e. code, data, symbol tables) and its decompression into memory or into a temporal file during execution.

Code protection can be done by a wide range of techniques (e.g. anti-debugging, anti-dumping, insertion of self-modifying code, interpretation of code in internal virtual machine). It is primarily used on MS Windows, but support of other platforms is on the rise over the last few years (e.g. gzexe and Elfcrypt for Linux, VMProtect for Mac OS X, multi-platform UPX and HASP).

Packers are proclaimed to be used for securing commercial code from cracking; however, they are massively abused by malware authors to avoid anti-virus detection. Decompilation of compressed or protected code is practically impossible, mainly because it is "just" a static code analysis and unpacking is done during run-time. Therefore, it is crucial to solve this issue in order to support decompilation of this kind of code.

UPX is a rare case of packers because it also supports decompression. Unpacking is a very popular discipline of reverse engineering and we can find tools for unpacking many versions of all popular packers (e.g. ASPackDie, tEunlock, UnArmadillo). We can also find unpacking scripts for popular debuggers, like OllyDbg, which do the same job.

Currently, about 80% to 90% of malware is packed [11] and about 10 to 15 new packers are created from existing ones every month [4], more and more often by using polymorhic code generators [81]. In the past, there were several attempts to create generic unpackers (e.g. ProcDump, GUW32), but their results were less accurate than packer-specific unpackers. On the other hand, the creation of single-purpose unpackers from scratch is a time consuming task.

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.

A plugin can be created in two different ways: either it can reverse all the techniques used by the packer and it produces the original file, or the plugin can execute the packed file, wait for its decompression, and dump its unprotected version from memory to file. The first one is harder to create because it takes a lot of time to analyse all the used protection techniques. Its advantage is that unpacking can be done on any platform because the file is not being executed. That is the main disadvantage of the second approach. Such a plugin can be created quickly; however, it must be executed on the same target platform.

At present, we support unpacking of several popular packers like UPX (Linux and Windows), AcidCrypt, and others by using both approaches. These plugins are able to generate valid executable files with removed protections.

After unpacking, the re-generated executable file is re-analysed. In rare cases, a second packer was used and we need to unpack this file once again. Otherwise, the analysis will try to detect the used compiler and its version, and generate a configuration file, which is used by other decompilation tools. This configuration file also contains information about the target architecture, endianness, bitwidth, address of OEP, etc.

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) [110].

For handling ELF, PE, and Mach-O, the Binary File Descriptor (BFD) library is used. BFD was developed by the Cygnus Support company, and currently forms a part of the GNU binutils package, as we discussed in Section 4.2. It is used for unified format manipulation because it supports dozens of EFFs [12]. Furthermore, for some minor actions related to manipulation with EFF-specific features (e.g. detection of `ImageBase` in WinPE), we use the aforementioned `libelf` library for the ELF format and the `PeLib`[80] library for the WinPE format.

During the conversion, the content of an input file is mapped to a BFD internal canonical structure, uniformly characterizing items like sections, section flags, symbols, and architecture type. Afterwards, this form is transformed into a structure suitable for the COFF format via the conversion plugin `bfd-to-coff`, and finally saved in this format by the internal library `objfilelib`, see Figure 6.15. It should be noted, that the inverse conversion (i.e. plugin `coff-to-bfd`) is also possible, and it is used for example for testing the Lissom project's retargetable C compiler listed in Section 4.7.

[80] http://www.pelib.com/index.php

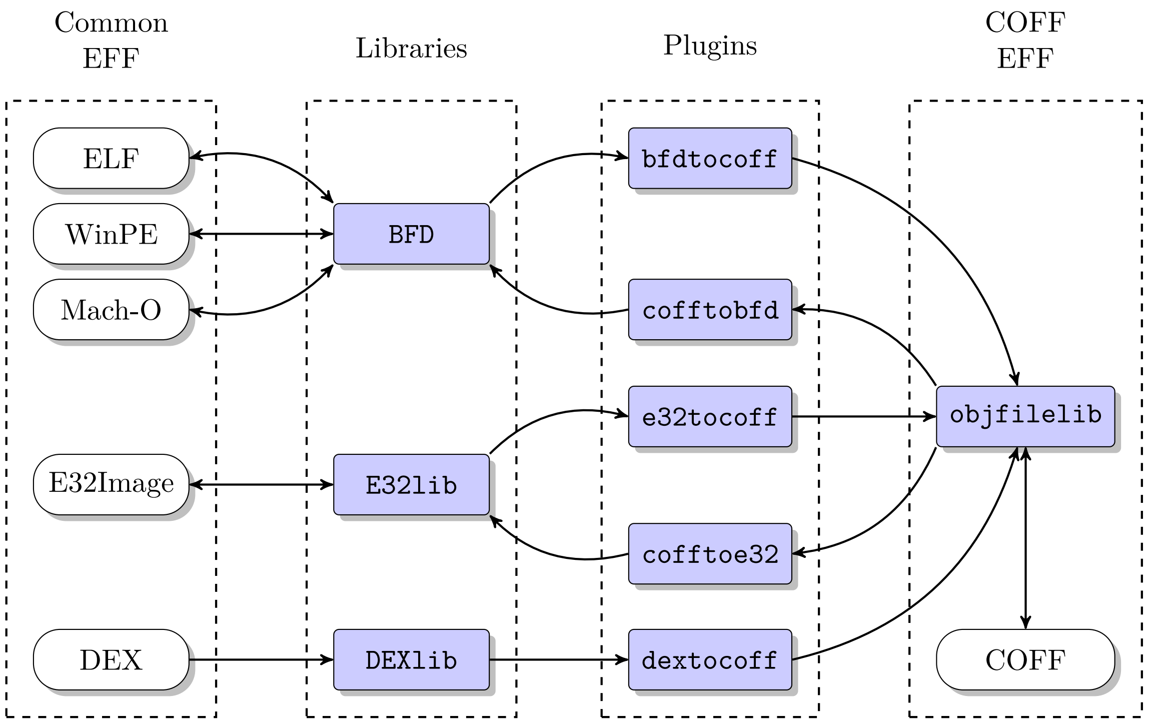


Figure 6.15: Principle of the plugin-based EFF converter `bintran`.

The conversion process can be divided into the following parts:

- *Conversion of the header* – By using the information from the configuration file (generated by `fileinfo`), the proper plugin and BFD target is specified. Afterwards, the plugin converts the fundamental parts of the input-EFF header (e.g. bit-width, endianity, application's EP) into the COFF header. The information, which is not useful during decompilation, is discarded (e.g. information about the original EFF type).

- *Conversion of sections* – The conversion of a section's data (e.g. machine code, data sections, debugging information) is done seamlessly without a need of any other actions. The original content is stored in a binary-encoded form, while the COFF output is textual. Furthermore, each section has several flags characterising its content (e.g. code, BSS) and how it should be treated by the loader during execution (e.g. code section has to be loaded, while a section with debugging information has not). Our COFF standard supports most these flags with a similar behavior. However, it is necessary to correctly map all flag combinations to those supported in COFF.

- *Conversion of symbols* – This part is more tricky because each EFF uses different set of symbol types and their storage. The symbol tables are stored as different special sections in each EFF. For example, the WinPE section `.export` usually contains symbols related to functions defined within the DLL file. Contrariwise, usage of such DLL function within an executable application is done by import symbols within the `.idata` section of this executable file. It would be a clumsy solution to leave the handling of these symbols up to the decompiler because an EFF-specific analysis would be required. We solve this issue by converting these EFF-specific symbol sections into two uniform symbol tables – one for export functions and one for import functions. Each such symbol contains information about its name, ordinal number, and address. The import symbol may also contain the name of the function's library.

- *Conversion of relocations* – This applies to conversions of libraries and archives only because executable files usually do not contain relocations [51]. We can find hundreds of relocation types (they differ for each EFF type and target architecture). However, support of relocations is limited in the COFF format and it is unable to describe some of the rare ones. Therefore, we only convert relocations that have their COFF counterpart. It should be noted that this simplification has only a minimal impact on conversion quality.

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. Their brief description has been given in Section 3.2. However, conversion of these two EFFs creates only a proof-of-the-concept of the converter because decompilation of applications in these formats is not supported at the moment. Symbian is basically no longer used in a mainstream (its development has been discontinued in 2012) and decompilation of such files is no longer needed. Furthermore, the DEX files contain bytecode, which makes them not decompilable by our machine-code decompiler. Anyway, we can find several existing decompilers, such as JD-GUI[81]. Nevertheless, the COFF representations of E32Image and DEX files may be still useful because it may give a human-readable information about the file structure, symbols, etc.

The conversion of the E32Image format is done by the conversion plugins `e32-to-coff` and `coff-to-e32` by using our own library named `E32lib`, which handles E32Image files. We also use the `deflate` library, which is a part of the Symbian SDK[82], for compression and decompression of E32Image files. The main principle of its functionality is very similar to the previously mentioned BFD conversion. As we can see from Figure 6.15, both directions of conversion are supported.

As has been discussed in Section 3.2, the structure of the DEX format differs significantly against other formats. We solved the conversion by transforming all the classes as separate sections containing all the bytecodes of class methods. Beginnings of the method codes are referenced by special symbols. The same principle was used to convert other parts of DEX files. This approach has been implemented as the `dex-to-coff` plugin, which uses the existing `libDEX` library[83]. The other direction of conversion is not implemented yet, since it is not important at the moment.

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 aforementioned plugin-based EFF converter has already been implemented as the `bintran` application and it is used in practice. Its main drawback is the implementation complexity of each newly created plugin because each such plugin has to be written manually either on the top of some existing library or written entirely from scratch. In other words, this part is not as generic (or retargetable) as the other parts of the decompiler.

[81] http://jd.benow.ca/
[82] http://developer.nokia.com/community/wiki/Symbian_C++
[83] https://github.com/android/platform_dalvik/tree/master/libdex

The complexity of existing parsing-libraries differs dramatically based on the target file format and supported features of the library, see Table 6.2. For example, the BFD library supports multiple file formats (more than 50), but it contains more than half a million code lines and its maintenance and extensibility is questionable. On the other hand, there exist lightweight libraries, like ELFIO, containing only a few thousand code lines, but they lack any advanced functionality, such as processing of the parsed files.

Table 6.2: Complexity of several existing EFF parsing libraries.

	lines of code
Binary File Descriptor library[84]	615,856
PeLib[85]	12,220
LibELF[86]	10,930
pyelftools[87]	10,582
ELFIO[88]	3,068

The existing plugins used within the decompiler have different complexity too, see Table 6.3. The first four plugins in the table use third party libraries. Therefore, they are relatively small. On the other hand, the E32Image is built from scratch and it is larger than the others.

Table 6.3: Complexity of Lissom project EFF-conversion plugins.

	lines of code
Android DEX	927
WinPE	2,831
ELF	2,154
Mach-O	2,227
E32Image	9,424

According to our experience, the manual implementation of conversion plugins is slow (in matter of implementation) and prone to errors. This approach is also complicated whenever implementing a non-common EFF (e.g. bFLT, XCEFF, OMF) because there are no suitable existing parsing libraries.

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 (see the description in Section 6.4). 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 (i.e. the user will need to describe

[84]http://sourceware.org/binutils/docs/bfd/
[85]http://www.pelib.com
[86]http://www.mr511.de/software/
[87]https://github.com/eliben/pyelftools
[88]http://elfio.sourceforge.net/

EFF by using several hundred lines without using any external libraries). However, as we will see in the following text, this is a quite challenging task.

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, see Section 3.2 for details. These elements create a context-sensitive behavior (see Definition 6), which is problematic for the design of such an EFF-description language because the theory of computer-language compilation has settled down to the concept of context-free parsing for most of the existing languages during the last sixty years [1, 60].

Within the context-free parsing concept, syntax of programs is usually processed by using automatically generated context-free parsers. Parser generators like YACC, Bison, or ANTLR are able to create a skeleton of target language-specific parser. However, this skeleton has to be enriched by hand-written HLL code implementing semantics checking (so-called semantic actions). This concept is prone to errors and each change of the target language requires re-implementation of the parser (at least its semantic actions).

As we discussed in Section 4.2, the existing languages for EFF description do not suit our needs. Therefore, we introduce our own EFF-description language called EFFDL that is also capable to describe the context-sensitive elements. This language should be able to describe structure of each particular EFF as well as its context-sensitive aspects and it is planned to be used as an alternative to the `bintran` application within our retargetable decompiler. However, this feature is still under development and it was tested only on the ELF format (an example with description of this EFF is given at the end of this subsection).

We specify the EFFDL language by using a grammar denoting this language and we add its brief description. Furthermore, we propose its context parser that is based on the newly created formal models (scattered context grammar with priority function, etc.). However, a detailed description of the context parser is beyond the scope of this thesis and we refer to its detailed description presented in [107].

Finally, this language is designed to be general enough for usage in other retargetable tools (e.g. loaders, disassemblers, debuggers) and the context parser can be used for parsing of different programming languages, not just EFF-description language.

Scattered Context Grammar with Priority

Before we present the EFFDL, we have to define a new formal model that is used for EFFDL description. In [116], we defined a regulated version of the scattered context grammars (see Definitions 9, 10, and 11), 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.

Definition 22. 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*

$$\pi : P \to \mathbb{N}.$$

Definition 23. Let $G = (V, T, P, S, \pi)$ be a (P)SCGP. We say that y directly derives z in (P)SCG G according to the production p, $y \Rightarrow_G z\ [p]$ (or simply $y \Rightarrow_G z$), if and only if:

- $y = u_1 A_1 u_2 \dots u_n \; A_n u_{n+1} \in V^*$,
- $z = u_1 x_1 u_2 \dots u_n x_n u_{n+1} \in V^*$,
- $p = (A_1, \dots, A_n) \to (x_1, \dots, x_n) \in P$, and
- there is no $p' = (A'_1, \dots, A'_n) \to (x'_1, \dots, x'_n) \in P$, such that:
  1. $y = u'_1 A'_1 u'_2 \dots u'_n A'_n u'_{n+1} \in V^*$, and
  2. $\pi(p') > \pi(p)$.

Furthermore, we can define a language generated by (P)SCGP.

Definition 24. A *(propagating) scattered context language with priority* is language generated by a (propagating) scattered context grammar with priority.

The family of (propagating) scattered context languages with priority is denoted by $\mathcal{L}$((P)SCP).

In [116], we provided a formal proof of generative power of these grammars:

Theorem 2. $\mathcal{L}(RE) = \mathcal{L}(SCP)$ and $\mathcal{L}(PSCP) = \mathcal{L}(CS)$, where RE stands for the set of all recursively enumerable languages (see Definition 4).

Finally, we also presented the benefits of this grammar via modelling of VLIW instruction constraints within an HLL compiler, see [115].

Grammar of the EFFDL

In programming language terminology, the grammars (see Definition 2) are used for describing syntax of programming languages. In other words, a grammar creates a core of a programming-language parser. Such a parser handles input programs (written in this language) by using the grammar productions (see Definition 3). A parser can be used within compilers, verification software, or just for syntax checking. Within the classical compilation concept (see [1]), grammar serves only for description of syntax. Therefore, the programming-language semantics have to be described manually (e.g. HLL code performing analysis of parsed code, semantic actions coupled with grammar productions).

In our case, we also use grammar to represent some kind of code — EFF structure. Each particular grammar description specifies one EFF; using this description, we are able to automatically generate the parser of this EFF. This parser will be used as a core of EFF converter to COFF format. Moreover, our grammar is more advanced and it can also describe context-sensitive properties as well as semantic actions on the level of the grammar itself (this is another difference to existing BFF grammars described in Section 4.2 that are based on classical context-free grammars as defined in Definition 7).

The language is designed for a description of the common EFFs (and hopefully the future ones, too). Executable file on parser's input is viewed as a binary stream. Its parsing is done via interconnected analysers that invoke each other whenever it is necessary. Analysers are also able to seek the desired file offset within the stream. This is the major difference to other existing languages and their parsers. The language is not limited to any particular EFF construction, and it is capable to describe optional or scattered parts of the EFFs.

Modified Extended Backus-Naur Form (EBNF) is used for a grammar's syntax description. Terminal symbols are typeset in `red`. The symbol $\sim$ 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 6.16. For clarity, only the most important productions are specified.

```
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 6.16: Simplified grammar of the EFFDL.

The symbol `start` is the start symbol of the grammar (see Definition 2). The keyword `root` denotes the starting analyser, which is executed at the beginning of parsing. Each analyser can be controlled by the begin and end `offsets`. In that case, the analyser executes its job from the beginning offset and it must finish analysis before the stop offset, otherwise it will end as a parsing error. Analysers read desired number of bits from an input stream, see Figure 6.17 for illustration.

The number of bits is specified by an `element` with different sizes (specified by `type`). Elements are continual sequences of bits in an input stream. The value of an element can be skipped (i.e. so-called *don't care* value), enforced (i.e. analyser ends with error if there is an unexpected value on input), or checked by the analyser, see Figure 6.18.

Elements and analysers may contain a list of `attributes`. Attributes contain information about properties such as an element's value or type. They can be used either in semantic actions (e.g. checking of an element) or in context productions. In general, attributes are used for re-referencing previously parsed parts, such as information from EFF header. This context behavior is not common in classical programming language grammars. Therefore, it is possible to use both synthesized and inherited attributes from previously parsed elements within the semantic actions, see [1] for details.

Checking of elements can be done via `semantic_actions`, which are statements of the C/C++ code. Semantic actions can be also used for an interaction with retargetable tools. In our case, they are used mainly for direct COFF generation. For an illustration see Figure 6.19.

Parsing can be also controlled via context `productions`; they are formatted as attributed SCGP productions (see Definition 22); therefore, the number of items within brackets must be the same on both sides (ε-rules are allowed). The nonterminals `id`$_{\texttt{attribute}}$ stand for `element` or `analyser_id` and they are rewritten according to the right-hand-side of those productions. Attributes are also taken into account during derivation. Finally, it is possible to describe the `priority` of each production. A higher value means a higher priority. This is useful whenever we need to perform some actions before any other production

```
// Root (starting) parser of a particular file format XY.
root analyser XY_EFF_parser ()
{
    /*
        It invokes an analyser of the file-header.
        The header is located on the first 64 bytes.
    */
    header_parser(0, 512);
    // ...
}

// Parser of header - limited by an offset range.
analyser header_parser (start_offset, end_offset)
{
    // ...
}

// Parser not limited by any offset range.
analyser another_parser ()
{
    // ...
}
```

Figure 6.17: Example of analysers definition in EFFDL.

```
analyser header_parser (start_offset, end_offset)
{
    uint8 'X';  // Two magic bytes - enforced values.
    uint8 'Y';
    int16;      // Don't care value (e.g. EFF version).
    // ...
}
```

Figure 6.18: Example of statement types that are usable within analysers in EFFDL.

(e.g. detecting a fault EFF structure as soon as possible).

Finally, analysers are interconnected via the `analyser_call` statement. An analyser can be invoked multiple times by using `times`, this is useful for a description of repeating parts (e.g. table items). analyser invocation can be also done within the semantic actions by a call to the function with analyser's name. Therefore, it is possible to conditionally invoke different analysers based on an actual context, see Figure 6.20.

For information about parsing of this language and implementation of this parser, please refer to [107]. The proposed parser is non-trivial because it further enhances the aforementioned (P)SCGP grammar by an addition of attributes and it also utilizes the regulated pushdown automata, see [46].

Usage Example

We can illustrate usage of the previously defined language on the 32-bit ELF format. A snippet of this description is depicted in Figure 6.21. Firstly, the header is analysed by

```
analyser header_parser (start_offset, end_offset)
{
    // ...
    int16 architecture <value>
    {
        if (architecture.value != 1)
        {   // Unsupported target architecture type.
            parse_error();
        }
        else
        {   // C++ code producing a part of EFF conversion.
            converter->setArchitectureType(architecture.value);
        }
    };
    // ...
}
```

Figure 6.19: Usage of attributes and semantic actions within analysers in EFFDL.

```
root analyser XY_EFF_parser ()
{
    // Invocation with offset range.
    header_parser(0, 512);
    // Invocation without offset range.
    another_parser();
    // Invocation 10 times.
    another_parser() [10];
    // ...
}
```

Figure 6.20: Example of different types of analyser calls in EFFDL.

invocation of the `elf_header` analyser. This analyser starts at zero offset and analyses all of its elements and makes necessary checks.

It also converts basic information (e.g. entry point, endianness) to the COFF format. The `value` attribute is used in several elements for referencing from other elements. At the end of the `elf_header` analyser, we can see a conditional invocation of section-header-table analyser. It will be executed only if the table is present. We can also see that the analyser `elf_sht` is invoked together with specification of its beginning and ending offset gathered from the previous attributes. This corresponds to the structure depicted in Figure 3.2a.

The last construction depicted in this example is a context production. It checks that executable files do not contain static relocations (e.g. static relocation `R_386_PC32`). It is marked with priority higher than the other productions; therefore, it will be checked first. Whenever the preliminary part is satisfied (e.g. executable file is not properly linked), it blocks parsing by nonterminal `error`, which leads to a parsing error.

```
root analyser ELF32 () {
    // Parse ELF header.
    elf_header(0) { check_header(); };
}

analyser elf_header (start_offset) {
    uint8 [16] e_ident { /* Check of the "ELF Identification"
                            field. */ };
    uint16 e_type <value> {
        if (e_type.value > 4)
            parse_error(); // Unsupported ELF file type.
    };
    uint16;              // e_machine - a don't care value.
    uint32 1;            // e_version has to be '1'.
    uint32 e_entry <value> { // Direct generation of COFF.
        COFF->setEntryPoint(e_entry.value);
    };
    // ...
    // Section header table's offset (SHT).
    uint32 e_shoff <value>;
    // ...
    // Size of entries in SHT and number of elements in SHT.
    uint16 e_shentsize <value>;
    uint32 e_shnum <value> {
        if (e_shoff.value != 0) // Analyse SHT.
            elf_sht(e_shoff,
                    e_shoff + e_shnum.value * e_shentsize);
    };
    // ...
}

analyser elf_sht (start_offset, end_offset) {
    // ...
    // Analysis of Section Header Table.
}

analyser elf_section (start_offset, end_offset) {
    // ...
    // Analysis of each particular section.
}

/*
    Productions describing context behavior.
    Simplified control of appearance of static relocations
    within executable files.
*/
(elf_header<is_executable>, elf_relocation<is_static>}) ->
    (error, error) [999] // High-priority production
// Other productions.
```

Figure 6.21: A code snippet of an ELF description by using EFFDL.

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 6.22.

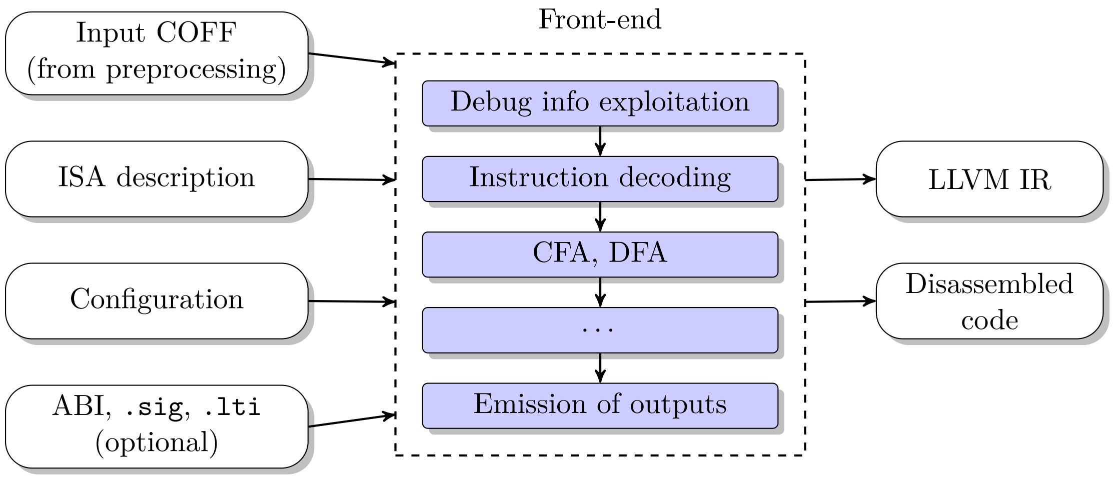


Figure 6.22: General structure of the front-end part.

The decompiled application is stored in the COFF representation produced by the `bintran` application. This COFF executable file is processed according to the configuration file generated within the preprocessing phase. By using the provided information (target architecture, used packer/compiler, etc.), the decompiler can selectively enable compiler-specific analyses (e.g. detection of entry-point function). The last mandatory input of this phase is a description of the instruction set (ISA) as described in Figure 4.5. This description is obtained from the semantic extractor described in Section 4.7 and it is used in the instruction-decoding analysis. Finally, the front-end also supports the following optional inputs:

- *ABI description* – this might be utilized within DFA;
- *Signatures of statically linked code* – these are used for elimination of such code;
- *Type information* – this optional input contains information about data types of return values and arguments of statically linked functions. It is utilized within the data-type recovery analysis.

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.

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[89] or symbolic information. A description of the former type has been given in Section 5.1 as well as a description of the libraries that we also reuse in the retargetable decompiler. Symbolic information is very similar to 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.

To be clear, presence of this information is optional (e.g. it is seldom present in malware), but we have to take advantage of it whenever it is available. Furthermore, this information is helpful during testing of the decompiler because we can test decompilation accuracy on two sets of samples – with and without debugging information. Whenever the results are the same (e.g. the same functions are detected, the same function calls are reconstructed), we can conclude that the decompilation methods are accurate for a particular test sample.

One of the first steps is to check for the debugging/symbolic information presence and load it to the internal canonical representation by using the already described libraries, see Section 5.1. After the debugging information is extracted by a parser, it is ready to be used in the decompiler's front-end, which updates its intermediate code representation based on this extracted information. In the following paragraphs, we depict the useful information and its usage within the front-end.

Module information

Based on the type of the original HLL, the decompiled binary application is created from one or more *modules* (e.g. source files). The debugging information is usually divided into smaller pieces, where each piece corresponds to one particular module. This is useful for the decompiler because it is possible to divide the resulting code into similar modules, e.g. the decompiler can divide the decompiled code into several files with the original names.

Function information

Probably the most important debugging information is related to functions. A proper function detection and recovery is a crucial task of each decompiler. Otherwise, the unstructured and hardly readable (*spaghetti*) code will be generated (e.g. code containing `goto` statements). Our decompiler utilizes its own function detection heuristics (it will be described later), but the debugging/symbolic information is used instead whenever it is available because it is essentially more precise. For example, it is possible to obtain the function location within the machine code, its name, the number of its arguments, their names and types, information about function inlining, and many others.

Variables

The decompiler is able to detect both global and local variables and guess their types. In the resulting code, it generates variable names from a fixed list of well-readable names, see [134] for details. However, the decompiler emits the original name whenever it is stored

[89]We support both PDB and DWARF formats within the decompiler, see [109].

in the debugging data. This is just a detail, but it gives the feeling of listing the original code. Furthermore, both DWARF and PDB support storage of variable types, too. This is useful because the recovery of composite data types is a non-trivial task, see [88].

Line information

As we discussed in Section 3.3, mapping of source-code locations to the machine-code is done via so-called *line information* and it is an essential part of each debugger. However, its usage within decompilation is not so important. It can be only used for re-formatting the decompiled code to better fit its original format (e.g. splitting the complex expressions into more statements, keeping the original order of function definitions).

At present, the decompiler uses all of these features. In Figure 6.23, we can see a brief comparison of three different decompilation cases on the executable file containing the factorial function `int fact(int n)`. Firstly, the exploitation of debugging information is depicted in Figure 6.23a. In this case, we were able to retrieve names and types of the function and argument, as well as details about the original module. In the second case, depicted in Figure 6.23b, only symbolic information is available, which allowed obtaining information about the `fact` function and its location, but nothing else. Therefore, our own argument and type reconstruction methods had to be used. Finally, in Figure 6.23c, the stripped version of this application is decompiled. Because there is no source of additional information (not even function/argument names are available), everything had to be reconstructed by our own methods.

```
/*
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 6.23: Decompilation of the factorial function under different circumstances.

6.4.2 Statically-Linked-Code Recognition

The decompiled application usually does not contain only the user functions, but also the code of statically-linked standard functions (e.g. `printf()`, `CreateWindow()`). However, decompilation of the second one is normally unwanted because it makes output less readable and it complicates the decompilation process too. Based on our tests [129], we can conclude that removal of this statically linked code[90] makes the decompiled code more readable. Furthermore, it also decreases the amount of code that needs to be decompiled.

A detection of such code is implemented via the so-called *signatures* (i.e. `.sig` files). By using signatures, we are able to decide, which code is statically linked and skip it during decompilation. The details about this analysis are omitted because the author made no contribution to it, see [91] instead.

6.4.3 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*, which converts machine instructions into a proper LLVM IR form.

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 the aforementioned illustration in Figure 4.5), which was originally intended only for generation of the Lissom C/C++ compiler. However, it contains all the necessary information for usage in decompilation: information about binary encoding, assembler syntax, and semantics of each particular instruction in an instruction set.

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, different instruction lengths, multiple instruction sets, etc.

We can divide the decoding process in two steps that will be described separately – creation of the instruction decoder and the main decoding process.

Creation of the Instruction Decoder

Before the decoding itself, the decoder has to be created. This is done based on the instruction encodings obtained from the extracted description. For example, the encoding of the instruction listed in Figure 4.5 is as follows:

```
; asm: ADD reg0, reg1, reg2
0b000000 gpregs1[4,0] gpregs2[4,0] gpregs0[4,0] 0b00000100000
```

Each description of such encoding may contain multiple elements that can be divided into two basic groups: fixed elements (e.g. encoding of opcode – `0b000000`) and varying elements (e.g. encoding of an immediate value or register operand – `gpregs0[4,0]`, i.e. register operand encoded on five bits). Based on these elements, we have to construct a decoder that is able to decode a sequence of input bits (machine code) and return a matching instruction. Furthermore, the decoding process has to be as fast as possible because it is one

[90]It is a code of functions from `.o` object files or `.a` libraries that is linked into application by a linker.

of the most time-consuming analyses within the decompilation process. The more complex the instruction set is (in a matter of its size and instruction length), the more lengthy the decoding is.

Furthermore, the semantics extractor (see its description in Subsection 4.7.3) usually generates more instruction descriptions than the original number of instructions (e.g. 2–10x more descriptions than instructions) because some of these instructions act differently based on the operand values. As an example we can mention MIPS architecture and its zero register `$0`. Most of the MIPS instructions expect a register operand from a register field, but when the `$0` register is used as a source operand, a zero value is obtained instead. Therefore, the semantics extractor tries to find all such differences and separate them as new instruction descriptions.

Therefore, the proposed decoder has to deal with several thousands of extracted instructions (e.g. 8,000 instructions on ARM) and ensure that the search complexity is as low as possible (i.e. $\mathcal{O}(1)$ in the best case). Furthermore, on architectures with multiple instruction sets (e.g. ARM+Thumb), it is necessary to construct one decoder for each such instruction set and switch between them during decoding. We illustrate three different approaches of how to construct the decoder and we discus their advantages and drawbacks.

(1) In a straightforward approach, we create a simple linked list of extracted instructions and match the input sequence of bits with all encodings in this list one after another (i.e. $\mathcal{O}(n)$). However, this will be terribly time-ineffective for the non-trivial applications and target architectures. For example decoding of an application with 1,000 instructions via 1,000 extracted instruction descriptions may take up to one million comparisons.

(2) In an ideal case, we could create some unordered associative container (e.g. hash table), which index represents a particular instruction encoding, such as:

```
; asm: ADD $r1, $r1, $r1
; 0b000000 0b00001 0b00001 0b00001 0b00000100000
00000000001000010000100000100000
```

This approach will satisfy that we can decode every instruction instantly (i.e. $\mathcal{O}(1)$). However, this approach would require up to $2^{instruction\_length}$ instruction-encoding permutations[91], e.g. 2^{32} for MIPS, up to 2^{120} for Intel x86. Unfortunately, this is unfeasible because of memory limitations and complexity of the initial precomputation.

(3) The last approach is the one actually used. It is similar to the previous one, i.e. it is also based on an unordered associative container because of its fast search, but we skip all the varying elements and we only compare the fixed ones on their expected bit positions within the input bit stream, e.g.:

```
; X = don't care bit
; simplified instruction encoding:
000000 XXXXX XXXXX XXXXX 00000100000
; matched input bit stream:
000000 11111 11111 11111 00000100000
```

In order to obtain the operand values, those (skipped) varying parts are evaluated once the instruction is found (i.e. once the input bit stream is matched with one of the extracted instruction descriptions). In practice (i.e. in all the supported target architectures), this modification prevents an exponential growth of all possible instruction encodings and preserves a fast search. However, it is uneasy for use the same hash table for all instructions

[91]This applies to instruction sets where all bit combinations can be decoded as a valid instruction. This mainly applies to Intel x86.

because different instructions use different bit positions of those varying parts. For example:

```
; asm: ADD reg0, reg1, reg2
0b000000 gpregs1[4,0] gpregs2[4,0] gpregs0[4,0] 0b00000100000
; asm: JAL imm0
0b000011 imm_0[25,0]
```

As we can see, there are two fixed elements in the first instruction, but only one in the second one. In a case of CISC instructions, the difference is even larger. Therefore, we have to have a different decoding hash table for each element *template* (i.e. number, size, and position of the fixed parts) and search for instruction in all of them. This is usually not important because the number of such templates (end therefore hash tables) is usually limited (e.g. several dozens) and they can be searched in parallel (e.g. in multiple threads). Furthermore, once an appropriate instruction is found, the search in the other hash-tables can be stopped because the instruction set cannot be ambiguous (i.e. two different instructions with the same encoding).

After the decoder is created, it is possible to start with decoding of machine code obtained from executable sections of the input COFF file.

Decoding Process

Instruction decoding is based on a modification of the hybrid-traversal approach, see its description in Section 4.3. The original algorithm was focused on the Intel x86 architecture. Therefore, its authors have a knowledge of the `call` and `ret` instruction encodings and they were able to more easily distinguish ranges of functions. However, this is not true in our case because each particular target architecture has a different instruction set and its encoding.

Therefore, our decoding algorithm tries to obtain information about functions either from debugging information, symbolic information, or other sources (e.g. utilization of C++ virtual method table). Such functions are decoded by using the lightweight recursive traversal approach first [80]. The *lightweight* specifier means that we are only able to inspect the targets of direct branches because evaluation of indirect branches can be done only in the later phases of decompilation.

Therefore, not all branch-target addresses are found by using this recursive traversal. Afterwards, the remaining instructions (i.e. those that do not belong to any known function or branch target) are processed by the linear sweep approach as in the original approach.

Later in this section, we discuss the other analyses over the decompiled code that can be useful if available during instruction decoding, e.g. CFA, code interpreter. However, this is not feasible because instruction decoding is a preliminary analysis for them. On the other hand, it is possible to iteratively improve the decoding results via re-invoking the decoding analysis once the results of the later analyses are available. For example, if we detect an indirect call to address, which has been improperly decoded by the linear sweep approach (e.g. decoding started at a wrong address and it decoded two invalid instructions overlapping the actual one), it is possible to call the decoder once again starting on this particular address. This also implies an update of the subsequent analyses.

The result of the application's decoding is a vector of decoded instructions, where each such instruction consists of its address, assembler syntax (for producing the disassembled code), semantics description (block of LLVM IR code), evaluation of its operands (e.g. `reg0`, `reg1`, and `reg2` in the aforementioned example), and other information.

6.4.4 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 16) and create a CFG over them (see Definition 17). 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). We distinguish the following basic branch types:

- *Unconditional branch* – This is a default type of branch that is usually used within functions, e.g. loops or `switch` statements.
- *Conditional branch* – A branch that is taken only if a particular condition is evaluated as true. This mark is applied whenever we detect such a condition associated to the branch on the LLVM IR level.
- *Function call* – Indicates that a branch is actually a function call. We cannot detect this branch type on a first sight in all cases (e.g. function may be called by using a branch instruction). In such cases, this mark is assigned dynamically either when the branch type is revealed by using a heuristic or when the target address is recognized as a function. The called function can be either internal (i.e. user defined) or external (i.e. statically-linked standard function).
- *Function return* – It is a mark for branch, which returns from a current function. The form of that branch differs a lot based on the target architecture, e.g. MIPS uses a special *link-register* for this purpose, while on ARM or x86 it is a branch to value stored on a stack.
- *Tail call* – This type is used in a combination with a function call, when we detect that the compiler uses a tail-call optimization, see [1]. In short, a function `A` calls `B`, `B` calls `C` by a tail call, and the return in `C` will return directly to `A`.
- *Pointer call* – It is a mark for a branch that calls a function indirectly by a pointer. That could be, e.g. a branch with a register operand and we are able to find out that this operand represents a function pointer.

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 described later in this section.

Another task of CFA is a calculation of instruction successors and predecessors. The instructions inside a basic block have only a single successor and a single predecessor. The first instruction of a basic block can have one or more predecessors and a single successor.

The last instruction of a basic block has a single predecessor and zero, one, or more successors (see Definition 16). Zero number of successors will be used for the last basic block of a function.

CFA also contains multiple specific analyses, such as *entry-point analysis* (i.e. analysis of the start-up code based on the detected compiler/packer and detection of the `main` function), or *loop-detection analysis* (i.e. detection of the loop patterns).

6.4.5 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. For example, an essential step of function detection is a recognition of branch targets. This step is not trivial for indirect branches, where the target of a branch is stored in a specified operand. For the sake of simplicity, in what follows, we use the term branch even for call instructions.

Branch instructions can be found in all commonly widespread architectures. The `jalr t9` instruction is used on MIPS, `mov pc, r2` on ARM, or `jmp eax` on Intel x86. If we want to resolve the target of an indirect branch, we have to resolve the value of its operand. The resolution is reached by static interpretation of the code and backtracking the value of the operand over CFG. The interpreter uses CFG for obtaining the order of instructions for processing because it contains all direct instruction predecessors. These predecessors are interpreted whenever the interpreter does not have all the necessary values. Furthermore, the processing is terminated whenever there are more predecessors and these predecessors differ in the modification of the tracked object. We explain the work-flow of the interpreter using a simple example. We take the code for the MIPS architecture listed in Figure 6.24.

```
 1  lui     gp, 0x42
 2  addiu   gp, gp, -29792
 3  sw      gp, 16(sp)
 4  sw      a0, 40(sp)
 5  ; ...
 6  lw      gp, 16(sp)
 7  nop
 8  lw      v0, -32700(gp)
 9  nop
10  move    t9, v0
11  jalr    t9
```

Figure 6.24: Assembly code with an indirect branch on the MIPS architecture.

On line 11, we find an indirect branch by the register `t9`. In this place, we call the interpreter to track the register to find its value. The previous instruction of the indirect branch is placed on line 10, there is written the value of the register `v0` into the register `t9`, from that place we start to track the register `v0`. The instruction `nop` is obviously skipped (line 9) and the check is passed to line 8. On this line, there is a load from memory into the register `v0`, but if we want a exact memory address, we need the value of the register `gp`. Therefore, we continue to track this register. On line 6, the value is loaded from a stack to this register.

From this point, the interpreter has to deal with a new situation because it must track values on the stack. It is similar to tracking a value in a register, but we have to control an exact offset on the stack. Line 5 represents more unrelated instructions similar to the instruction on line 4 (the instruction stores a value to the stack, but on the offset that is not interesting for us). On line 3, we finally find a store of a value on the tracked offset to the stack. The stored value is taken from the register `gp`, so we again start to track this register. The register is set on lines 1 and 2, so we successfully find the tracking value. The algorithm works recursively, all related instructions are resolved on a way back. Finally, we have the address that is stored in the register `t9`.

The interpreter is also capable to cache results in order to save redundant calculations, when the same task has to be interpreted again. Furthermore, we employ the use-def chains (see Definition 20) for detection of operand definition, which optimizes the backtracking process. Finally, the interpreter is also able to detect a dead-lock when control flow creates a cycle and interpretation in this case can run forever.

6.4.6 Function Detection

Within the function-detection analysis, we use a combination of two detection methods that are based on the principles described in [8, 39, 43]. 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 [129].

Top-Down Analysis

The top-down analysis takes a single block containing the whole program (i.e. all application instructions) and tries to split it into the smaller blocks (functions) until there remain only the detected functions. It works in iterations, where each iteration consists of finding targets of call-type branches (see description in CFA) and splitting of original blocks based on these detected branch targets. This approach is also used in the PROPAN system introduced in [39].

The first iteration is a prologue for the subsequent iterations. It builds two sets of instructions. (1) J_{known} consisting of all function calls with known target addresses. We initialize this set by the direct branches because their target addresses are available from the instruction encoding (e.g. `call 0x401560`). (2) Set $J_{unknown}$ containing all the other branch instructions such as branches with indirect target address (e.g. `call eax`), that may be used as function calls. We have to preserve that branches unrelated to function calls are not included in this set, e.g. branches used within loops (`while`, `for`, etc.), conditional statements (`if-then-else`), `switch` statements. Otherwise, we could break the consistency of such a construct. For this task, we use branch *marks* obtained in CFA.

In the following iterations, we try to determine targets of calls from the set $J_{unknown}$ by using code interpretation described in the previous subsection. Whenever we successfully detect a target address of a possible call from the set $J_{unknown}$, we move such branch into the J_{known} set and we check position of such target address in the existing blocks. Whenever the address is not at a beginning of any block (i.e. such address is contained *within* a block), the block split is performed at this position. This address becomes the first address of a new block and the previous address becomes the last address of another block.

After every iteration, we adjust predecessors and successors of a particular instruction to get more accurate results from the code interpreter. The top-down detection ends whenever the set $J_{unknown}$ is empty or there is no change within a particular iteration.

An example is presented in Figure 6.25. The first block `0x0368–0x0468` contains the whole program and in each iteration it is divided by addresses detected from set $J_{unknown}$. Firstly `0x039C`, `0x041C` afterwards, and finally `0x03DC`. The resulting address ranges (on right of the figure) represent detected functions.

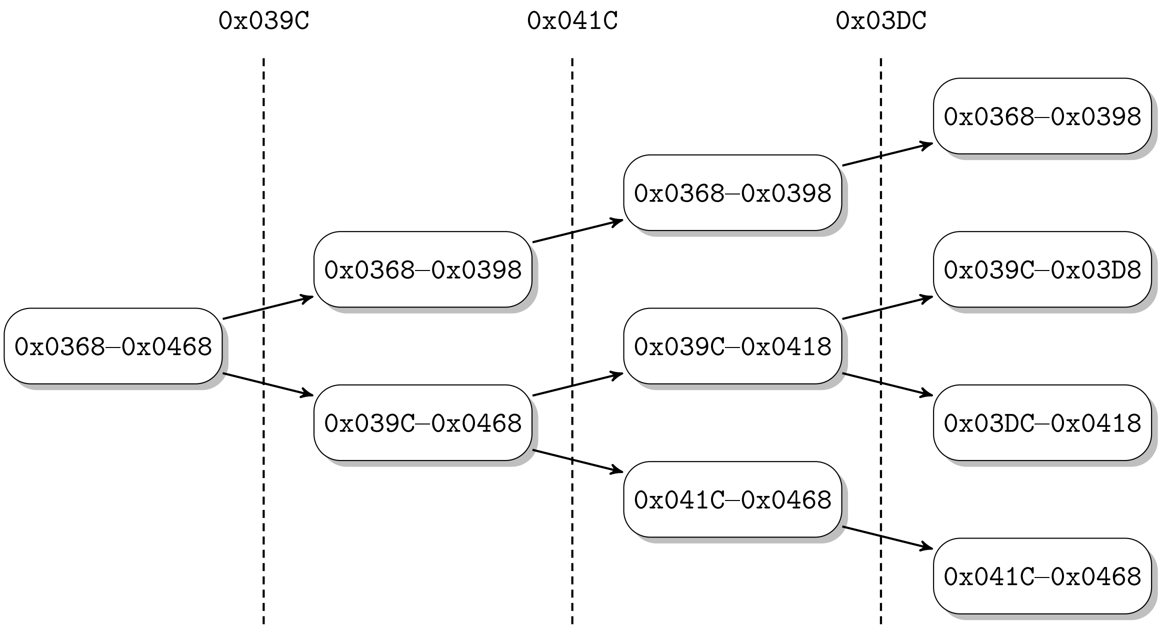


Figure 6.25: An example of the top-down function detection.

Bottom-Up Analysis

The bottom-up analysis was originally introduced in [85] and tested on TriCore and PowerPC ELF executables. Roughly speaking, it is an opposite of the top-down analysis. The original algorithm is as follows.

In the first step, every instruction is considered to be a block. Then, the blocks are iteratively joined, until basic blocks are created. Finally, basic blocks are connected to form functions.

A basic operation of this analysis is a join of two blocks into a single block. The start address of the new block is the start address of the first block and the end address is the end address of a second block. Therefore, the order of blocks is important. An example is presented in Figure 6.26, where single instructions on addresses `0x0368` – `0x0468` are iteratively joined into four resulting functions.

The most difficult phase is deciding, which blocks should be joined. Similarly to the

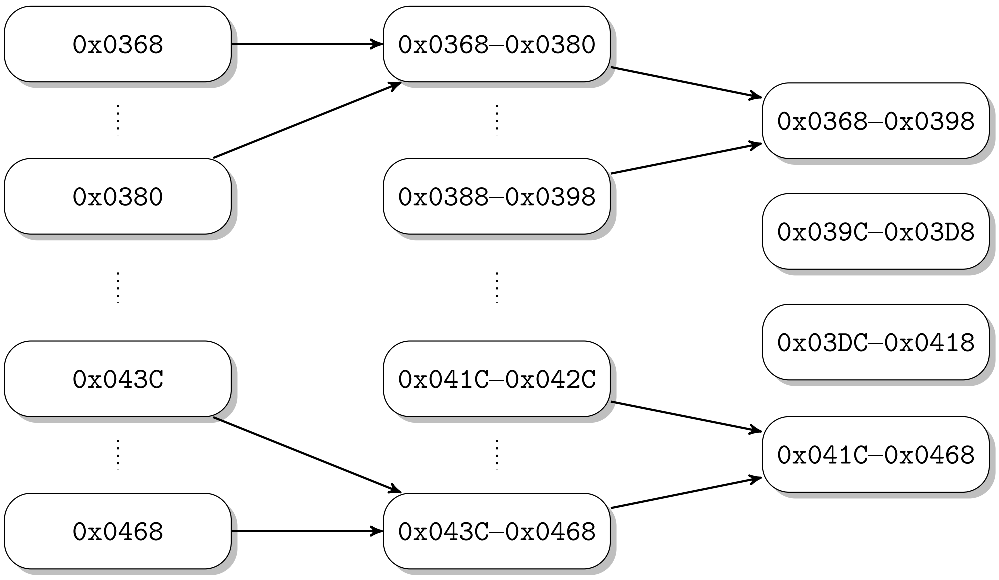


Figure 6.26: An example of the bottom-up function detection.

top-down analysis, we have to find all instructions, which call functions and create the sets J_{known} and $J_{unknown}$. The ideal state is $J_{unknown}$ being empty and the set J_{known} containing only branches that are calls of functions. However,it is not possible to always reach such a state as has been proved in [39].

In other words, the bottom-up detector joins blocks together to create functions. In our implementation, we already have basic blocks (from CFA). Therefore, the algorithm passes through basic blocks and it tries to merge them to functions. If we cannot merge a basic block to any existing function, we create a new function containing just this basic block. The merge has to be done according the correct order of block addresses to avoid their incorrect swapping. Afterwards, we determine predecessors and successors of these instructions.

This detector is used to improve the top-down approach; after every iteration of the top-down detector, the bottom-up detector is called to make its analysis on every detected function. At this point, it is also possible to construct an application's call-graph (see Definition 18).

6.4.7 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.

Function-Arguments Recognition

A recognition of arguments and return values is made by analysing function bodies and function calls. If we know the used ABI, this task is much easier. Indeed, it is enough to check instructions that work with registers or a stack according to the ABI description. However, attention has to be paid to instructions that access registers only locally. This means that inside of a function, the registers that are defined for argument passing are used, but the arguments were not passed by them. This is the main weakness of ABI use and, therefore, it is useful mainly for speeding up the recognition.

The data-flow analysis is based on a *memory-places* analysis originally presented in [101], 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 [129] 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. The data-flow analysis computes these arguments from instructions that access the stack or registers.

The input arguments are stored in the analysed function and output arguments are passed to the called function at a point of a call so not all of them are considered. After the computation, real arguments are recognized as the intersection of input arguments and arguments received from the caller. An important role is played by the order of the write and read instructions into some memory place. If there is a write before a read in the callee, it signals an occurrence of a local variable. It is important to detect such situations; otherwise, it may lead to the incorrect number of detected arguments.

The same principle as for function arguments is also applied to recognition of return values. The only difference is a reversed order of read and write instruction (i.e. write in callee and read in caller).

Furthermore, memory-places are also used for storage of a function return address. However, this value has to be skipped by the algorithm because it is not an argument. The return address is a memory place containing a value of the program counter (PC). The analysis is looking for storing the PC to a memory place and handles it correspondingly.

This algorithm is advantageous due to no dependency on call conventions. Indeed, it can handle custom call conventions. An exception is a statically or dynamically linked code of standard libraries, which follows call conventions of the used compiler. Therefore, the data-flow analysis utilizes this fact for functions from such libraries.

For this reason, we support description of a particular ABI in a simple description language. Such descriptions are passed to front-end as `.abi` files that support the following constructs:

- `data` – contains mapping of generic data types to processor-specific data types, such as alignment for storing smaller types to larger ones, description of register couples);
- `stack-direction` – the first function argument (the leftmost) is stored first (i.e. left-to-right, LTR) or vice versa (i.e. right-to-left, RTL);
- `stack` – description of a stack (e.g. specification of a stack pointer, its initial value, alignment);
- `jump` – a memory place (register or stack), where the function return address is stored;
- `return` – how to return a return-value from a function;
- `passing` – description of argument passing to function.

An example of MIPS ABI, which is based on [66], is depicted in Figure 6.27.

```
section data
    i1   i32
    i8   i32
    i16  i32
    i64  i32:i32
    *    i32

section stack-direction
    RTL

section stack
    Reg gpregs 29
    start 0
    align 4

section jump
    i32     Reg    gpregs    31

section return
    i32     Reg    gpregs    2
    float   fReg   fpuregs_s 0
    double  fReg   fpuregs_d 0
    i32:i32 Reg    gpregs    2:3

section passing
    i32:i64 from gpregs 0 to 4 Reg start 4
    i32:i64 Stack
    float from fpuregs_s 0 to 2 fReg start 12 step 2
    float Stack
    double from fpuregs_d 0 to 0 fReg start 0 step 2
    double Stack
```

Figure 6.27: Example of a MIPS ABI description.

6.4.8 Other Analyses

There are many more analyses used within the front-end part. However, we will omit their description because they are either not so important as the aforementioned ones or the author's contribution within these is none or very limited. However, we will briefly outline several of them:

- *Data-section analysis* – detection of objects stored within data sections, such as strings, (floating-point) numbers, etc. This is also used for detection of object's data type.

- *Stack analysis* and *local-variables detection* – it investigates the creation and usage of a stack in every function in order to detect local variables.

- *Type analysis* – type analysis reconstructs data types of arguments, return values, and variables. This analysis is quite complex and it is able to reconstruct the basic data types (integer, floating point, pointers, character strings, etc.) as well as composite data types (e.g. arrays, structures, recursive types). It is based on [53]. This analysis can be also enhanced by providing the optional `.lti` files, which describe data types of particular functions.

6.4.9 Outputs Generation

The front-end produces two basic outputs – LLVM IR representation of the input application and the disassembled code. The former one is further processed by the subsequent parts, while the second is not (there is no need to further optimize this code).

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). Firstly, it has to generate the declarations of all linked functions and global variables. The next step is producing the IR code, which is divided into functions that were recognized by the function-detection analysis. Finally, the generator emits metadata that contains additional information for the subsequent parts of the decompiler—middle-end and back-end. This includes the number of decompiled functions, real names of variables and arguments (this is acquired from debugging information), etc. An example of LLVM IR code snippet produced by the front-end is presented in Figure 6.28.

```
;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 6.28: Example of a generated LLVM IR code for one machine-code instruction.

Generation of Disassembled Code

The front-end part is also able to emit the disassembled code[92]. In Figure 6.29, we can see a shortened example of a disassembled factorial application. In comparison with other disassemblers (e.g. `objdump` depicted in Figure 6.10), 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 6.30.

[92]The decompiler emits the disassembled code in the Intel syntax, see https://en.wikipedia.org/wiki/X86_assembly_language#Syntax for differences between AT&T syntax and Intel syntax.

```
;;
;; This file was generated by the Retargetable Decompiler
;; Website: http://decompiler.fit.vutbr.cz
;; Copyright (c) 2011-2014 Lissom <decompiler@fit.vutbr.cz>
;;
;; Detected functions in front-end: 2
;; Decompiler release: dev (Sep  5 2014)
;; Decompilation date: Sep 06 2014 16:36:35
;; Architecture: Intel x86
;;

;;
;; Code Segment
;;
;  section: .text
; ...
;  function: factorial at 0x401560 -- 0x401587
0x401560:  55                         push ebp
; ...
0x401586:  c3                         ret
;  function: main at 0x401587 -- 0x4015d0
0x401587:  55                         push ebp
0x401588:  89 e5                      mov ebp, esp
0x40158a:  83 e4 f0                   and esp, 0xfffffff0
0x40158d:  83 ec 20                   sub esp, 0x20
0x401590:  e8 d9 0d 00 00             call 0x402370 <___main>
0x401595:  e8 2e 5f 00 00             call 0x4074c8 <rand>
0x40159a:  89 44 24 1c                mov dword [ esp + 0x1c ], eax
0x40159e:  8b 44 24 1c                mov eax, dword [ esp  + 0x1c ]
0x4015a2:  89 04 24                   mov dword [ esp ], eax
0x4015a5:  e8 b6 ff ff ff             call 0x401560 <factorial>
0x4015aa:  89 44 24 18                mov dword [ esp + 0x18 ], eax
0x4015ae:  8b 44 24 18                mov eax, dword [ esp + 0x18 ]
0x4015b2:  89 44 24 08                mov dword [ esp + 0x8 ], eax
0x4015b6:  8b 44 24 1c                mov eax, dword [ esp + 0x1c ]
0x4015ba:  89 44 24 04                mov dword [ esp + 0x4 ], eax
0x4015be:  c7 04 24 44 90 40 00       mov dword [ esp ], 0x409044
                                      ; "factorial(%d) = %d\n\x00"
0x4015c5:  e8 06 5f 00 00             call 0x4074d0 <printf>
0x4015ca:  8b 44 24 18                mov eax, dword [ esp + 0x18 ]
0x4015ce:  c9                         leave
0x4015cf:  c3                         ret
;...

;;
;; Data Segment
;;
;  section: .rdata
; ...
0x409044:  66 61 63 74 6f 72 69 61 ...   |factorial(%d) = |
0x409054:  25 64 0a 00                   |%d..            |
```

Figure 6.29: Example of a disassembled code (Intel x86).

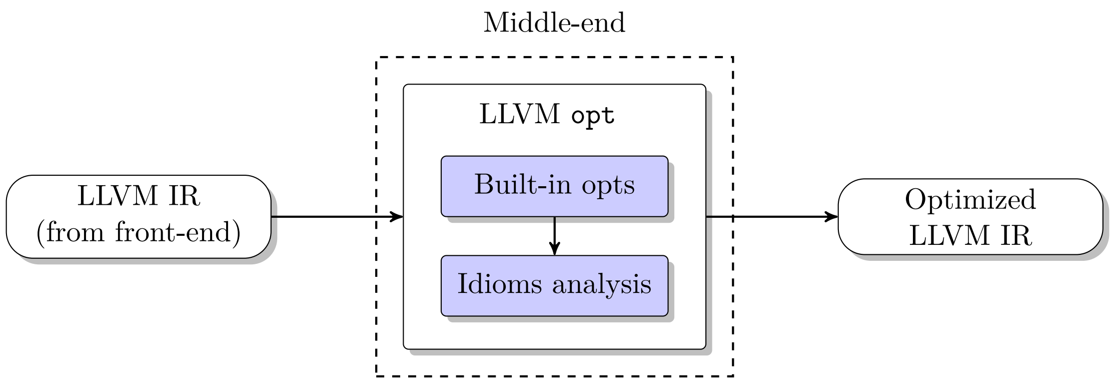


Figure 6.30: A general view on the middle-end part.

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.

As we mentioned in Section 6.2, the middle-end is based on the `opt` tool[93] from the LLVM platform. This tool provides many built-in optimizations and analyses[94]. 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. In Subsection 6.5.1, we briefly described the built-in optimizations that we re-use during decompilation. Furthermore, we describe our own code-transformation pass implemented within the `opt` tool in Subsection 6.5.2. 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 Built-in Optimization

We utilize the `opt` optimization passes that are normally used during compilation with optimization level `-O3` (i.e. even the aggressive optimizations are used). However, before we do this, it should be noted that the optimizations cannot be adopted as they are. The reason is that in the decompiler, we will convert low-level representations (binary applications) into high-level representations with focus on readability and analysability of the resulting code.

The goal of a compiler is quite different. Indeed, a compiler goes in a converse way, starting from source code written in some high-level language and ending with either assembly code or a binary file. To this end, a compiler utilizes transformations to ease this process, like lowering high-level constructs or transforming statements into several instructions.

We do not want to include such optimizations because they make the code less readable. However, many times, such optimizations are part of other optimizations that are useful for us. Therefore, it is necessary to be careful which parts of optimizations to enable. To conclude this section, we use the following built-in `-O3` passes for the code optimization:

```
-adce -basicaa -basiccg -constmerge -correlated-propagation
-domtree -dse -early-cse -globaldce -globalopt -gvn -indvars
```

[93] http://llvm.org/docs/CommandGuide/opt.html
[94] Complete list: http://llvm.org/docs/Passes.html

```
-instcombine -instcombine -ipsccp -jump-threading
-lazy-value-info -lcssa -licm -loop-deletion -loop-idiom
-loop-rotate -loops -loop-simplify -lower-expect -memdep -no-aa
-preverify -reassociate -scalar-evolution -sccp -simplifycfg
-strip-dead-prototypes -targetlibinfo -tbaa -verify
```

Since in the decompiler we go from a low-level representation into a high-level representation, some optimizations that compilers usually use should not be utilized. For example, consider the optimization called scalar replacement of aggregates (see [60]), available in `opt` by using the `-scalarrepl` option. This optimization breaks aggregated types, like structures or arrays, into individual variables. For example, this optimization turns the following C code

```
struct ComplexNum {
    double r;
    double i;
};

ComplexNum a = {1.5, 2.3};
```

into

```
double var1 = 1.5;
double var2 = 2.3;
```

Although such an optimization is perfectly reasonable to be used by a compiler, in the decompiler, it does more harm than good. Indeed, we would like to keep aggregated types untouched because such high-level types are used by programmers when they write their code.

Therefore, we removed the following passes from the `-O3` list (e.g. function inlining, loop unrolling):

```
-argpromotion -deadargelim -functionattrs -inline -inline-cost
-internalize -loop-unroll -loop-unswitch -memcpyopt -notti
-prune-eh -scalarrepl -simplify-libcalls -sroa -tailcallelim
```

Finally, we use other passes supported by LLVM that are not included in the `-O3` optimization level:

```
-constprop -dce -die -disable-inlining -disable-internalize
-disable-simplify-libcalls -instnamer -ipconstprop
-scalarrepl-ssa
```

Next, we give a brief overview of the most important optimizations used within this phase. For a detailed description, see [1, 60].

- *Alias analysis* – Alias analysis, also known as pointer analysis, is used to determine, which memory locations may be accessed indirectly by using pointers. For example, there may be a pointer to a local variable, and its value may be changed by modifying the variable itself or by using the pointer. The `opt` tool supports several alias analyses, which may be used to improve the results of many optimizations.

- *Dead-code elimination* – By using the `-adce` and `-dce` options, we may eliminate the so-called *dead code*. This is code that is either not reachable or that does not perform any meaningful computation. Furthermore, we may remove other dead constructs, like types, global variables, loops, function prototypes, or `store` instructions.

This removal is enabled by using the `-deadtypeelim`, `-globaldce`, `-loop-deletion`, `-strip-dead-prototypes`, and `-dse` switches of `opt`, respectively.

- *Constant propagation* – The `-constprop` switch provides intraprocedural constant propagation and merging. For example,

  ```
  add i32 1, 6
  ```

  will be simplified to `i32 7`. Furthermore, `-ipconstprop` implements a simple interprocedural constant propagation. The difference between an intraprocedural and interprocedural optimization is that the former optimizes each function in separation without considering function calls while the latter optimizes all functions as a whole and takes function calls into account.

- *Combination of redundant instructions* – By using the `-instcombine` option, we may combine several instructions to form fewer, simple instructions. For example,

  ```
  %Y = add i32 %X, 1
  %Z = add i32 %Y, 1
  ```

  will be simplified to

  ```
  %Z = add i32 %X, 2
  ```

  Many sub-optimizations of this optimization require a special treatment. For example, we do not want it to generate other instruction idioms that have to be eliminated in the idiom analysis (see Subsection 6.5.2). As this optimization, like others mentioned in this list, are primarily utilized during compilation, high-level constructs may be lowered by using various idioms into a less-readable form, which is what we do not want.

- *Conversion of `switch` instructions into branches* – If the target high-level language does not support the `switch` statement, we may use this option to convert all occurrences of the `switch` instruction into a series of branches.

- *Reassociation of expressions* – The `-reassociate` option reassociates commutative expressions in an order that is designed to promote better results of other optimizations. For example, `4 + (x + 5)` is converted into `x + (4 + 5)`, which can be further simplified into `x + 9`.

6.5.2 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* that was briefly mentioned in Section 6.3. 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 [96].

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`).

In one of our papers [111], we presented an approach of dealing with instruction idioms within the front-end part. The implementation was tested on several modern compilers and target architectures. According to the experimental results, the proposed solution was highly accurate on the RISC (Reduced Instruction Set Computer) processor families—up to 98%; however, this approach was inaccurate (only 21%) for more complex architectures, such as CISC (Complex Instruction Set Computer) [111].

Therefore, in [112], we presented an enhanced approach of instruction-idioms transformation within the middle-end part, which can be effectively used even on CISC architectures. The novel approach is described in the rest of this section.

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. For example, the GCC compiler supports these optimization levels[95]:

- `O0` – without optimizations;
- `O1` – basic level of speed optimizations;
- `O2` – the common level of optimizations (the ones contained in `O1` together with basic function inlining, peephole optimizations, etc.);
- `O3` – the most aggressive level of optimizations;
- `Os` – optimize for size rather than speed.

In modern compilers, the emission of instruction idioms cannot be explicitly turned on by some switch or command line option. Instead, these compilers use selected sets of idioms within different optimization levels. Each set may have different purpose, but multiple sets may share the same (universal) idioms.

The main reasons of why to use instruction idioms are:

- The most straightforward reason is to exchange slower instructions with faster ones. These optimizations are commonly used even in the lower optimization levels.
- The floating-point unit (FPU) might be missing, but a programmer still wants to use floating-point numbers and arithmetic. Compilers solve this task via floating-point emulation routines (also known as *software floating point* or *soft-float* in short). Such routines are generated instead of hardware floating-point instructions and they perform the same operation by using available (integer) instructions.
- Compilers often support an optimization-for-size option. This optimization is useful when the target machine is an embedded system with a limited memory size. An executable produced by a compiler should be as small as possible. In this case, the compiler substitutes a sequence of instructions encoded in more bits with a sequence of instructions encoded in less bits in general. This can save some space in instruction cache too.

[95]See `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html` for details.

Another type of optimization classification is to distinguish them based on the target architecture. Some of them depend on a particular target architecture. If a compiler uses platform-specific information about the generated instructions, these instructions can be classified as platform-specific. Otherwise, they are classified as platform-independent.

As an example of a platform-independent idiom, we can mention the `div` instruction representing a fixed-point division. Fixed-point division (signed or unsigned) is one of the most expensive instructions in general. Optimizing division leads to a platform-independent optimization.

On the other hand, clearing the content of a register by using the aforementioned `xor` instruction is a highly platform-specific optimization. Different platforms can use different approaches to clear the register content. As an example, consider the aforementioned `$zero` register on MIPS, which always contains the value of 0. By using this register as a source of zero bits may be a faster solution than using the `xor` instruction.

Furthermore, different compilers use different instruction idioms to fit their optimization strategies. For example, GCC uses an interesting optimization when making a signed comparison of a variable. When a number is represented on 32-bits and bit number 31 is representing the sign, logically shifting the variable right by 31 bits causes to set the zeroth bit equal to the original sign bit. The C programming language classifies 1 as *true* and 0 as a *false*, which is the expected result of the given less-than-zero comparison. This idiom is shown in Figure 6.31. Figure 6.31a represents a part of a source code with this construction. The result of its compilation with optimizations enabled is depicted in Figure 6.31b. We illustrate the generated code on the C level rather than machine-code level for better readability.

```
int main(void)
{
    int a, b;

    /* ... */

    b = a < 0;

    /* ... */
}
```

(a) Original C code.

```
int main(void)
{
    int a, b;

    /* ... */

    b = lshr(a, 31);

    /*... */
}
```

(b) Code generated by the GCC compiler (for better readability in C).

Figure 6.31: Example of an instruction idiom – usage of the `lshr()` function.

The compiler used the aforementioned instructions idiom—replacing the comparison by the shift operation. The non-standardized `lshr()` function is used in the output listed in Figure 6.31b. The C standard does not specify whether operator „>>“ means logical or arithmetical right shift. Compilers deal with it in an implementation-defined manner. Usually, if the left-hand-side number used in the shift operation is signed, arithmetical right shift is used. Analogically, logical right shift is used for unsigned numbers.

In Table 6.4, we can see a shortened list of instruction idioms used in common compilers. This list was retrieved by studying the source codes responsible for code generation (this applies to open-source compilers—GCC 4.7.1, Open Watcom 1.9, and LLVM/`clang` 3.3) and via reverse engineering of executable files generated by these compilers (this method was used for other compilers—Microsoft Visual Studio C++ Compiler 16 and 17, Borland 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 6.4: Shortened list of instruction idioms found in compilers.

Instruction idiom	**GCC**	**Visual Studio C++**	**Intel C/C++ Compiler**	**Open Watcom**	**Borland C Compiler**	**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	✓	×	×	×	×	×
Substitution by `copysignf()`	✓	×	×	×	×	×
Substitution by `fabsf()`	✓	×	×	×	×	×

Idiom Analysis in the Retargetable Decompiler

The aim of the decompiler presented in this thesis is to allow retargetable decompilation independently on the particular target platform or the used compiler. Therefore, the methods of instruction-idioms detection and code reconstruction have to be retargetable too. For this purpose, we present an approach that uses the unified code representation in the LLVM IR format.

As we described in this chapter, LLVM IR is a set of low-level instructions similar to assembly instructions. Moreover, LLVM IR is platform-independent and strongly typed, which meets our requirements. Therefore, machine instructions from different architectures can be easily mapped to sequences of LLVM IR instructions. This brings an ability to implement a platform-independent instruction-idioms analysis.

Instruction-Idiom Analysis within the Middle-end

As has been noted, the original approach was implemented in the front-end phase. However, this analysis does not fit in this phase. The LLVM IR representation in the front-end is on a very low level. With only a few exceptions, one machine code instruction is usually translated to multiple LLVM IR instructions. If we realise that a program usually contains thousands of instructions, we have a very large set of LLVM IR instructions to be inspected. This causes a negative impact on decompilation time and accuracy. However, both of these metrics can be enhanced if we inspect instruction idioms in a more optimized form.

Moreover, the front-end represents instructions in a native way—as a list of LLVM IR instructions. Implementing instruction-idioms analysis in this way is not easy due to the position of instructions, especially on CISC architectures (e.g. Intel x86). For example, CISC superscalar processors have instructions with varying execution time that can occupy different CPU units (e.g. adder, multiplier, branch unit) at a different time. Furthermore, different techniques are used to reduce the run-time, such as maximal utilization of CPU units. By using these techniques, superscalar processors can fetch and decode instructions more effectively. Therefore, modern compilers try to optimize instruction positions to improve performance and they also try to avoid instruction hazards via spreading of instructions to different places.

As well as any other instruction, instruction idioms can be also spread across basic blocks. Looking for such a shuffled instruction idiom in a linear search, used in the original approach, can lead to a failure if advanced optimizations were turned on at the compile time. Therefore, a more sophisticated algorithm has to be used.

On the other hand, the middle-end phase represents instructions also as a sequence. However, these instructions can be easily inspected in a tree way—by using an abstract syntax tree (AST, see LLVM pattern matching [84]). Inspecting AST can remove problems with the positioning of instructions. Furthermore, the main goal of the middle-end part is to optimize instructions and remove duplicate or redundant instructions, which results in a low-level transformation from machine code into LLVM IR. Therefore, we decided to move the front-end implementation of instruction-idioms analysis to the middle-end part.

The difference of the code complexity between the front-end and middle-end is significant in favor of the middle-end. As an example, see Figure 6.32. This figure contains a simple machine-code sequence calculating multiplication, which represents a part of a program that is being decompiled.

```
; ...
movl address, %eax   ; eax = address
mull 21, %eax        ; eax = address * 21
; ...
```

Figure 6.32: Example of a program for decompilation (assembly code).

Instructions in this example have to be decoded and stored as LLVM IR in the front-end part, see Figure 6.33a. Translation from machine instructions into LLVM IR is done for every single instruction and every dependence is omitted because there is no context information yet. This approach causes generation of a very large number of LLVM IR instructions. Such a representation is used in the front-end phase because it is a very early phase of the decompilation process.

```
; ...
%0 = load i32* %address
store i32 %0, i32* @eax
%1 = load i32* @eax
%1_64 = sext i32 %1 to i64
%tmp1 = add i32 0, 0
%2 = add i32 21, 0
%2_64 = sext i32 %2 to i64
%3 = mul i64 %1_64, %2_64
%imm_32 = add i64 32, 0
%4 = lshr i64 %3, %imm_32
%5 = trunc i64 %4 to i32
%6 = trunc i64 %3 to i32
%tmp2 = add i32 0, 0
store i32 %6, i32* @reg0
%tmp3 = add i32 2, 0
store i32 %5, i32* @eax
; ...
```

(a) Front-end part.

```
; ...
%1 = load i32* @eax
%2 = mul i32 %1, 21
store i32 %2, i32* @eax
; ...
```

(b) Middle-end part.

Figure 6.33: Comparison of a front-end and middle-end LLVM IR code complexity for code from Figure 6.32.

However, the LLVM IR representation in the front-end phase is not suitable for a high-level analysis, such as instruction-idioms analysis because of its complexity. Moreover, the representation of instructions in LLVM IR is highly dependent on the target architecture. As can be seen in Figure 6.33a, the result of the multiplication instruction (`mul`) on the used architecture is a 64-bit number stored in two registers. On some architectures the result of such a multiplication instruction is only a 32-bit number. As has been stated in the introduction of this subsection, instruction-idioms analysis should be architecture independent. This is only one of the problems related to inspecting instruction idioms in the front-end.

Contrariwise, the code depicted in Figure 6.33a is heavily optimized during the middle-end phase and the result is shown in Figure 6.33b. As we can see, all the architecture-dependent computations are removed; moreover, if the higher 32-bits of a 64-bit result are not used, they are removed in the dead-code-elimination optimization too. Looking for an instruction idiom in such a straightforward representation is much easier, platform independent, and it leads to better instruction-detection results.

A simplified algorithm used in the middle-end is specified in 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 AST containing one particular instruction idiom. If so, the AST is inspected and whenever an instruction idiom is found (via a function `FindIdiom()`), 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.

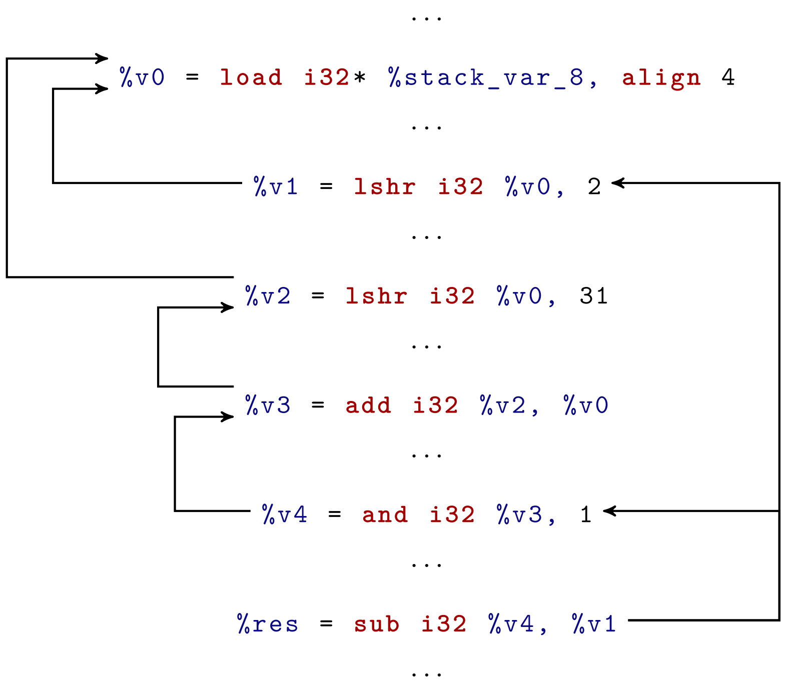
An example of idiom detection based on this algorithm is depicted in Figure 6.34. In this example, a root of a particular idiom is detected at the `sub` instruction. The arrows denote an AST inspection. For clarity, the AST itself is illustrated in Figure 6.35.

Once a particular idiom is detected, it is substituted into a human-readable form on

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
```



```
                    ...
%v0 = load i32* %stack_var_8, align 4
                    ...
        %v1 = lshr i32 %v0, 2
                    ...
        %v2 = lshr i32 %v0, 31
                    ...
        %v3 = add i32 %v2, %v0
                    ...
        %v4 = and i32 %v3, 1
                    ...
      %res = sub i32 %v4, %v1
                    ...
```

Figure 6.34: Example of instruction-idioms inspection in the middle-end part. Instructions are inspected in a tree-way.

the level of LLVM IR instructions. An example demonstrating this substitution on the LLVM IR level is shown in Figure 6.36. Figure 6.36a 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 6.36b. In this case, the typed variable `%2` holds zero, which can be directly stored in the register.

As stated above, the main goal of the middle-end part is to optimize LLVM IR that was analysed in the front-end. The middle-end uses different `opt` passes to get optimal code for the back-end part. Instruction-idioms analysis has been developed as one of the function passes of `opt`. Some transformations are useful for instruction-idioms pass, thus

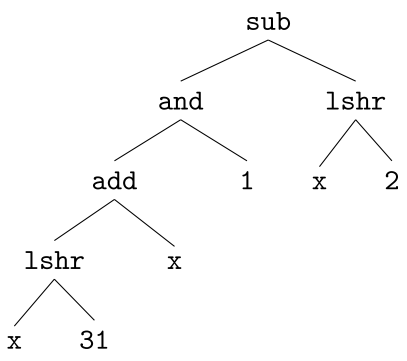


Figure 6.35: AST created based on Figure 6.34.

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

it is important to fit the instruction-idioms pass into a proper position within other passes.

It is also important to mention that decompilation of an executable file is a time consuming process. The decompilation time highly depends on the size of the input executable file. A good approach of how to optimize instruction-idioms analysis is to use any available information to save decompilation time. This is especially important when we support many instruction idioms. Some of them are specific for a particular compiler and therefore, they can be omitted from the detection phase whenever another compiler is detected. On the other hand, detection of the used compiler (as described in Subsection 6.3.1) may be inaccurate in some cases and the algorithm will not detect any used compiler. In that case, the idiom analysis tries to detect all the supported idioms. Another optimization approach is to detect only the platform-specific idioms based on the target architecture and omit idioms for other architectures.

However, the information about the architecture and compiler has to be propagated into the middle-end because it is usually not available in this phase. The only input in `opt` is a file with LLVM IR so this file has to carry this information by using the LLVM metadata mechanism. This gives us an ability to inspect only instruction idioms that are used by compilers on a given architecture.

At present, we support detection of all instruction-idioms specified in Table 6.4, among others. In Figure 6.37, we demonstrate a code reconstruction for one of these idioms. In this figure, we can compare decompilation results with and without the instruction-idioms analysis. Figure 6.37a illustrates a simple C program containing the division idiom. The decompilation result obtained without instruction-idioms analysis is depicted in Figure 6.37c. It contains three shift operations and one multiplication by a magic value. Without the knowledge of the fundamentals behind this idiom, it is almost impossible to understand the resulting code. On the other hand, the decompilation result with instruction-idioms analysis enabled is well readable and a user can focus on the meaning of the program, not on deciphering optimizations done by a compiler, see Figure 6.37b.

```
int main(void)
{
    int a;

    /* ... */

    a = a / 10;

    /* ... */
}
```

(a) Input.

```
int main(void)
{
    int a;

    /* ... */

    a = a / 10;

    /* ... */
}
```

(b) Output with idiom analysis enabled.

```
int main(void)
{
    int a;

    /* ... */

    a = (lshr(a * 1717986919, 32) >> 2) - (a >> 31)
        ;

    /* ... */
}
```

(c) Output with idiom analysis disabled.

Figure 6.37: C code example of decompilation with and without the idiom analysis.

In the conclusion of this section, we can state that the novel approach is more powerful and robust than the original one.

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 purpose of this section is to give a full description of the proposed decompiler.

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. The latter is very similar to Python, except for a few differences—whenever there is no support in Python for a specific construction, we use C-like constructs. Furthermore, instead of arrays, we use lists, and instead of structures, we utilize dictionaries. We also use the address and dereference operators from C. As there are cases when the code cannot be structured by high-level constructs only (e.g. an irreducible sub-graph of the CFG is detected, see [13], which means that `goto` is needed), an explicit `goto` represents a necessary addition to our language.

Like the middle-end, the back-end part is also based on the LLVM toolchain. This time, we re-use the `llc` tool[96], 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

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

for LLVM IR handling and the remaining methods are created by us. The general structure of this part is displayed in Figure 6.38.

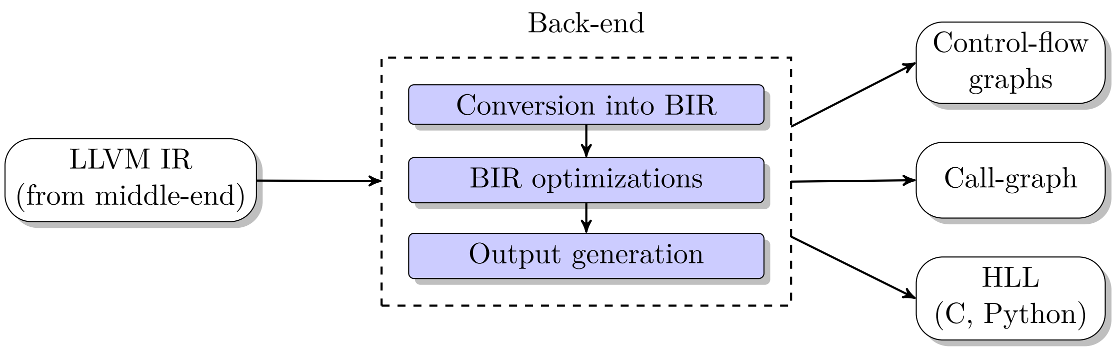


Figure 6.38: General structure of the back-end part.

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.

These steps are described in the following subsections, but first, we will briefly describe the BIR format.

6.6.1 Back-End Intermediate Representation

BIR is an internal representation of LLVM IR used within the back-end part [99]. Since it is only internal, everything can be kept in memory so we do not need any textual representation. This representation is able to model all constructs from LLVM IR and it is also able to group several statements and expressions into a single expression so high-level constructs may be presented. This is not possible in LLVM IR since it is fairly low-level. For example, consider the following piece of code: `return a + 1`. An LLVM IR representation of this piece of code is given in Figure 6.39.

```
%1 = load i32* %a, align 4
%2 = add i32 %a, 1
ret i32 %2
```

Figure 6.39: An LLVM IR representation of `return a + 1`.

In BIR, we are able to represent a single return statement returning the expression `a + 1`. The BIR representation is designed to effectively represent all the fundamental HLL constructs such as functions, statements, or expressions. At present, the BIR supports the essential C and Python language constructs, but it is modular enough to be extended by the other constructs from these or other languages. Figure 6.40 presents an example of how the expression `a * (b + 1)` is represented in BIR. Notice that in this form of representation, priority-enforcing brackets are not needed.

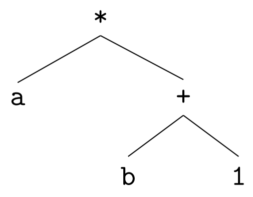


Figure 6.40: Representation of the expression `a * (b + 1)` in BIR.

6.6.2 Conversion into BIR

The LLVM IR is a fairly low-level representation, in the sense that instead of using conditional statements (i.e. `if-then-else`), there are only conditional jumps (i.e. `br`). However, conditional statements and loops are easier to read than a spaghetti code full of `goto` statements. To this end, we have to reconstruct high-level constructs from these low-level constructs.

Figure 6.41 shows several types of regions in a CFG that define high-level constructs. Region (a), composed of blocks `B1` through `B3`, shows a sequence of statements, where all the blocks are executed right after each other. Region (b) displays the structure of an `if-then-else` statement. In there, block `B1` is executed, and then, depending on a condition, either `B2` (`then`) or `B3` (`else`) are taken. After that, no matter which of the two blocks was executed, `B4` is performed.

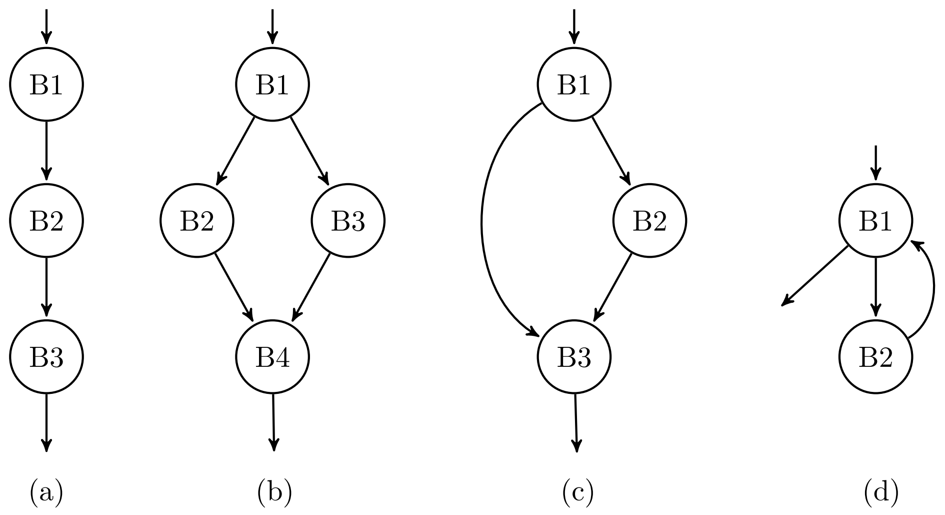


Figure 6.41: Several types of regions defining high-level constructs.

Region (c) shows a simplified version of an `if-then-else` statement—an `if-then` statement. The difference is that in (c), if the condition evaluates to false, then `B3` is directly executed, without going over `B2`. Otherwise, it works in the same way. The last region, (d), displays a `while` loop, where `B1` contains a condition that is evaluated before every iteration and `B2` represents the loop's body. Once the condition is not satisfied, the loop is exited.

To structure the input LLVM IR, we use the following method. For every function in the current module, we start by an unstructured CFG that is obtained by a direct conversion of LLVM IR into BIR. Then, we keep iterating over the CFG and during every iteration, we try to identify some of the high-level constructs from Figure 6.41. If there is a group of blocks that matches a high-level construct, we convert it into a representation of such a construct. For example, if we identify a `while` loop, we create an instance of `WhileLoopStmt`, which is the representation of a `while` loop in BIR. The resulting loop will then behave as a single block, so the number of nodes in the CFG is decreased. We keep performing iterations until there are no changes in the CFG, which means that all high-level constructs that could be identified have been identified.

Furthermore, within the conversion process, there are many other analyses over the LLVM IR code. For example analysis of instruction-operand signess (in LLVM IR, there is no distinction between signed and unsigned integers in terms of types) or processing debugging information produced by the front-end. See [94] for details.

6.6.3 BIR Optimizations

Even though the main place for optimizing the decompiled code is the middle-end (see Section 6.5), there is still a need for doing optimizations also in the back-end. As we will shortly describe, the reason is that after converting LLVM IR to BIR and reconstructing high-level constructs, additional passes should be performed to improve code readability[97]. It is also important not to alter the functionality of the decompiled application during these optimizations (i.e. to preserve the functional equivalence of code). At present, there are more than thirty implemented optimizations, but we will discuss only a fraction of them:

- *Simplification of Arithmetical Expressions* – for example, the following statement

  ```
  a = -1 * (rand() + -1) * 2
  ```

  may be simplified into

  ```
  a = -2 * (rand() - 1)
  ```

  Our method for simplification of arithmetical expressions iteratively traverses the BIR representation of the decompiled program and it tries to identify constructions that can be simplified.

- *Copy Propagation* – to make the code more readable, we have to eliminate all auxiliary assignments that just copy expressions by propagating the expression through the code. For example, the following code

  ```
  a = func()
  var5 = a
  return var5;
  ```

[97]Even the same types of optimizations as in middle-end (e.g. copy propagation) are performed because they further improve the HLL code.

may be simplified into

```
return func()
```

Firstly, we compute so-called *def-use* and *use-def chains* (see Definitions 19 and 20) by using the standard data-flow algorithms (see [1]). We then use the information from both of these chains during the actual optimization.

- Removal of self assignment statements (e.g. `a = a`).
- Removal of dereference of address operator (e.g. transform expression `*&a` into `a`).
- Conversion of `while` loops to `for` loops if they suit better.
- Conversion of `if/else-if/else` sequences to the `switch` statement.
- Conversion of bitwise operations used in conditions into logical operations.
- Removal of dead-code and unnecessary type casts.

Furthermore, the back-end contains several optimizations focused on generation of a more readable code:

- *Renaming of variables* – The decompiled application in most cases does not contain the originally used variable name (those are wiped by the compiler). Therefore, we try to make the resulting source code as readable as possible by renaming these variables. It is always better to give variable a meaningful name than just some address of its definition (e.g. `var_ffffc408`, `var_ffffc412`, ...). A program containing such names of variables is not very readable because two different variables with similar numbers may be mistakenly interchanged. Furthermore, such variable names do not say anything about their purpose. The following variable renamers are supported:
  - *Debugging information* – We generate the originally used names whenever these are available in the debugging information.
  - *More readable names* – Instead of hardly-readable names (e.g. `var_ffffc408`), we generate more readable variable names for arguments (e.g. `a1`, `a2`), global variables (e.g. `g1`, `g2`), and local variables (e.g. `v1`, `v2`).
  - *Renaming of an induction variable by a usual name* (e.g. `i`, `j`, `k`).
  - *Renaming of a variable containing a function's result to* `result`.
  - *Renaming of a variable containing a result of standard-function call* – Whenever some standard-library function returns a result, it is stored into a variable. Therefore, we can rename such a variable based on the called function, e.g. `len` for `strlen()` call.
- *Conversion of literals to symbolic constants* – Some of the standard-library functions expect numerical arguments that are specified as symbolic constants in headers by using `#define` or enumerations. However, the machine code only contains numeric values, which makes the decompiled code hardly readable. See the following example of a `socket()` function:

```
var_ffff7dc6 = socket(2, 3, 255);
```

Our solution is able to substitute these values back into their symbolic names based on the called function:

```
var_ffff7dc6 = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

Furthermore, we can use the aforementioned renamer of variables:

```
sock_id = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

6.7 Output Generation

The final phase of the decompiler is generation of the outputs. At present, we support generation of a call graph, control-flow graphs, and the HLL representation of the decompiled application. Furthermore, the decompiler also supports generation of a disassembled code that is produced by the front-end part, see Section 6.4.

6.7.1 Generation of Control-Flow Graphs

The back-end is able to emit one CFG for every particular decompiled function. Such CFG illustrates a relation within the function's basic blocks, see Definition 17. An example of such graph is depicted in Figure 6.42. It was generated by using a function containing a `switch` statement.

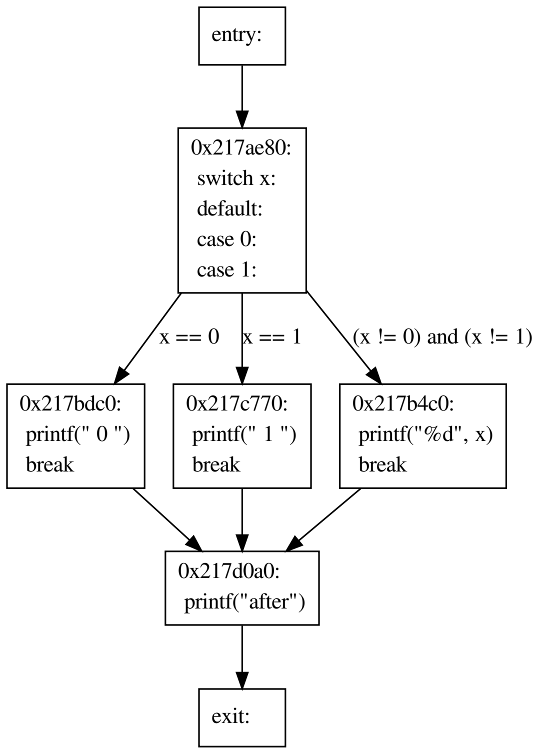


Figure 6.42: Example of a generated CFG for a function containing a `switch` statement.

6.7.2 Generation of Call Graphs

Furthermore, the back-end supports generation of a call graph, which depicts relations between functions (statically linked as well as dynamically linked), see Definition 18. An example is illustrated in Figure 6.43. Here, we can notice several user-defined functions calling each other and two other standard-library functions. The black nodes denote user-defined functions and the gray ones denote standard-library functions. Furthermore, whenever the debugging information is available, we are able to group functions from the same module (blue).

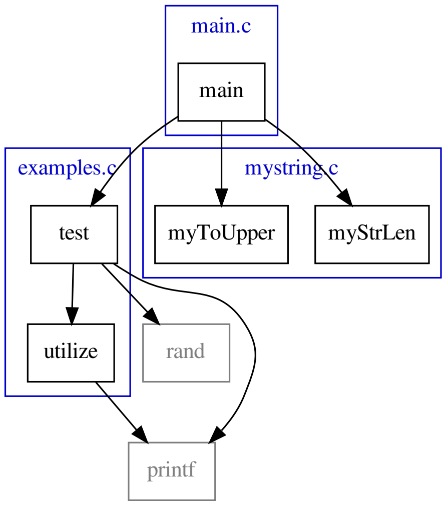


Figure 6.43: Example of a generated call graph.

6.7.3 Generation of the Output HLL

The decompiled application, in BIR code, is emitted in the chosen HLL representation. In Figure 6.44, we can see an illustration of a decompiled code in our Python-like language. The most important part is the section with decompiled functions (`main()` and `factorial()` in this case). As we can see, the decompiled file contains debugging information. Therefore, we were able to generate the original variable and function names, and comments related to placement of these functions within original modules. At the end of the listing, we also emit decompilation summary as a comment (decompiler version, number of decompiled functions, detected compiler based on the analysis described in Subsection 6.3.1, etc.).

Figure 6.45 illustrates decompilation of the same file[98], but into the C language. We can notice several differences. Firstly, the C output contains type information, which lacks in the Python output. Furthermore the C-code generator has to take care about emission of header includes for each standard-library function (e.g. `stdio.h` declaring `printf()`).

[98]For simplicity, comments were removed this time.

```
# ------------ Functions ---------------

# From module:   factorial.c
# Address range: 0x401560 - 0x401585
# Line range:    3 - 7
def factorial(n):
    if n != 0:
        x = factorial(n - 1) * n
    else:
        x = 1
    return x

# From module:   factorial.c
# Address range: 0x401587 - 0x4015bc
# Line range:    9 - 12
def main(argc, argv):
    printf("factorial(%d) = %d\n", 10, factorial(10))
    return 0

# --------- External Functions ---------

# int printf (const char *restrict, ...)

# --------- Meta-Information -----------

# Detected compiler: gcc (mingw32-x86-pe) 4.7.3
# Detected functions: 2 (2 in front-end)
# Decompiler release: v1.7 (Jun 30 2014)
# Decompilation date: Aug 02 2014 14:00:34
```

Figure 6.44: Example of a decompiled Python-like code.

```
#include <stdint.h>
#include <stdio.h>

int32_t factorial(int32_t n) {
    int32_t x;
    if (n != 0) {
        x = factorial(n - 1) * n;
    } else {
        x = 1;
    }
    return x;
}

int main(int argc, char **argv) {
    printf("factorial(%d) = %d\n", 10, factorial(10));
    return 0;
}
```

Figure 6.45: Example of a decompiled C code.

Chapter 7

Experimental Results

"It doesn't matter how beautiful your theory is.
If it doesn't agree with experiment, it's wrong."

Richard P. Feynman

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 (frequency 2.6 GHz, TurboBoost frequency 3.1 GHz, 6 cores, Hyper-Threading) 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, and HP VEX architectures were used as the target architectures. All of these architectures are described as instruction-accurate models in the ISAC language. A brief description of these architectures can be found in Subsection 3.1.2. The MIPS processor description (including FPU) is based on the MIPS32 Release 2 specification [58]. The ARM architecture model is based on the ARMv7-A processor type [3] and it includes the Thumb instruction set. The prototype of a PowerPC model is based on the instruction-set description v2.02 [37] and it lacks support of floating-point instructions. The Intel x86 model consists of user-mode instructions and x87 FPU instructions. The model does not contain a description of the x86-64 architecture or any instruction-set extensions such as SSE, AVX, or VMX. Finally, the ISAC model of the VLIW VEX processor is based on a description published in [25]. However, the original VEX specification does not contain description of instructions binary encoding. Therefore, we used encoding from the ρ-VEX project[99] [98]. The complexity of these ISAC models can be seen in Table 7.1.

Finally, we briefly describe the test-suite. In the following experiments, we use multiple existing benchmark suites (e.g. SPEC CINT2006[100] and MiBench[101]), compiler test-suites (e.g. GCC, LLVM, Perennial C Compiler Validation Suite), open-source applications

[99] `https://code.google.com/p/r-vex/`
[100] `http://www.spec.org/cpu2006/CINT2006/index.html`
[101] `http://www.eecs.umich.edu/mibench/`

Table 7.1: Complexity of the referred ISAC models in term of code lines. The auxiliary C code represents the code from ISAC `BEHAVIORAL` section that is stored in a separate file for better readability of the model.

	lines of code		
	ISAC	auxiliary C	total
PowerPC	2,500	500	**3,000**
VEX	2,500	1,500	**4,000**
MIPS	3,300	1,500	**4,800**
ARM	3,400	2,000	**5,400**
Intel x86	7,000	2,600	**9,600**

(e.g. `sgrep`, `sed`), as well as our own tests. Most of the source files are written in the C language, some of them are C++, and others in assembly, Delphi, or other languages.

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 [112, 113, 118, 119, 129], 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 architectures we chose MIPS and VEX in order to test simulation on the RISC and VLIW architectures, which are the most important for this tool[102]. Benchmarking applications are part of the MiBench test-suite (e.g. crc32, sha, dijkstra). 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.

Figures 7.1 and 7.2 show a comparison of simulation speeds for different simulation types by using MiBench algorithms and custom SMC. The former figure depicts results for the MIPS architecture and the second one depicts results for VEX architecture. 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`).

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

[102] To be clear, we are not focused on the x86 simulation running on the x86-based testing machine because there are other projects focused directly on this particular combination of host and target platform, e.g. the aforementioned Bochs project that uses a direct virtualization of instructions.

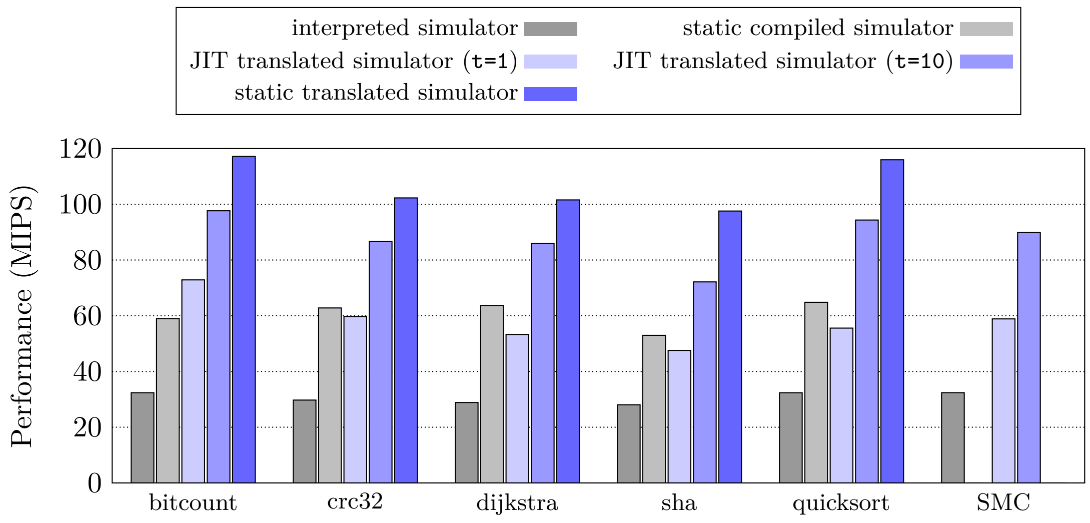


Figure 7.1: Performance comparison of all simulator types (MIPS architecture).

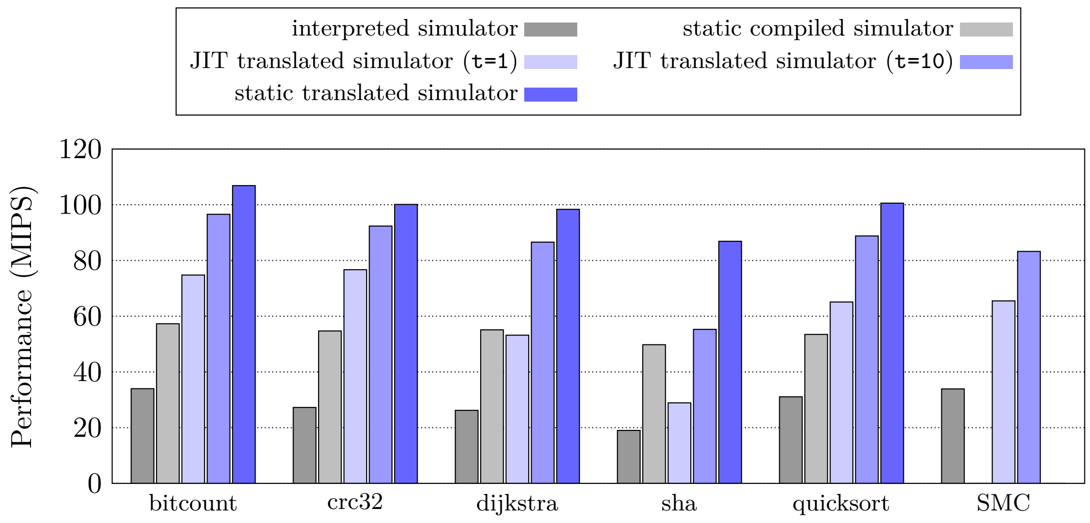


Figure 7.2: Performance comparison of all simulator types (VEX architecture).

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. Furthermore, this simulation type can handle applications containing SMC.

Furthermore, we focused on the time needed for creation of these tools. We used the aforementioned test-suite and averaged the creation times (we used threshold `t=10` for the JIT translated simulator). These values reflect the creation time of the analyser (only for static compiled simulation and static translated simulation) and creation of the main simulator (for all simulator types). The results are depicted in Table 7.2.

Table 7.2: Times needed for simulator creation (in seconds).

	MIPS		VEX	
	analyser	simulator	analyser	simulator
interpreted simulator	-	9.56	-	21.25
static compiled simulator	25.10	3.02	259.41	3.68
JIT translated simulator	-	33.82	-	189.57
static translated simulator	25.23	3.11	259.83	3.82

Once again, we can make the following conclusions based on these results. (1) The time needed for generation of an interpreted simulator is constant because it is application independent. Furthermore, creation of the static compiled simulator is dependent on a particular application, but it can be up to 8 times faster than the interpreted simulator. However, it is necessary to generate the analyser at first. This process may take several minutes, which is a drawback of this simulation type. (2) The times needed for creation of the newly proposed static translated simulator (and its analyser) are almost the same as in the static compiled simulation. (3) Within the last simulation type, the JIT translated simulation, there is no analyser-creation phase, but the time needed for simulator generation is longer than in the other types.

As we discussed in Section 4.6, there is only one competitive project that also supports the concept of the static translated simulation – xADL. However, this solution is not publicly available and we can only compare our results based on the xADL results published in [10]. In this paper, the authors used different set of target architectures (MIPS and CHILI) and test applications (crc32, jpeg, sha, etc.). Because we cannot re-test their solution with our own, we can only compare the results of the same tests that both of our solutions were tested on (i.e. MIPS architecture and bitcount, crc32, dijkstra, and sha tests). Furthermore, the xADL tests were performed on a much slower machine (AMD Athlon 64, 2.2 GHz, 1 GB RAM running a 32-bit Linux-based operating system) than our solution and a direct comparison will be unfair. We do not have a machine with the same configuration for re-testing of our solution. Therefore, we re-tested our solution on a similar configuration (Intel Core 2 Quad with 2.8 GHz, 4 GB RAM running a 32-bit Linux-based operating system). Therefore, the comparison depicted in Figure 7.3 is only illustrative because it may be inaccurate.

The xADL authors claim (see [10]) that an average speed of their solution on MIPS architecture is 43 MIPS. However, we have obviously chosen the tests that are faster for xADL because the average simulation speed for these tests is 73 MIPS. Contrariwise, our solution achieved an average speed of 94 MIPS. Because of the inaccuracy of the measurement, we can only conclude by stating that our concept of static translation simulation is fully competitive with the existing solution.

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.

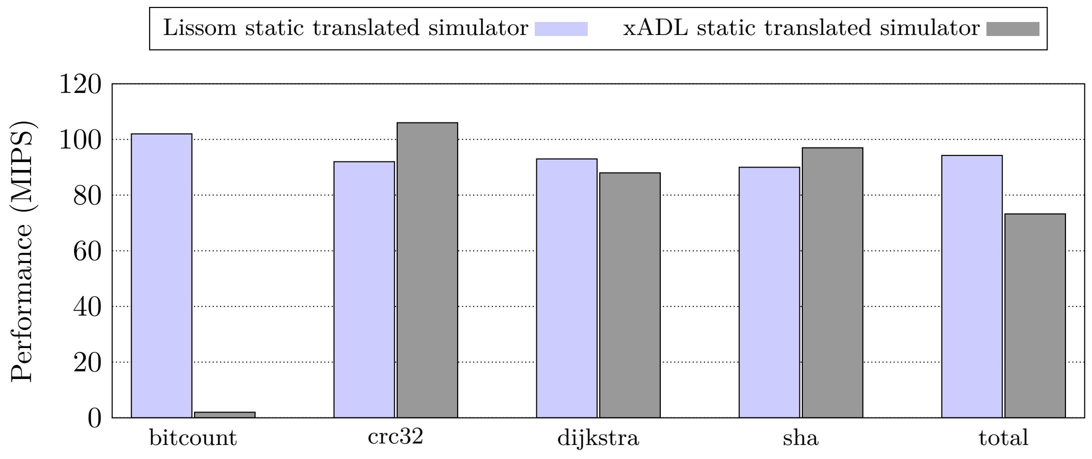


Figure 7.3: Performance comparison between Lissom and xADL translated simulators.

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 once again used the MIPS and VEX processor architectures 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. We used the automatically generated interpreted and compiled simulators. We cannot use the translated simulator in this scenario because the debugging information is not available, but, as we will see, the speed drop-off ratio is independent of the simulator type.

(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[103], see [113].

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), i.e. the breakpoint was ignored for a defined number of hits. The breakpoints were set on frequently-executed instructions, such as loops.

Figures 7.4 and 7.5 show the comparison of simulation speeds for the MIPS and VEX architecture respectively.

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. These values depend on the application type and the breakpoint position.

[103]Based on the same speed-measuring metric, the usage of an invalid opcode implementation is about 10% slower than the address-checking solution.

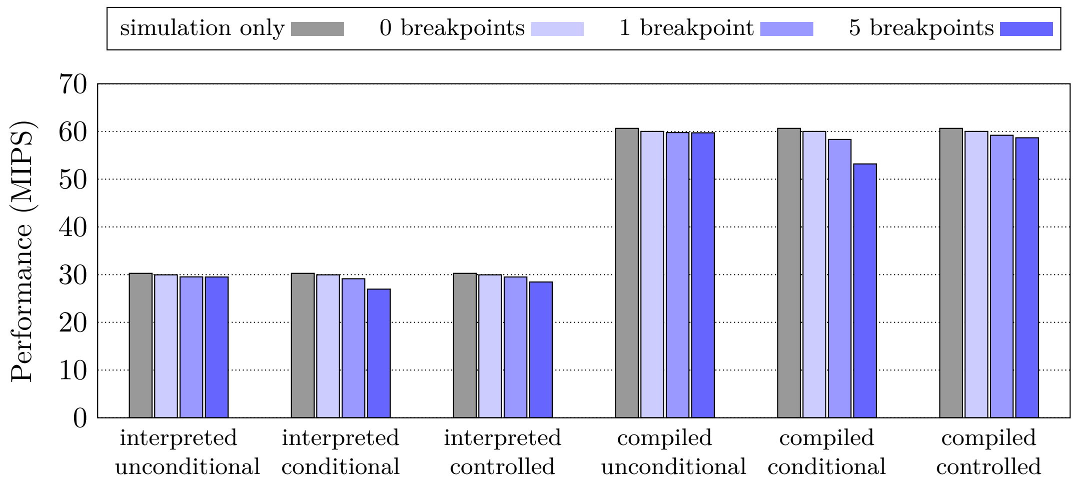


Figure 7.4: Simulator speeds of MIPS processor in different debugging scenarios.

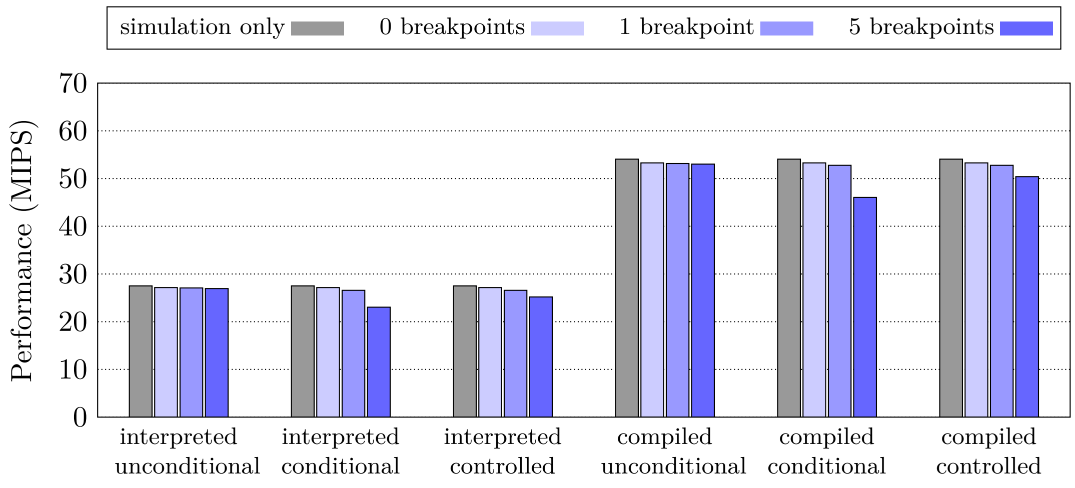


Figure 7.5: Simulator speeds of VEX processor in different debugging scenarios.

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 mentioned in Section 4.1.

All of these detection tools use the same approach as our solution—detection by using signature matching. As we can see in Table 7.3, most of them use a combination of pre-compiled internal signatures and a large external database created by the user community. The competitive solutions (except of DiE) are limited to WinPE EFF and a number of their signatures varies between hundreds and thousands. The number of internal signatures is not always absolutely precise because some authors do not specify this number, like RDG or FastScanner. Therefore, we had to analyse such applications and try to find their databases manually (e.g. by using reverse engineering). We were unable to find this information in the RDG and DiE detectors. Our solution consists of 2,282 internal signatures for all supported EFFs and we also support the concept of external signatures.

Table 7.3: Overview of existing compiler/packer detection tools.

tool		**signatures**		
name	version	internal	external	total
`fileinfo`	1.8	2,282	0	2,282
RDG Packer Detector[104]	0.7.2	?	10	?
ProtectionID (PID)[105]	0.6.5.5	543	0	543
Exeinfo PE[106]	0.0.3.4	718	7,076	7,794
Detect It Easy (DiE)[107]	0.81	?	2,100	?
NtCore PE Detective[108]	1.2.1.1	0	2,806	2,806
FastScanner[109]	3.0	1,605	1,832	3,437
PEiD[110]	0.95	672	1,774	2,446

We compared these detectors in three test sets: (1) WinPE packers, (2) WinPE polymorphic packers, and (3) ELF packers and compilers.

WinPE Packers

In total, 40 WinPE packers (e.g. ASPack, FSG, UPX, PECompact, EXEStealth, MEW) and several of their versions (107 different tools in total) were used for comparison of the

[104] http://rdgsoft.8k.com/
[105] http://pid.gamecopyworld.com/
[106] http://www.exeinfo.xwp.pl/
[107] http://hellspawn.nm.ru/
[108] http://www.ntcore.com/
[109] http://www.at4re.com/
[110] http://www.peid.info/

aforementioned detectors. We used these packers for packing several compiler-generated executables—with different sizes (from 50 kB up to 5 MB), used compiler, compilation options, and packer options. The purpose is that some packers create different start-up code based on the file size and characteristics (data-section size, PE header flags, etc.). 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 type of detection is the most common and also the easiest to implement because generic signatures can be applied (i.e. signatures with only a few fixed nibbles describing complete packer family). On the other hand, 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 7.6.

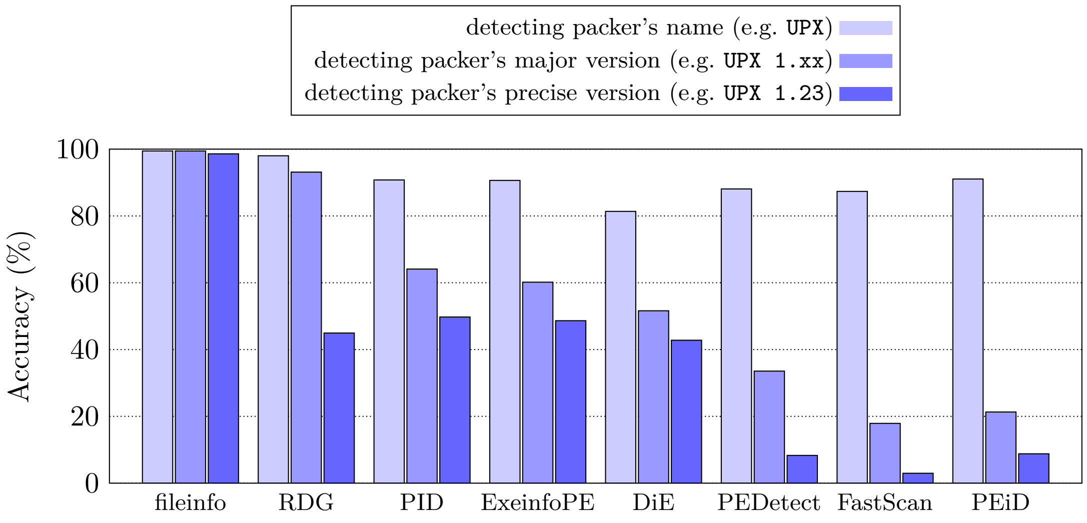


Figure 7.6: Comparison of packer-detection accuracy.

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%. We can also notice that larger signature databases do not imply better results in this category (e.g. Exeinfo PE). Such large databases are hard to maintain and they can produce incorrect results because of too many generic signatures.

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. For example, the compiler may use in its newer versions more aggressive optimizations that have a very specific meaning and they need special attention by the decompiler (e.g. loops transformation, jump tables), see Subsection 6.3.1 for an example. The results are depicted in the second data set of Figure 7.6.

Within this test case, our solution and RDG once again achieved the best results (`fileinfo` scored 99%, RDG scored 93%). None of the other detectors exceeded 80% and only ExeinfoPE and ProtectionID achieved more than 60%.

Finally, we tested the ratio of a precise-version detection. This task is the most challenging because it is necessary to create one signature for each particular version of each particular packer. 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 7.6, our detector achieved the best results in this category with 98% accuracy. The results of the other solutions were much lower (50% at most). This is mainly because we focus primarily on detecting the precise version and we also support search in the entire PE file and its overlay and not just on its entry point. Furthermore, our solution also uses several detection heuristics.

WinPE Polymorphic Packers

The second test set is focused on detection of one of the most common WinPE polymorphic packers/encryptors called Morphine. We used two versions of this packer (v1.2 and v2.7) and we created the test set consisting of 339 executable files. We used the same testing methodology as in the previous case, but only three detectors (`fileinfo`, RDG, and PEiD) were able to detect this packer. Therefore, other detectors were excluded from the results depicted in Table 7.4.

Table 7.4: Comparison of Morphine-encryptor detection (in percents).

	type of detection test		
	name	major version	precise version
`fileinfo`	100.00	100.00	100.00
RDG	94.69	27.73	27.73
PEiD	59.88	59.88	35.10

By using reverse engineering, we figured out that PEiD and RDG detectors use additional heuristic techniques that are similar to our solution described in Subsection 6.3.1. By using these heuristics, PEiD and RDG detectors are able to detect these polymorphic packers like the Morphine encryptor.

As we can see, our heuristics detection is the most successful (with a ratio of 100% in all cases). The RDG detector exceeded a 90% success ratio in detection of a packer name, but its results are poor in other cases. Not even PEiD has achieved good results.

ELF Packers and Compilers

The last test set is focused on detection of ELF compilers (e.g. GCC) and packers (ELFCrypt, UPX)—we used 18 tools and 197 files in total. Only two detectors (`fileinfo` and DiE) support processing of ELF EFF files. Thus, only these detectors were tested. The results are compared in Table 7.5.

Table 7.5: Comparison of ELF compilers and packers detection (in percents).

	type of detection test		
	name	major version	precise version
`fileinfo`	98.98	98.98	87.31
DiE	69.04	4.57	4.57

Even in this test suite, our tool had the most accurate results in all cases. Poor performance of the DiE detector in two from three test categories is probably caused by a small number of signatures for ELF EFF.

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) that is a plugin to the IDA disassembler (v6.6), see Section 4.4. The Hex-Rays Decompiler is not an automatically generated retargetable decompiler, such as our solution, and it 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. The default settings of both decompilers were used in all tests.

For testing, we used several open-source applications (e.g. `sgrep`, `sed`) and algorithms from benchmarking test-suites (e.g. SPEC CINT2006 and MiBench). The source files of all tests are written in the C language.

In order to keep the same test conditions on all architectures, we used the GNU compiler GCC of different versions compiling into the ELF EFF. More precisely, the Minimalist PSPSDK compiler[111] (version 4.3.5) was used for compiling MIPS binaries, GCC (version 4.5.1) was used for compiling tests for the PowerPC architecture, the GNU ARM toolchain[112] (version 4.1.1) for ARM binaries, and the GNU compiler GCC (version 4.7.2) for x86 executables (the 32-bit mode was forced by the `-m32` option).

Different compiler options were used in particular test cases; e.g. with or without symbolic information, optimizations, and different linkage types. To preserve the number of functions from the source code in the executable, function inlining was always turned off.

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. Results of these three test cases are described in the following text.

(1) The first test case was focused on the control-flow analysis and function detection and recovery from real-world applications. The test C files were compiled only with the `-O1` optimizations level in order to avoid automatic function inlining. Furthermore, the executable files were stripped (`-s`) in order to remove any additional information that can be used for function extraction (e.g. symbols, debugging information). Finally, the dynamic linkage of library code was used whenever available in order to decompile only user functions and not the standard functions that were included by compiler. The results are depicted in Table 7.6.

Based on the results, the function-recovery accuracy of our retargetable decompiler varies between 90.4% (PowerPC) and 92.7% (ARM). As we can see, the proposed method of function detection is highly accurate on all tested architectures, but there is still a room for improvement related to not-so-common constructions (e.g. `rep ret` instruction for function

[111] http://sourceforge.net/projects/minpspw/
[112] http://www.gnuarm.com/

Table 7.6: Accuracy of function detection and reconstruction (in percents).

		Lissom				**Hex-Rays**	
	functions	MIPS	PPC	ARM	x86	ARM	x86
blowfish	4	100.0	100.0	100.0	100.0	100.0	100.0
bzip2	84	86.9	83.3	84.5	86.9	82.1	90.5
dos2unix	14	100.0	100.0	100.0	100.0	100.0	100.0
gzip	67	94.0	89.6	94.0	91.0	100.0	97.0
rijndael	7	100.0	100.0	100.0	100.0	100.0	100.0
sed	24	91.7	95.8	91.7	87.5	91.7	87.5
sgrep	91	96.7	96.7	96.7	96.7	98.9	98.9
sha	8	100.0	100.0	100.0	100.0	100.0	100.0
strsearch	5	100.0	100.0	100.0	100.0	100.0	80.0
strp	10	70.0	50.0	90.0	70.0	70.0	90.0
total	**314**	**92.7**	**90.4**	**92.7**	**91.7**	**93.6**	**94.9**

return[113] on Intel x86, function pointers) and optimizations (e.g. tail-call optimization).

The Hex-Rays Decompiler achieved even better accuracy in the same test case – 93.6% for ARM and 94.9% for Intel x86, which makes a 2 percentage point difference between averaged results of both decompilers.

(2) The detection and recovery of function arguments was evaluated in the second test case. To eliminate errors in function detection and to avoid situations where the used compiler eliminated unused arguments, we used non-stripped binary files generated without optimizations (i.e. `-O0`) and without debugging information. In this case, both decompilers detected symbolic information and they used this information instead of function detector tested in (1). However, the symbols contained no information about function arguments. Therefore, this information had to be retrieved via the data-flow analysis described in Section 6.4.

We evaluated this test case twice. (a) Firstly, the testing methodology validated only the correct number of function arguments, not their types. The results are illustrated in Table 7.7.

As we can see, the results on PowerPC are inaccurate in comparison with the remaining architectures (20% vs. up to 98% accuracy for Intel x86). As we previously stated, the support of PowerPC is currently in a prototype phase and the data-flow analysis has to be tweaked appropriately. Currently, the PowerPC link register `LR` is falsly detected as the first argument in most cases, which ruins the total accuracy. When excluding the PowerPC results, our solution was able to detect and reconstruct 91% of all arguments. The Hex-Rays Decompiler was successful in 88% of all evaluated tests (82% for ARM and 93% for Intel x86).

(b) Afterwards, we re-evaluated the same test with taking into account the basic data types of arguments: integer numbers, floating point numbers, and pointers. The bit width and signedness are omitted and each particular test is successful only if the number of arguments is correct and all their basic data types matches the original basic data types, which makes this test case harder than the previous one. The results are depicted in Table 7.8.

[113] http://repzret.org/p/repzret/

Table 7.7: Accuracy of arguments reconstruction without data types (in percents).

		Lissom				**Hex-Rays**	
	functions	MIPS	PPC	ARM	x86	ARM	x86
blowfish	4	50.0	75.0	75.0	100.0	100.0	100.0
bzip2	84	91.7	13.1	85.7	98.8	83.3	86.9
dos2unix	14	92.9	28.6	85.7	100.0	100.0	100.0
gzip	67	92.5	17.9	85.1	97.0	94.0	97.0
rijndael	7	71.4	71.4	71.4	100.0	100.0	100.0
sed	24	91.7	37.5	83.3	100.0	87.5	100.0
sgrep	91	95.6	11.0	81.3	97.8	62.6	93.4
sha	8	62.5	87.5	37.5	100.0	100.0	100.0
strsearch	5	100.0	60.0	100.0	100.0	100.0	80.0
strp	10	100.0	10.0	90.0	100.0	100.0	100.0
total	**314**	**91.7**	**20.7**	**82.8**	**98.4**	**82.5**	**93.6**

Table 7.8: Accuracy of arguments reconstruction including data types (in percents).

		Lissom				**Hex-Rays**	
	functions	MIPS	PPC	ARM	x86	ARM	x86
blowfish	4	50.0	25.0	75.0	100.0	25.0	25.0
bzip2	84	75.0	9.5	65.5	76.2	22.6	25.0
dos2unix	14	57.1	14.3	42.9	100.0	42.9	50.0
gzip	67	67.2	10.5	64.2	71.6	64.2	67.2
rijndael	7	14.3	14.3	28.6	85.7	14.3	14.3
sed	24	70.8	16.7	58.3	79.2	37.5	37.5
sgrep	91	62.6	7.6	50.5	74.7	20.9	39.5
sha	8	50.0	12.5	25.0	100.0	12.5	12.5
strsearch	5	40.0	40.0	40.0	100.0	20.0	20.0
strp	10	60.0	10.0	50.0	60.0	70.0	70.0
total	**314**	**65.3**	**10.8**	**56.7**	**77.1**	**34.1**	**41.1**

Because of the aforementioned reasons, the results of PowerPC are once again inaccurate. On the other hand, the remaining architectures achieved quite a high accuracy (between 55% and 77%) via the type-recovery analysis presented in [53]. Contrariwise, the Hex-Rays Decompiler achieved only 37% accuracy in total. It should be noted that both decompilers still have a lot of room for improvement in terms of type detection.

(3) Finally, the recovery of return values was tested on the same conditions as in (2a). Each test was rated as successful only if the tested decompiler correctly distinguished between `void` and non-`void` return values. Although it may seem a simple test, it is quite challenging according to the results listed in Table 7.9.

As we can see, the deviation between all decompilation tests was minimal (50% to 55%). The most problematic part for both decompilers was decompilation of `void` functions (i.e. functions without a return value), which were detected as returning an `int` value. Normally, the detection of the return value is done by detection of a write into the return

Table 7.9: Accuracy of return-values reconstruction (in percents).

		Lissom				**Hex-Rays**	
	functions	MIPS	PPC	ARM	x86	ARM	x86
blowfish	4	25.0	25.0	25.0	25.0	25.0	25.0
bzip2	84	47.6	47.6	47.6	47.6	47.6	47.6
dos2unix	14	64.3	64.3	64.3	64.3	64.3	64.3
gzip	67	52.2	52.2	52.2	52.2	53.7	55.2
rijndael	7	85.7	85.7	85.7	85.7	85.7	85.7
sed	24	62.5	62.5	62.5	62.5	62.5	62.5
sgrep	91	56.0	56.0	56.0	56.0	42.9	59.3
sha	8	12.5	12.5	12.5	12.5	12.5	12.5
strsearch	5	60.0	60.0	60.0	60.0	60.0	60.0
strp	10	60.0	60.0	60.0	60.0	60.0	60.0
total	**314**	**53.2**	**53.2**	**53.2**	**53.2**	**49.7**	**54.8**

register based on a particular API (e.g. `eax` on Intel x86 for returning 32-bit integer values) in the data-flow analysis. However, this approach may be inaccurate whenever this register is used for any other reason (e.g. `eax` register also serves as accumulator, for division and multiplication). Based on the results, the data-flow analysis of the retargetable decompiler works in the same way for all architectures. On the other hand, Hex-Rays' results differ on both supported architectures.

Finally, Figure 7.7 depicts the overall results of the previous 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.

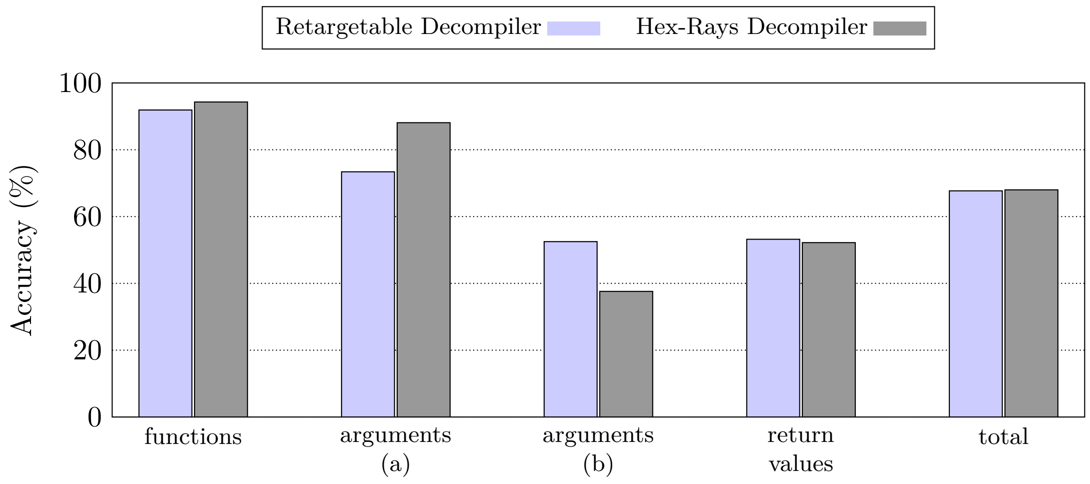


Figure 7.7: Total accuracy of function, argument, and return value reconstruction.

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. We have chosen the Hex-Rays decompiler because it achieved the best idiom-detection accuracy among the existing decompilers, see Section 4.4. A brief comparison of the other decompilers is available in [112].

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.

Finally, we enabled the emission of debugging information in the DWARF standard by using the `-g` option because both decompilers exploit this information to produce a more accurate code, see [109] for details. The debugging information helps to eliminate inaccuracy of decompilation (e.g. entry-point detection, function reconstruction) that may influence testing. However, the debugging information does not contain information about the usage of the idioms and therefore, its usage does not affect the idiom-detection accuracy.

All test cases are listed in Table 7.10. The results of our retargetable decompiler are marked as *RD* and the results of the Hex-Rays decompiler are marked as *HR*.

The first column represents the description of a particular idiom used within the test. The maximum number of points for each test on each architecture is five (i.e. one point for each optimization level—`O0`, `O1`, `O2`, `O3`, and `Os`). Some idioms are not used by compilers based on the optimization level or target architecture. Therefore, the number of total points can be lower than five. For example, the MIPS and ARM architectures lack a floating-point unit (FPU) and the essential FPU operations are emulated via *soft-float* idioms. On the other hand, the Intel x86 architecture implements these operations via the x87 floating-point instruction extension. Therefore, the instruction idioms are not used in this case.

We can observe two basic facts based on the results.

(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

Table 7.10: Number of successfully detected and reconstructed instruction idioms (in percents). Note: several tests differ only in the used numeric constant; however, different instruction idioms are emitted based on this value.

			MIPS		**PowerPC**		**ARM**			**Intel x86**		
			tests	**RD**	tests	**RD**	tests	**RD**	**HR**	tests	**RD**	**HR**
`intA`	=	`intB < 0`	5	100.0	5	100.0	5	100.0	0.0	5	100.0	0.0
`intA`	=	`intB >= 0`	5	100.0	5	100.0	5	100.0	0.0	5	100.0	0.0
`intA`	=	`0`	0	-	0	-	0	-	-	1	100.0	100.0
`intA`	=	`intB * 4`	5	100.0	5	100.0	5	100.0	100.0	5	100.0	100.0
`intA`	=	`intB / -2`	0	-	5	100.0	5	100.0	100.0	4	100.0	100.0
`intA`	=	`intB / 4`	1	100.0	5	100.0	5	100.0	100.0	4	100.0	100.0
`intA`	=	`intB / 10`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`intA`	=	`intB / 120`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`uintA`	=	`uintB / 7`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`uintA`	=	`uintB / 9`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`intA`	=	`-intB - 1`	5	100.0	5	100.0	5	100.0	0.0	5	100.0	0.0
`intA`	=	`intB % 2`	0	-	5	100.0	5	80.0	40.0	4	100.0	0.0
`intA`	=	`intB % 3`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`intA`	=	`intB % 5`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`intA`	=	`intB % 8`	0	-	5	100.0	4	100.0	75.0	4	100.0	0.0
`uintA`	=	`uintB % 3`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`uintA`	=	`uintB % 5`	0	-	4	100.0	4	100.0	100.0	4	100.0	100.0
`uintA`	=	`uintB % 8`	5	100.0	5	100.0	5	100.0	0.0	5	100.0	0.0
`floatA`	=	`-floatB`	5	100.0	5	0.0	5	100.0	0.0	0	-	-
`floatA`	=	`copysign(floatB, floatC)`	5	100.0	5	0.0	5	100.0	0.0	0	-	-
`floatA`	=	`fabs(floatB)`	5	100.0	5	0.0	5	100.0	0.0	0	-	-
total			**41**	**100.0**	**92**	**83.7**	**91**	**98.9**	**57.1**	**74**	**100.0**	**62.2**

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%). As we stated in the introduction of this chapter, 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. The overall decompilation results are depicted in Figure 7.8.

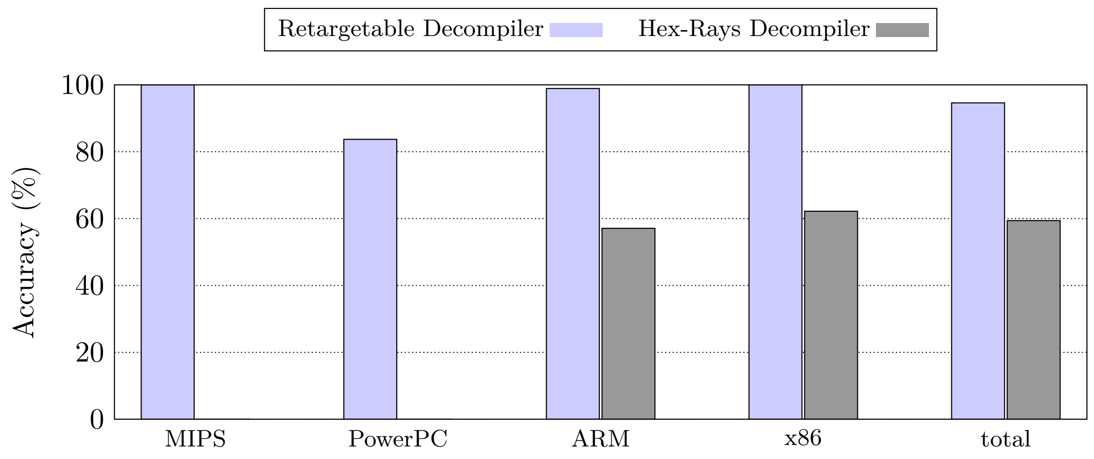


Figure 7.8: Accuracy of instruction-idioms detection and code reconstruction. Note: the total accuracy of Hex-Rays decompiler is calculated based on ARM and Intel x86 only.

Chapter 8

Conclusion

“The outcome of any serious research can only be to make two questions grow where only one grew before.”

Thorstein B. Veblen

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[114] 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[115].

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

[114] 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/

[115] http://qemu.org

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.

(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 [107].

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 [53].

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 [125].

(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 [106] and [47]. 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. ISBN 0-321-48681-1.

[2] Apple Inc. *Mac OS X ABI Mach-O File Format Reference*, 2009.

[3] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*, ARM DDI 0406C edition, 2011. https://silver.arm.com/download/download.tm?pv=1199569.

[4] K. Babar and F. Khalid. *Generic Unpacking Techniques*. In 2nd International Conference on Computer, Control and Communication, pages 1–6, 2009. ISBN 978-1-4244-3314-8.

[5] B. Bailey. *ESL Design and Verification: A Prescription for Electronic System-Level Methodology*. Morgan Kaufmann Publishers, San Francisco, US-CA, 2007. ISBN 978-0123735515.

[6] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms, and Applications*. Springer-Verlag, London, GB, 2001. ISBN 1-85233-268-9.

[7] P. Barbe. *The PILER System of Computer Program Translation*. Technical Report, Probe Consultants Inc, 1974.

[8] N. Bermudo, A. Krall, and N. Horspool. *Control Flow Graph Reconstruction for Assembly Language Programs with Delayed Instructions*. In 5th IEEE International Workshop on Source Code Analysis and Manipulation SCAM'05, pages 107–118, Washington, US-DC, 2005. IEEE Computer Society. ISBN 0-7695-2292-0.

[9] F. Brandner. *Compiler Backend Generation from Structural Processor Models*. PhD Thesis, Vienna University of Technology, Vienna, AT, 2009.

[10] F. Brandner, A. Fellnhofer, A. Krall, and D. Riegler. *Fast and Accurate Simulation Using the LLVM Compiler Framework*. In 1st Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools (RAPIDO'09), pages 1–6, Paphos, CY, 2009.

[11] T. Brosch and M. Morgenstern. *Runtime Packers: The Hidden Problem?* In Black Hat, 2006.

[12] S. Chamberlain. *The Binary File Descriptor Library*. Iuniverse Inc., 2000. ISBN 9780595136193.

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

[14] C. Cifuentes and S. Sendall. *Specifying the Semantics of Machine Instructions.* In 6th International Workshop on Program Comprehension (IWPC'98), pages 126–133, Washington, US-DC, 1998. IEEE Computer Society. ISBN 0-8186-8560-3.

[15] J. Cocke. *Global Common Subexpression Elimination.* SIGPLAN Not., 5(7):20–24, 1970. ISSN 0362-1340.

[16] 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. ISBN 978-0-7695-3429-9.

[17] DroneBL. *Network Bluepill – Stealth Router-Based Botnet Has Been DDoSing DroneBL For the Last Couple of Weeks.* http://dronebl.org/blog/8, 2009.

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

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

[20] E. Eilam. *Reversing: Secrets of Reverse Engineering.* Wiley, 2005.

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

[22] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoll, and F. Homewood. *Lx: A Technology Platform for Customizable VLIW Embedded Processing.* In 27th International Symposium on Computer Architecture (ISCA'00), pages 203–213, New York, US-NY, 2000. IEEE Computer Society.

[23] A. Fauth, J. Van Praet, and M. Freericks. *Describing Instruction Set Processors Using nML.* In European Conference on Design and Test (DATE'95), pages 503–507, Washington, US-DC, 1995. IEEE Computer Society. ISBN 0-8186-7039-8.

[24] M. F. Fernández. *Simple and Effective Link-Time Optimization of Modula-3 Programs.* SIGPLAN Not., 30(6):103–115, 1995. ISSN 0362-1340.

[25] J. A. Fisher, P. Faraboschi, and C. Young. *Embedded Computing a VLIW Approach to Architecture, Compilers and Tools.* Morgan Kaufmann Publishers, San Francisco, US-CA, 2005. ISBN 1-55860-766-8.

[26] G. R. Gircys. *Understanding and Using COFF.* O'Reilly & Associates, Inc., Sebastopol, US-CA, 1988. ISBN 0937175315.

[27] S. Greibach and J. Hopcroft. *Scattered Context Grammars.* Journal of Computer and System Sciences, 3(3):233–247, 1969. ISSN 0022-0000.

[28] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. *EXPRESSION: A Language for Architecture Exploration through Compiler/Simulator Retargetability.* In Design, Automation, and Test in Europe (DATE'99), pages 31–45, New York, NY, USA, 1999. ACM. ISBN 1-58113-121-6.

[29] M. H. Halstead. *Machine-Independent Computer Programming.* Spartan Books, 1962. ISBN B0006AXVVE.

[30] B. Hayes. *The Discovery of Debugging.* The Sciences, 33(4):10–13, 1993. ISSN 2326-1951.

[31] K. Hazelwood and A. Klauser. *A Dynamic Binary Instrumentation Engine for the ARM Architecture.* In International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES'06), pages 261–270, Seoul, KR, 2006. ACM. ISBN 1-59593-543-6.

[32] A. Hoffmann, H. Meyr, and R. Leupers. *Architecture Exploration for Embedded Processors with LISA.* Kluwer Academic Publishers, 2002. ISBN 978-1402073380.

[33] P. Hohensee, M. Myszewski, and D. Reese. *Wabi CPU Emulation.* Hot Chips VIII, 1996.

[34] T. Hruška. *Instruction Set Architecture C (ISAC).* Internal Document, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2012.

[35] T. Hruška, D. Kolář, R. Lukáš, and E. Zámečníková. *Two-Way Coupled Finite Automaton and Its Usage in Translators.* Computer Science Challenges, 2008(07):445–449, 2008. ISSN 1790-5117.

[36] 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. ISBN 978-80-87342-10-7.

[37] IBM. *PowerPC User Instruction Set Architecture: Book I*, Version 2.02 edition, 2005. http://www.ibm.com/developerworks/systems/library/es-archguide-v2.html.

[38] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, 2013. http://download.intel.com/products/processor/manual/253665.pdf.

[39] D. Kästner and S. Wilhelm. *Generic Control Flow Reconstruction From Assembly Code.* ACM SIGPLAN Notices, 37(7), 2002. ISSN 03621340.

[40] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach.* Morgan Kaufmann Publishers, San Francisco, US-CA, 2002. ISBN 1-55860-286-0.

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

[42] J. Kinder and H. Veith. *Jakstab: A Static Analysis Platform for Binaries.* In Computer Aided Verification, volume 5123 of Lecture Notes in Computer Science, pages 423–427. Berlin, Heidelberg, DE, 2008. ISBN 978-3-540-70543-7.

[43] J. Kinder, F. Zuleger, and H. Veith. *An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries.* In 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09), pages 214–228, Berlin, Heidelberg, DE, 2009. Springer-Verlag. ISBN 978-3-540-93899-6.

[44] D. Kolář. *Scattered Context Grammars Parsers.* In 14th International Congress of Cybernetics and Systems of WOCS, pages 491–500, Wroclaw, PL, 2008. Wroclaw University of Technology, PL. ISBN 978-83-7493-400-8.

[45] 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.

[46] D. Kolář and A. Meduna. *Regulated Pushdown Automata.* Acta Cybernetica, 2000(4):653–664, 2000. ISSN 0324-721X.

[47] 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.

[48] J. Křoustek. *Code Analysis and Transformation.* Bachelor's Thesis, Brno University of Technology, Faculty of Information Technology, 2007.

[49] J. Křoustek. *Code Analysis and Transformation to a High-Level Language.* Master's Thesis, Brno University of Technology, Faculty of Information Technology, 2009.

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

[51] J. R. Levine. *Linkers and Loaders.* Operating Systems. Morgan Kaufmann Publishers, 2000. ISBN 9781558604964.

[52] K. Masařík. *System for Hardware-Software Co-Design.* VUTIUM. Brno University of Technology, Faculty of Information Technology, Brno, CZ, 1st edition, 2008. ISBN 978-80-214-3863-7.

[53] 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.

[54] A. Meduna. *Automata and Languages: Theory and Applications.* Springer-Verlag, London, GB, 2005. ISBN 1-85233-074-0.

[55] A. Meduna and J. Techet. *Scattered Context Grammars and their Applications.* WIT Press, Southampton, GB, 2010. ISBN 978-1-84564-426-0.

[56] F. Merz, C. Sinz, H. Post, T. Gorges, and T. Kropf. *Abstract Testing: Connecting Source Code Verification with Requirements.* In Quality of Information and Communications Technology (QUATIC'10), pages 89–96. IEEE Computer Society, 2010. ISBN 978-1-4244-8539-0.

[57] Microsoft Corporation. *Microsoft Portable Executable and Common Object File Format Specification.* http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx, 2014. Version 8.3.

[58] MIPS Technologies Inc. *MIPS32 Architecture for Programmers Volume II-A: The MIPS32 Instruction Set*, MIPS MD00086 edition, 2010. https://www.mips.com/products/architectures/mips32/.

[59] P. Mishra and N. Dutt. *Processor Description Languages: Applications and Methodologies.* Morgan Kaufmann Publishers, San Francisco, US-CA, 2008. ISBN 978-0123742872.

[60] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, San Francisco, US-CA, 1997. ISBN 1558603204.

[61] M. Murdocca and V. P. Heuring. *Principles of Computer Architecture.* Prentice Hall, Upper Saddle River, US-NJ, 2000. ISBN 978-0201436648.

[62] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, Berlin, Heidelberg, DE, 1999. ISBN 978-3540654100.

[63] G. Nolan. *Decompiling Java.* Apress, Berkeley, US-CA, 2004. ISBN 978-1590592656.

[64] G. Nolan. *Decompiling Android.* Apress, Berkeley, US-CA, 2012. ISBN 978-1430242482.

[65] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. *Pinpointing Representative Portions of Large Intel Itanium Programs with Dynamic Instrumentation.* In 37th International Symposium on Microarchitecture (MICRO'04), pages 81–92, Portland, US-OR, 2004. IEEE Computer Society.

[66] D. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann Publishers, Burlington, US-MA, 4 edition, 2008. ISBN 0123744938.

[67] D. A. Patterson. *Computer Organization and Design the Hardware/Software Interface.* Morgan Kaufmann Publishers, San Francisco, US-CA, 2007. ISBN 978-0123706065.

[68] A. Podgurski and L. A. Clarke. *A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance.* IEEE Transactions on Software Engineering, 16(9):965–979, 1990. ISSN 0098-5589.

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

[70] Z. Přikryl, T. Hruška, K. Masařík, and A. Husár. *Fast Cycle-Accurate Compiled Simulation.* In 10th IFAC Workshop on Programmable Devices and Embedded Systems (PDeS'10), pages 97–102. IFAC, 2010. ISBN 978-3-902661-95-1.

[71] 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. ISBN 978-0-7695-4000-9.

[72] Z. Přikryl, K. Masařík, T. Hruška, and A. Husár. *Generated Cycle-Accurate Profiler for C Language.* In 13th EUROMICRO Conference on Digital System Design (DSD'2010), pages 263–268. IEEE Computer Society, 2010. ISBN 978-0-7695-4171-6.

[73] D. A. Ramos and D. R. Engler. *Practical, Low-Effort Equivalence Verification of Real Code.* In 23rd International Conference on Computer Aided Verification (CAV'11), pages 669–685, Berlin, Heidelberg, DE, 2011. Springer-Verlag. ISBN 978-3-642-22109-5.

[74] N. Ramsey and M. Fernández. *Specifying Representations of Machine Instructions.* ACM Transactions on Programming Languages and Systems, 19(3):492–524, 1997. ISSN 0164-0925.

[75] N. Ramsey and M. F. Fernandez. *The New Jersey Machine-Code Toolkit.* In USENIX Technical Conference, pages 289–302, 1995.

[76] N. Ramsey and D. R. Hanson. *A Retargetable Debugger.* Technical Report, Princeton University, Princeton, US-NJ, 1992.

[77] J. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture.* John Wiley, New York, US-NY, 1996. ISBN 0471149667.

[78] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages*, volume 1: Word, Language, Grammar. Springer-Verlag, New York, US-NY, 1997. ISBN 3540604200.

[79] A. Salomaa. *Formal Languages.* Academic Press, New York, US-NY, 1973. ISBN 0126157502.

[80] B. Schwarz, S. Debray, and G. Andrews. *Disassembly of Executable Code Revisited.* In 9th Working Conference on Reverse Engineering (WCRE'02), pages 45–54, Washington, US-DC, 2002. IEEE Computer Society. ISBN 0-7695-1799-4.

[81] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo. *On the Infeasibility of Modeling Polymorphic Shellcode.* In 14th ACM Conference on Computer and Communications Security (CCS'07), pages 541–551. ACM, 2007. ISBN 978-1-59593-703-2.

[82] S. Subramanian and J. V.Cook. *Automatic Verification of Object Code Against Source Code.* In Computer Assurance (COMPASS'96), pages 46–55. IEEE Computer Society, 1996. ISBN 0-7803-3390-X.

[83] G. Taha. *Counterattacking the Packers.* In Anti-Virus Asia Researchers Conference (AVAR'07), 2007.

[84] The LLVM Compiler Infrastructure. `http://llvm.org/`, 2014.

[85] H. Theiling. *Extracting Safe and Precise Control Flow from Binaries.* In 7th Conference On Real-Time Computing Systems and Applications (RTCSA'00), pages 23–30. IEEE Computer Society, 2000. ISBN 0-7695-0930-4.

[86] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*, 1995. `http://refspecs.freestandards.org/elf/elf.pdf`.

[87] K. Troshina, A. Chernov, and Y. Derevenets. *C Decompilation: Is It Possible?* In International Workshop on Program Understanding (IWPU'09), pages 18–27, 2009.

[88] K. Troshina, Y. Derevenets, and A. Chernov. *Reconstruction of Composite Types for Decompilation.* In 10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10), pages 179–188. IEEE Computer Society, 2010. ISBN 978-0-7695-4178-5.

[89] D. Ung and C. Cifuentes. *SRL - A Simple Retargetable Loader.* In The Australia Software Engineering Conference, pages 60–69. IEEE Computer Society, 1997. ISBN 0-8186-8081-4.

[90] M. Y. Vardi and P. Wolper. *An Automata-Theoretic Approach to Automatic Program Verification.* In 1st Logic in Computer Science (LICS'86), pages 332–344, Cambridge, US-MA, 1986. IEEE Computer Society. ISBN 0-8186-0720-3.

[91] L. Ďurfina and D. Kolář. *Generic Detection of the Statically Linked Code.* In 12th International Conference on Informatics (INFORMATICS'13), pages 157–161. Faculty of Electrical Engineering and Informatics, University of Technology Košice, 2013.

[92] L. Ďurfina and J. Křoustek. *Machine-Code Decompilation: Past, Present, and Future.* Technical Report, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2012.

[93] 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.

[94] 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.

[95] T. Vojnar and O. Lengál. *Formal Analysis and Verification — Abstract Interpretation.* Technical Report, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2010.

[96] H. S. Warren. *Hacker's Delight.* Addison-Wesley, Boston, US-MA, 2003. ISBN 978-02-0191-465-8.

[97] M. Weiser. *Program Slicing.* In 5th International Conference on Software Engineering (ICSE'81), pages 439–449. IEEE Computer Society, 1981. ISBN 0-89791-146-6.

[98] S. Wong, T. van As, and G. Brown. *ρ-VEX: A Reconfigurable and Extensible Softcore VLIW Processor.* In International Conference on Field-Programmable Technology (ICFPT'08), pages 369–372. IEEE Computer Society, 2008.

[99] P. Zemek. *Design of a Language for Unified Code Representation.* Technical Report, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2012.

[100] P. Zemek. *Selection of a Platform for Implementing a Retargetable Decompiler.* Technical Report, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2012.

[101] 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. ISBN 0-7695-2870-8.

Author's Publications

[102] J. Křoustek. *Code Analysis and Transformation To a High-Level Language.* In 15th Conference and Competition Student (EEICT'09), pages 196–198. Brno University of Technology, 2009. ISBN 978-80-214-3868-2.

[103] 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. ISBN 978-80-86840-47-5.

[104] J. Křoustek. *Decompilation of VLIW Executable Files — Caveats and Pitfalls.* In 3nd International Scientific Conference Theoretical and Applied Aspects of Cybernetics (TAAC'13), pages 287–296, Kyiv, UA, 2013. Kyiv: Bukrek. ISBN 978-966-399-538-0.

[105] J. Křoustek and D. Kolář. *Object-File-Format Description Language and Its Usage in Retargetable Decompilation.* In AIP Conference Proceedings (SCLIT'12), volume 1479, pages 466–469. American Institute of Physics (AIP), 2012. ISBN 978-0-7354-1091-6.

[106] J. Křoustek and D. Kolář. *Approaching Retargetable Static, Dynamic, and Hybrid Executable-Code Analysis.* Acta Informatica Pragensia (AIP), 2(1):18–29, 2013. ISSN 1805-4951.

[107] 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. ISSN 1820-0214.

[108] J. Křoustek and D. Kolář. *Preprocessing of Binary Executable Files Towards Retargetable Decompilation.* In 8th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13), pages 259–264, Nice, FR, 2013. International Academy, Research, and Industry Association (IARIA). ISBN 978-1-61208-283-7.

[109] 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. ISBN 978-1-61208-209-7.

[110] J. Křoustek, P. Matula, and L. Ďurfina. *Generic Plugin-Based Convertor of Executable File Formats and Its Usage in Retargetable Decompilation.* In 6th International Scientific and Technical Conference (CSIT'11), pages 127–130. Ministry of Education, Science, Youth and Sports of Ukraine, Lviv Polytechnic National University, Institute of Computer Science and Information Technologies, 2011. ISBN 978-966-2191-04-2.

[111] J. Křoustek and F. Pokorný. *Reconstruction of Instruction Idioms in a Retargetable Decompiler.* In 4th Workshop on Advances in Programming Languages (WAPL'13), pages 1507–1514, Krakow, PL, 2013. IEEE Computer Society. ISBN 978-1-4673-4471-5.

[112] 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. ISSN 1820-0214.

[113] 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. ISBN 978-1-4577-2209-7.

[114] J. Křoustek and S. Židek. *Generating Proper VLIW Assembler Code Using Scattered Context Grammars.* In 16th Conference and Competition Student (EEICT'10), volume 5, pages 181–185. Brno University of Technology, 2010. ISBN 978-80-214-4080-7.

[115] J. Křoustek, S. Židek, D. Kolář, and A. Meduna. *Exploitation of Scattered Context Grammars to Model VLIW Instruction Constraints.* In 12th Biennial Baltic Electronics Conference (BEC'10), pages 165–168. IEEE Computer Society, 2010. ISBN 978-1-4244-7357-1.

[116] 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. ISSN 0976-5697.

[117] Z. Přikryl, J. Křoustek, T. Hruška, and D. Kolář. *Fast Translated Simulation of ASIPs.* In 6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'10), pages 135–142. Masaryk University, 2010. ISBN 978-80-87342-10-7.

[118] 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. ISBN 978-1-4244-9753-9.

[119] Z. Přikryl, J. Křoustek, T. Hruška, and D. Kolář. *Fast Translated Simulation of ASIPs.* OpenAccess Series in Informatics (OASIcs), 16(1):93–100, 2011. ISSN 2190-6807.

[120] Z. Přikryl, J. Křoustek, T. Hruška, D. Kolář, K. Masařík, and A. Husár. *Design and Debugging of Parallel Architectures Using the ISAC Language.* In Annual International Conference on Advanced Distributed and Parallel Computing and Real-Time and Embedded Systems (RTES'10), pages 213–221. Global Science and Technology Forum (GTSF), 2010. ISBN 978-981-08-7656-2.

[121] 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. ISSN 2010-2283.

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

[123] 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. ISSN 1942-2628.

[124] 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. ISSN 2160-2174.

[125] 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. ISBN 978-90-77381-71-7.

[126] L. Ďurfina, J. Křoustek, and P. Zemek. *Psyb0t Malware: A Step-by-Step Decompilation Case Study.* In 20th Working Conference on Reverse Engineering (WCRE'13), pages 449–456, Koblenz, DE, 2013. IEEE Computer Society. ISBN 978-1-4799-2930-6.

[127] L. Ďurfina, J. Křoustek, and P. Zemek. *Retargetable Machine-Code Decompilation in Your Web Browser.* In 3rd IEEE World Congress on Information and Communication Technologies (WICT'13), pages 57–62, Hanoi, VN, 2013. IEEE Computer Society. ISBN 978-1-4799-3230-6.

[128] L. Ďurfina, J. Křoustek, P. Zemek, and B. Kábele. *Accurate Recovery of Functions in a Retargetable Decompiler.* In 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'12), LNCS 7462, pages 390–392, Berlin, Heidelberg, DE, 2012. Springer-Verlag. ISBN 978-3-642-33337-8.

[129] 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. ISBN 978-0-7695-4891-3.

[130] L. Ďurfina, J. Křoustek, P. Zemek, B. Kábele, and D. Kolář. *On Complex Reconstruction of Functions from Binary Executable Files.* In 8th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'12), pages 100–101. Masaryk University, 2012. ISBN 978-80-87342-15-2.

[131] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, T. Hruška, K. Masařík, and A. Meduna. *Advanced Static Analysis for Decompilation Using Scattered Context Grammars.* In Applied Computing Conference (ACC'11), pages 164–169. World Scientific and Engineering Academy and Society (WSEAS), 2011. ISBN 978-1-61804-051-0.

[132] 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.* In 7th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science (MEMICS'11), pages 114–114. Masaryk University, 2011. ISBN 978-80-214-4305-1.

[133] 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.* In 5th International Conference on Information Security and Assurance (ISA'11),

volume 200 of Communications in Computer and Information Science, pages 72–86, Berlin, Heidelberg, DE, 2011. Springer-Verlag. ISBN 978-3-642-23140-7.

[134] 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. ISSN 1738-9976.

[135] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, K. Masařík, T. Hruška, and A. Meduna. *Design of an Automatically Generated Retargetable Decompiler.* In 2nd European Conference of Computer Science (ECCS'11), pages 199–204. North Atlantic University Union, 2011. ISBN 978-1-61804-056-5.

List of Abbreviations

Compiler Construction

(O)EP	(**O**riginal) **E**ntry **P**oint
(Win)PE	Microsoft **P**ortable **E**xecutable and Common Object File Format
ABI	**A**pplication **B**inary **I**nterface
AST	**A**bstract **S**yntax **T**ree
BB	**B**asic **B**lock
CFA	**C**ontrol-**F**low **A**nalysis
CFG	**C**ontrol-**F**low **G**raph
CG	**C**all **G**raph
COFF	**C**ommon **O**bject **F**ile **F**ormat
DFA	**D**ata-**F**low **A**nalysis
DIE	**D**ebugging **I**nformation **E**ntry
DLL	**D**ynamically-**L**inked **L**ibrary
EFF	**E**xecutable-**F**ile **F**ormat
ELF	**E**xecutable and **L**inking **F**ormat
GCC	**G**NU **C**ompiler **C**ollection
HLL	**H**igh-**L**evel **L**anguage
IR	**I**ntermediate **R**epresentation
JIT	**J**ust-**i**n-**T**ime
LLVM	Compiler Infrastructure, formerly **L**ow **L**evel **V**irtual **M**achine
PDB	**P**rogram **D**ata**b**ase
RTL	**R**egister **T**ransfer **L**evel
SMC	**S**elf-**M**odifying **C**ode

SSA	**S**tatic **S**ingle **A**ssignment

Theory of Formal Languages

(P)SC	(**P**ropagating) **S**cattered **C**ontext
(P)SCG	(**P**ropagating) **S**cattered **C**ontext **G**rammar
(P)SCGP	(**P**ropagating) **S**cattered **C**ontext **G**rammar with **P**riority
CF	**C**ontext **F**ree
CS	**C**ontext **S**ensitive
RE	**R**ecursively **E**numerable

Processor Architectures

ADL	**A**rchitecture **D**escription **L**anguage
ARM	**A**dvanced **R**ISC **M**achines
ASIP	**A**pplication **S**pecific **I**nstruction-Set **P**rocessor
CISC	**C**omplex **I**nstruction **S**et **C**omputer
FPU	**F**loating **P**oint **U**nit
ILP	**I**nstruction **L**evel **P**arallelism
ISA	**I**nstruction **S**et **A**rchitecture
ISAC	**I**nstruction **S**et **A**rchitecture **C**
MIPS	**M**icroprocessor without **I**nterlocked **P**ipeline **S**tages
MPSoC	**M**ulti**p**rocessor **S**ystem-**o**n-**C**hip
NOP	**N**o **Op**eration
PC	**P**rogram **C**ounter
PPC	**P**ower**PC**
RISC	**R**educed **I**nstruction **S**et **C**omputer
VLIW	**V**ery **L**ong **I**nstruction **W**ord
VM	**V**irutal **M**achine
x86	32-bit Intel Architecture, also known as IA-32
x86-64	64-bit version of x86 microprocessor family, also known as AMD64

Appendix A

Decompilation Results

"*The last good thing written in C was Franz Schubert's Symphony No. 9*"

Erwin Dieterich

In the last part of this thesis, we present several illustrations of HLL code obtained from the proposed retargetable decompiler (v1.8 released in September 2014). Firstly, in Section A.1, we give a comparison of several simple C programs and their decompiled counterparts. These examples are not intended to be exhaustive and too complex because this will be out of the range of this thesis. The reader is advised to try decompilation of the more complex applications by using the aforementioned web-based decompilation service available at `http://decompiler.fit.vutbr.cz/decompilation/`.

Furthermore, in Section A.2, we present a case-study of malware decompilation, which is a shortened version of our previous paper [126]. As we will see, the retargetable decompiler is fully capable to decompile such real-world applications.

A.1 Decompilation of Simple Programs

In this section, we give five simple decompilation examples illustrating accuracy of our tool. Each test is written in the C language and each of them differs in test configuration:

- *Application* – Ackermann function (Figure A.1), factorial function (Figure A.2), Fibonacci function (Figure A.3), greatest-common-divisor function (Figure A.4), and instruction-idioms test (Figure A.5);
- *Target architecture* – MIPS, PIC32, PowerPC, ARM, and Intel x86;
- *EFF* – ELF and WinPE;
- *Used compiler* – GCC (the same versions as listed in Chapter 7) and LLVM Clang (v3.5);
- *Compiler optimizations* – `O0`, `O1`, `O2`, `O3`, and `Os` (optimize for size). The last three levels may use function inlining;
- *Additional information* – debugging information, symbolic information, or stripped file.

```
#include <stdio.h>
#include <stdlib.h>

unsigned ack(unsigned m, unsigned n) {
  if (m == 0)
    return n + 1;
  else if (n == 0)
    return ack(m - 1, 1);
  else
    return ack(m - 1, ack(m, n - 1));
}

int main(int argc,char *argv[]) {
  unsigned int m, n, result;
  m = rand();
  n = rand();
  result = ack(m, n);
  printf("Ackermann(%d, %d) = %d\n", m, n, result);
  return result;
}
```

(a) Original code.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t ack(int32_t m, int32_t n) {
  if (m == 0)
    return n + 1;
  int32_t v1 = m - 1;
  int32_t result;
  if (n == 0)
    result = ack(v1, 1);
  else
    result = ack(v1, ack(m, n - 1));
  return result;
}

int main(int argc, char ** argv) {
  int32_t m = rand();
  int32_t n = rand();
  int32_t result = 0;
  int32_t v1 = ack(m, n);
  result = v1;
  printf("Ackermann(%d, %d) = %d\n", m, n, v1);
  return result;
}
```

(b) Decompiled code.

Figure A.1: Decompilation of the Ackermann function (PowerPC, ELF, GCC, `-O0`, DWARF).

```c
#include <stdio.h>

int factorial(int input) {
  int x, fact = 1;
  for (x = input; x > 1; x--)
    fact *= x;
  return fact;
}

int main(int argc, char *argv[]) {
  int num;
  scanf("%d", &num);
  int res = factorial(num);
  printf("factorial(%d) = %d\n", num, res);
  return res;
}
```

(a) Original code.

```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t function_9d002874(int32_t a1) {
  if (a1 < 2) return 1;
  int32_t v1 = a1 - 1;
  int32_t result = v1 * a1;
  while (v1 != 1) {
    v1 = v1 - 1;
    result *= v1;
  }
  return result;
}

int main(int argc, char ** argv) {
  int32_t v1;
  scanf("%d", &v1);
  int32_t result = function_9d002874(v1);
  printf("factorial(%d) = %d\n", v1, result);
  return result;
}
```

(b) Decompiled code.

Figure A.2: Decompilation of the factorial function (PIC32, ELF, GCC, `-O1`, stripped).

```
#include <stdio.h>
#include <stdlib.h>

int fib(int n) {
  if (n == 1) return 0;
  if (n == 2) return 1;
  return fib(n - 1) + fib(n - 2);
}

int main(int argc, char *argv[]) {
  int x = rand();
  int res = fib(x);
  printf("fib(%d) = %d\n", x, res);
  return res;
}
```

(a) Original code.

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

int32_t fib(int32_t a1) {
  if (a1 == 1)
    return 0;
  int32_t result;
  if (a1 != 2) {
    result = fib(a1 - 1);
    result += fib(a1 - 2);
  } else {
    result = 1;
  }
  return result;
}

int main(int argc, char **argv) {
  int32_t v1 = rand();
  int32_t result = fib(v1);
  printf("fib(%d) = %d\n", v1, result);
  return result;
}
```

(b) Decompiled code.

Figure A.3: Decompilation of the Fibonacci function (MIPS, ELF, Clang, `-O2`, symbols).

```c
#include <stdio.h>

int gcd(int a, int b) {
  int remainder;
  while (b != 0) {
    remainder = a % b;
    a = b;
    b = remainder;
  }
  return a;
}

int main(int argc,char *argv[]) {
  int a, b;
  scanf("%d %d", &a, &b);
  int result = gcd(a, b);
  printf("GCD(%d, %d) = %d\n", a, b, result);
  return result;
}
```

(a) Original code.

```c
#include <stdint.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  int32_t v1, v2, result;
  scanf("%d %d", &v1, &v2);
  if (v2 == 0) {
    result = v1;
    printf("GCD(%d, %d) = %d\n", v1, v2, result);
    return result;
  }
  int32_t v3 = v2;
  int32_t v4 = v1 % v3;
  result = v3;
  while (v4 != 0) {
    int32_t v5 = v3;
    v3 = v4;
    v4 = v5 % v3;
    result = v3;
  }
  printf("GCD(%d, %d) = %d\n", v1, v2, result);
  return result;
}
```

(b) Decompiled code.

Figure A.4: Decompilation of the greatest-common-divisor function (ARM, ELF, Clang, `-Os`, inlining, stripped).

```
#include <stdio.h>

int main(int argc, char *argv[]) {
  int a;
  scanf("%d", &a);
  printf("1. Multiply: %d\n", a * 4);
  printf("2. Divide: %d\n", a / 8);
  printf("3. >= 0 idiom: %d\n", a >= 0);
  printf("4. Magic sign-div: %d\n", a / 10);
  printf("5. XOR by -1: %d\n", -a - 1);
  return a;
}
```

(a) Original code.

```
#include <stdint.h>
#include <stdio.h>

int main(int argc, char ** argv) {
  int32_t result;
  scanf("%d", &result);
  printf("1. Multiply: %d\n", 4 * result);
  printf("2. Divide: %d\n", result / 8);
  printf("3. >= 0 idiom: %d\n", result > -1);
  printf("4. Magic sign-div: %d\n", result / 10);
  printf("5. XOR by -1: %d\n", -1 - result);
  return result;
}
```

(b) Decompiled code.

Figure A.5: Decompilation of the instruction idioms (Intel x86, WinPE, `-O3`, stripped).

A.2 Decompilation of a Real-World Malware

This section gives a case study of analysing a computer malware by the previously described retargetable decompiler. The target of our examination is a computer worm called *psyb0t*, which attacks network infrastructure devices (e.g. modems and routers) running MIPS processors with Linux-based operating systems and creates a botnet operated by IRC (Internet Relay Chat) command-and-control (C&C) servers. It received quite an attention during its discovery [17]. The following text describes all the major decompilation phases with illustrations.

A.2.1 Preprocessing Phase

The size of the examined binary file is 29,264 bytes and its MD5 hash is `58f00c14942cae1e9f24b03d55cd295d`. This is the latest known version of this malware[116]. As will be discussed later, it marks itself as "`PSYBOT v2.9L`". The previous mentioned articles about psyb0t

[116]It should be noted that psyb0t has successors, like the Chuck Norris botnet – `http://www.muni.cz/ics/research/cyber/chuck_norris_botnet`.

analysis were focused mainly on the older version 2.5L[117].

Our decompiler performs the initial analysis via the file-information-gathering application `fileinfo` described in Subsection 6.3.1. The very first step of this initial analysis is detection of EFF and the target platform. The file starts with an identifier "'0x7f'ELF". In other words, it is the ELF file format, see Section 3.2. Our tool is also able to retrieve other valuable information, such as target architecture (32-bit MIPS), endianity (little), EP address (0x106828), etc.

However, the EP address (i.e. address of the first instruction executed during the application run-time) is not typical for MIPS compilers because it is usually placed somewhere nearby 0x08000000; the section and symbol tables are empty, which is also unusual but correct. The information about the originally used compiler is usually stored in the optional `.comment` section, but this file lacks such a section. Moreover, the file content is also atypical because there are no visible strings, such as symbol names, section names, or strings for user interaction during run-time.

Based on these clues, we can guess that the file is packed and maybe obfuscated by some packer or protector. In the classical approach, presented in [17], it is necessary to distinguish the used packer manually, unpack it, and analyse it by using a MIPS disassembler. Luckily, our retargetable decompiler can handle such situation automatically.

The sequence of the entry-point instructions retrieved by `fileinfo` for the psyb0t malware consists of "e00011040000f7272028a4000000e6ac00800d3c"; its translation to the MIPS instructions is illustrated in Figure A.6.

```
; address      hex dump     MIPS instruction
;------------------------------------------
  00106828     041100e0     bal 0x00006bac
  0010682c     27f70000     addiu s7,ra,0
  00106830     00a42820     add a1,a1,a0
  00106834     ace60000     sw a2,0(a3)
  00106838     3c0d8000     lui t5,0x8000
```

Figure A.6: Psyb0t entry-point instructions.

This sequence is matched with the internal signature for the MIPSel/ELF UPX packer[118] of the shortened little-endian form "----11040000f7272028a4000000e6ac". Therefore, we figured out that the UPX packer for the MIPS architecture was used for application packing. The used version of UPX was 3.03 and this was the up-to-date version when the malware started spreading. In normal circumstances, we are able to unpack such a file by using our internal plugin-based unpacker, see Subsection 6.3.2 for details.

However, the psyb0t's author wiped out (i.e. replaced by zero bytes) several UPX control sequences. The first (file offset 0x0078), second (0x6803, near the entry point), and third (0x722c) part consists of the string "UPX!" and are mandatory for UPX detection. The fourth part laying at file offset (0x6848) is not necessary for the detection and it originally contained the following string:

```
$Info: This file is packed with the UPX executable packer
http://upx.sf.net $ $Id: UPX 3.03
Copyright (C) 1996-2008 the UPX Team. All Rights Reserved. $
```

[117] http://www.baume.id.au/psyb0t/PSYBOT.pdf?info=EXLINK
[118] http://upx.sourceforge.net/

Because of these modifications, our UPX-unpacking plugin was unable to successfully unpack it. Furthermore, the generic unpacker, mentioned in conclusion, is unavailable yet. Therefore, we had to manually patch the three missing "`UPX!`" strings before unpacking by the plugin. The unpacked file size is 127,892 bytes that gives us a 22.88% compression ratio. The unpacked file contains 20 sections, 133 symbols, and several hundred strings.

The last part of the preprocessing phase is a conversion of the unpacked ELF file into an internal COFF based format. This is done by using our `bintran` application mentioned in Subsection 6.3.3.

A.2.2 Front-End Phase

Next, the unpacked psyb0t application in the COFF format is processed in the front-end phase. Firstly, the instruction decoder has to be automatically generated based on the MIPS architecture model in the ISAC language. The model is relatively simple—about 4,800 lines in this ADL, see Table 7.1. After that, the instruction decoder translates the MIPS machine-code instructions stored in COFF into LLVM IR platform-independent representation that is further processed by the following analyses, see Subsection 6.4.3.

In the front-end phase, various analyses are applied, but in what follows, we focus only on those related to our subject. This means that we exclude, for example, a description of analysis that reads DWARF debugging information from the executable because psyb0t does not contain any DWARF data.

Firstly, we process the whole executable to reveal data as strings. This is important for later usage of these strings in function calls. It is implemented by analysing data sections. The analysis tries to find a sequence of printable characters terminated by the zero byte. Such a sequence is marked as a string and its address is stored. If we detect an access to this address, we know that it uses a specific string and we have the value of that string.

The executable contains also symbols for functions. As we will see later, it does not have the symbols for all functions, but we can use the available symbols to improve the decompilation results, see Subsection 6.4.1. This analysis is simple and just stores the pairs with the address and name of each symbol, see Table A.1 for illustration. Since there is a symbol for the `main` function, we can skip the entry-point analysis. If the executable was without that symbol (i.e. stripped), this analysis would try to find the address of `main` by using its internal compiler-specific database or by using a heuristic detection.

Afterwards, we create a CFG based on the method described in Subsection 6.4.4. All branch instructions are examined in order to obtain their target addresses and resolve the type of branch. The goal is to recognize conditional and unconditional branches, function calls, and returns from a function. A challenge hidden in this executable file is the usage of *position independent code* (PIC). This means that functions are called by indirect branches.

On the MIPS platform, the indirect branch is of the form `jalr t9`. Therefore, if we want to know the called function, we have to track the value that is stored in register `t9`. This is the same problem as described in Figure 6.24 and we also solve it by using our static code interpreter that was described in Subsection 6.4.5.

After CFA, we are able to detect functions based on algorithms presented in Subsection 6.4.6. As we mentioned before, the decompiled executable has symbols, but this analysis is run nevertheless because the set of symbols can be incomplete. This is also that case. The number of available function symbols is 34, but the overall number of detected functions is 91. This can be caused by linked code from libraries without symbols or by special compiler routines.

Table A.1: Shortened list of functions extracted from symbols.

function address	**function name**
0x404c20	main
0x402da0	cgen
0x40450c	ddos
0x406f44	IrcPrivmsg
0x40eb14	rsgen
0x4156ac	rscan
0x4162cc	backup
0x41646c	spoof
0x416b98	kill_all
0x416f08	is_running
0x4173d8	BurnData
0x417a94	Daemonize
0x4183a0	getip
0x419260	stats_refresh

After all the remaining front-end analyses, we generate LLVM IR code, which is processed by the middle-end phase.

A.2.3 Middle-End Phase

In this stage, we have a very low-level LLVM IR of the input binary. Each basic block represents a single assembly instruction, and there may be many redundant instructions (recall that each assembly instruction is decompiled in isolation). The key role of the middle-end part of our decompiler is to optimize the input LLVM IR code and prepare it for the back-end.

For example, consider the block in Figure A.7, which was generated by the front-end for the instruction `jalr t9` on address `0x41a088`.

```
%u0_41a088 = add i32 4300940, 0
%_c_41a088 = add i32 4, 0
%u1_41a088 = add i32 %u0_41a088,  %_c_41a088
%_e_41a088 = add i32 31, 0
%u2_41a088 = load i32* @gpregs25
%u0_ds_41a088 = add i16 119, 0
%u1_ds_41a088 = sext i16 %u0_ds_41a088 to i32
store i32 %u1_ds_41a088, i32* @gpregs24
%arg1049_41a088 = load i32*  @gpregs4
%r_41a088 = call i32 @usleep(i32 %arg1049_41a088)
store i32 %r_41a088, i32*  @gpregs2
```

Figure A.7: A block generated by the front-end for the instruction `jalr t9` on address `0x41a088`.

As described in Subsection 6.4.5, `jalr` is an indirect branch to an address stored in a register, and a store of the return address in another register. By using our interpreter and the import table from the executable file, we were able to detect that the branch is actually a call to the function `usleep` from `<unistd.h>`. However, due to generality, a lot of boilerplate code has to be emitted along with the call, which is optimized in the middle-end. The block from Figure A.7 after optimizations can be seen in Figure A.8.

```
%res0_41a088 = tail call i32 @usleep(i32 %arg1)
store i32 %res0_41a088, i32* @gpregs2, align 4
```

Figure A.8: The block from Figure A.7 after optimizations.

The `arg1` variable is actually the name of the function argument, in which this call to `usleep` appears. It should be noted that the optimizer in the middle-end takes into account also the surrounding blocks so it performs the optimization globally, not just locally over a single block.

Furthermore, the idiom analysis of the middle-end part, successfully detected and transformed all the commonly used idioms that appeared within this executable file (e.g. bit shift multiplication/division, multiplication by an invariant, modulo by power of two).

A.2.4 Back-End Phase

Finally, the back-end part takes the middle-end optimized LLVM IR, and it produces the resulting HLL code by using the methods mentioned in Section 6.6. Because this phase is out of the scope of this thesis, we omit the details that can be found in [126].

A.2.5 Analysis of the Obtained Results

Psyb0t is an IRC bot, which reads the topic of the IRC channel after connecting to the server and gets commands from this topic. It scans devices in the network and tries to log in by default usernames and passwords or uses an exploit when the login fails. Once a shell of the vulnerable device is acquired, psyb0t downloads itself from a remote server by using the `wget` application into the victim's location `/var/tmp/udhcpc.env`. This new instance of psyb0t is executed afterwards. It supports classical malware actions like DDoS attacks, brute-force attacks on router passwords, download of files, visitation of web pages, or executing shell commands.

There are two known versions of psyb0t. We have decompiled the newer one, which identifies itself as `[PRIVATE] PSYBOT v2.9L`. This version is better secured against unpacking by UPX and it affects more network devices, mainly models by Linksys, Netgear, and other routers running DD-WRT or OpenWrt firmware. The application is written in the C language. This can be spotted by the names of called functions and also by the usage of position independent code, which can be simply turned on by flag `fPIC` of the `GCC` compiler.

In this subsection, we introduce a brief description of psyb0t's behavior by using snippets of code from the decompiler in order to show how the decompiler is useful for faster analysis of malware.

In the previous subsections, we have presented the whole decompilation process in a step-by-step way, and we have shown the code from our own decompiler. Now, we can analyse the obtained HLL source code. We describe the behavior of psyb0t immediately after its execution, i.e. the code starting at the entry-point—the `main` function.

Firstly, we can take a look on the call graph generated by our decompiler. It is good for a fast detection of relations between functions. A part of that graph is shown in Figure A.10. A complete call graph is omitted due to space constraints. The most important parts of the main function are listed in Figure A.9. The comments were added manually. Selected parts are listed separately with describing notes.

```
int main(int argc, char **argv) {
  //...
  uint32_t *file = fopen("/var/tmp/udhcpd.mtx","w");
  //...
  uint32_t fd = fileno((uint32_t *)file);
  //...
  uint32_t err_code = flock(fd, LOCK_EX | LOCK_NB);
  //...
  RSeed();
  //...
  Daemonize();
  //...
  system("/etc/firewall_start");
  system("iptables -A INPUT -p tcp --dport 23 -j DROP");
  system("rm -f /var/tmp/udhcpc.env");
  //...
  backup();           // Backup file /var/tmp/hosts
  //...
  function_404b1c(); // Prepare IRC nickname
  //...
  function_4056cc(); // Await for commands
  //...
  fclose(fd);         // Remove mutex file and quit
  //...
}
```

Figure A.9: Psyb0t: simplified code of the decompiled main function.

The first operation in main is opening of a file named udhcpd.mtx in a temporary folder. It is opened in the writing mode. The author of psyb0t followed good practice and checked the result of the operation.

```
uint32_t *file = fopen("/var/tmp/udhcpd.mtx", "w");
var3 = (uint32_t)file;
if (file == NULL) {
    return 1;
}
```

Subsequently, there is the obtained file descriptor, which is checked for validity. If it is valid, the application tries to lock the file. After this operation, we can better understand the suffix .mtx in the name of the file, because it serves as a mutex. The lock is exclusive and it is does not block when the locking is done. The mutex is acquired only if there is no other running instance of psyb0t. Otherwise, the application is terminated.

```
uint32_t fd = fileno(file);
if (fd == -1) {
    var3 = 1;
    return 1;
```

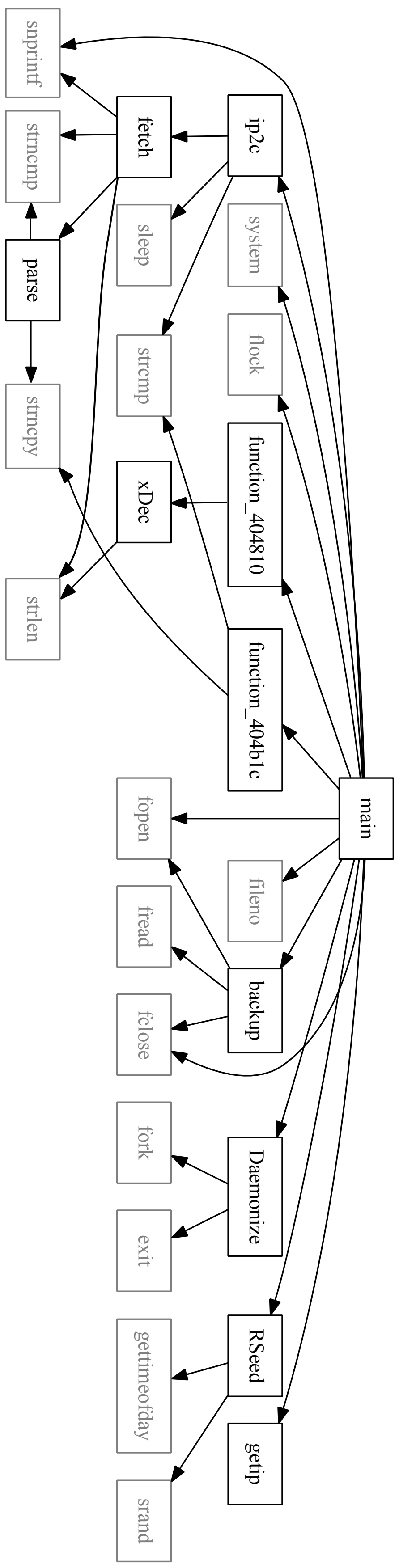


Figure A.10: A part of the call graph for `main`. Nodes in black are user-defined functions while grey nodes denote external functions.

```
}
var9 = 6;
uint32_t err_code = flock(fd, LOCK_EX | LOCK_NB);
```

In all the three previous calls of linked functions, the back-end applies renaming of variables storing the returned values. For `fopen`, it uses the common name `file`. For `fileno`, it uses `fd` as a file descriptor, and finally, for `flock`, it uses `err_code`. We can take a closer look on the call of `flock`. The original second argument is `6`, but the back-end is able to find out the names of the symbolic constants that form this value.

If the lock is acquired, the application calls internal function `RSeed`, which initializes the pseudo-random generator of numbers by calling `srand`. An important call is that of function `Daemonize`, where the application is forked and the parent process is terminated. The child process continues in its execution on background with starting and setting a firewall, and removing itself from the file system. The second call of `system` updates firewall rules to drop all the packets on `tcp` port `23` (i.e. disable inbound telnet communication). The third command removes the file that psyb0t uses for spreading, probably to cover its tracks. After removal, psyb0t is located only in memory and a reset of the infected device will disinfect it. The executed shell commands are of the following form:

```
/etc/firewall_start
iptables -A INPUT -p tcp --dport 23 -j DROP
rm -f /var/tmp/udhcpc.env
```

Afterwards, the memory-located psyb0t backups the file `/var/tmp/hosts` inside the `backup` function and reports itself to the C&C IRC channel naming itself as a regular expression `\[NIP\]-[A-Z0-9]{9}` (inside `function_404b1c`). The last nine symbols are generated randomly as an index to string `"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"` using the previously initialised pseudo-random generator.

In Subsection 6.4.8, we have briefly mentioned the data section analysis. It provides us an array with strings that are the names of commands which are accepted by psyb0t. These commands are received from the topic of the connected IRC channel.

```
const char *STRINGS[] = {
  "mode", "login", "logout", "_exit_", "sh", "tlist", "kill",
  "killall", "silent", "getip", "visit", "scan", "rscan", "sel",
  "sleep", "esel", "rejoin", "upgrade", "wupgrade", "ver",
  "wget", "lscan", "rlscan", "getinfo", "rsgen", "vsel", "split",
  "gsel", "sflood", "uflood", "iflood", "pscan", "fscan", "r00t",
  "sql", "pma", "socks", "rsloop", "report", "uptime", "usel",
  "spoof", "viri", "smb", "cgen"
};
```

Some of these strings are the same as names of the reconstructed functions and we can presume that such functions implement these commands.

One of the commands is `fetch` and we have a function with the same name. If we take a look at it, we can see the following code:

```
snprintf((uint8_t *)&var_9, 5120,
  "GET /servlet/view/banner/javascript/zone?zid=81&
  pid=0&random=%d&millis=%lu HTTP/1.1\r\nHost: %s\r
  \n%s%sReferer: %s\r\n\r\n",
  var_18, var_20, (uint8_t *)&var_12, (uint8_t *)
  &var_13, (uint8_t *)&var_16, (uint8_t *)&var_17);
len = strlen((uint8_t *)&var_9);
```

```
dpage(-23184, (uint16_t)var_9, 0, 1);
```

There is a preparation of an `HTTP` command that is used in internal function `dpage` that uses standard functions `socket`, `connect`, `send`, and `recv` for network communication. Psyb0t uses a timeout by registering a function for handling `SIGALRM`. Before `connect`, there is a call `alarm(3)` to wait at most three seconds for connection, and before `recv`, there is `alarm(12)`.

During its run-time, psyb0t loops in `function_4056cc` awaiting for other commands obtained either from IRC channel topic or through a private massage. Commands `scan`, `rscan`, `lscan`, `rlscan`, `pscan`, and `fscan` tell psyb0t to scan for other vulnerable devices and try to spread itself to them (as described in the beginning of this subsection).

Finally, in Table A.2, we provide some statistics about the output from the decompiler. The result in the Python-like language is shorter because it does not use types. The size of both files is quite large, which is caused by the generation of many assignments, where some of them are not needed. In the future, we plan to improve our optimization algorithms in the back-end part to remove more such code and produce even more readable output.

Table A.2: Statistics about decompiler output for psyb0t.

feature	**value**
Internal functions count	91
External functions count	57
Function calls	1278
C output size	553 kB
Python-like output size	453 kB