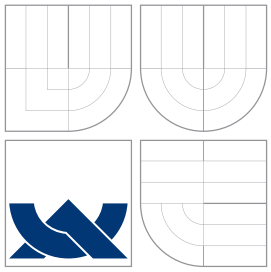
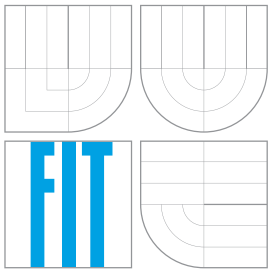


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GENERICKÝ ZPĚTNÝ PŘEKLAD ZA ÚČELEM ROZPOZNÁNÍ CHOVÁNÍ

GENERIC REVERSE COMPILATION TO RECOGNIZE SPECIFIC BEHAVIOR

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE — Ing. LUKÁŠ ĎURFINA
AUTHOR

VEDOUCÍ PRÁCE — Doc. Dr. Ing. DUŠAN KOLÁŘ
SUPERVISOR

BRNO 2014

Abstrakt

Práce je zaměřena na rozpoznávání specifického chování pomocí generického zpětného překladu. Generický zpětný překlad je proces, který transformuje spustitelné soubory z různých architektur a formátů objektových souborů na stejný jazyk na vysoké úrovni. Tento proces se vztahuje k nástroji Lissom Decompiler. Pro účely rozpoznání chování práce zavádí Language for Decompilation – LfD. LfD představuje jednoduchý imperativní jazyk, který je vhodný pro srovnávaní. Konkrétní chování je dáno známým spustitelným souborem (např. malware) a rozpoznání se provádí jako najití poměru podobnosti s jiným neznámým spustitelným souborem. Tento poměr podobnosti je vypočítán nástrojem LfDComparator, který zpracovává dva vstupy v LfD a rozhoduje o jejich podobnosti.

Abstract

Thesis is aimed on recognition of specific behavior by generic reverse compilation. The generic reverse compilation is a process that transforms executables from different architectures and object file formats to same high level language. This process is covered by a tool Lissom Decompiler. For purpose of behavior recognition the thesis introduces Language for Decompilation – LfD. LfD represents a simple imperative language, which is suitable for a comparison. The specific behavior is given by the known executable (e.g. malware) and the recognition is performed as finding the ratio of similarity with other unknown executable. This ratio of similarity is calculated by a tool LfDComparator, which processes two sources in LfD to decide their similarity.

Klíčová slova

zpětný překlad, dekompilace, obfuskace, malware, chování programu, podobnost

Keywords

reverse compilation, decompilation, obfuscation, malware, program behavior, similarity

Citace

Lukáš Ďurfina: Generic Reverse Compilation to Recognize Specific Behavior, disertační práce, Brno, FIT VUT v Brně, 2014

Generic Reverse Compilation to Recognize Specific Behavior

Prohlášení

Prehlasujem,že dizertačnú prácu som vypracoval samostantne pod vedením Doc. Dr. Ing. Dušana Koláře a uviedol som všetky literárne zdroje, z ktorých som čerpal.

. . . . . .
Lukáš Ďurfina
December 10, 2014

Poděkování

Ďakujem môjmu školiteľovi Doc. Dr. Ing. Dušanovi Kolářovi a kolegom z projektu Lissom za ich pripomienky, spoluprácu a odborné rady. Taktiež ďakujem mojej rodine a priateľom, ktorí ma podporovali počas štúdia.

Contents

Chapter 1

Introduction

Reverse compilation is a process which has been researched for many years. Due to new architectures and new compilers for these architectures it is very evolving and difficult process. A wide-known name for reverse compilation is also decompilation. The main aim of decompilation is a gain of high level source code, which was used to create an executable. So it is a reverse process to the compilation.

The motivation for getting original source code from executable can be various: debugging, theft of intellectual property, or analysis of its behavior. This thesis is aimed on the last point and its goal is to recognize specific behavior of different executables, mainly malware. Thesis summarizes the various types of obfuscations, which are used to protect malware against detection by antivirus system. Obfuscations change the binary code, but the behavior remains the same.

Generic decompiler has to be able to process executables of different object file formats from different architectures. We created the decompiler that successfully completes this task. Other task of decompiler is suppressing of differences caused by various obfuscations. The ability to produce the source code without damages by obfuscations is important for finding the similarities between source codes obtained by decompilation from related executables. Therefore, the design of decompiler is adapted to more complicated conditions, which are given by malware environment.

The decompiler transforms executable to high level language (C, Python', or LfD). LfD is a simple imperative language, which is designed for the purpose of this thesis. By getting the output in this language, we are able to compare behavior of executables, which are originally from different architectures and file formats or obfuscated. This part is completed by the tool LfDComparator. One of possible usage is detection of malware for the architecture as ARM, MIPS, or PowerPC, if we have recognized its binary for x86.

The organization of this thesis is following: Chapter 2 defines the most important keywords for this thesis. The following Chapter 3 presents new trend in malware, called Internet of things. It also introduces various types of obfuscations. The history of reverse engineering is summarized in Chapter 4. Except the history, the currently available and well-known decompilers are listed and compared. The generic decompiler is described in Chapter 5. It covers the overall design, front-end, middle-end, back-end, and real experience with malware analysis on the output source code from the decompiler. The next Chapter 6 studies the opportunities for the comparing of the output source codes and recognizing the specific behavior. Experimental results are shown in Chapter 7. There are the examples of C output from decompiler and the results of specific behavior recognition. The last Chapter 8 concludes the thesis.

Chapter 2

Definitions

These chapter defines the most important keywords of this thesis. We assume that the reader is familiar with the basics of formal language theory and the theory of compiler design (see [4]).

- *Alphabet* [46]. An alphabet is a finite, nonempty set of elements, which are called *symbols*.
- *Language* [46]. Let Σ be an alphabet and $L \subseteq \Sigma*$. Then, L is a language over Σ.
- *Context free grammar* [46] is a quadruple (N, T, P, S), where N is a finite set of nonterminals, T is a finite set of terminals, $S \in N$ is the starting nonterminal, and P is a finite set of rules of the form $A \to w$, where $A \in N$ and $w \in (N \cup T)^*$.
- *Basic block* [15] is a portion of the binary code that has one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program, and it has one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block. Under these circumstances, whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once.
- *Control flow graph* [78] is a single-root, connected and directed graph for describing control flow information of a program. It is represented by a triple (N, E, h), where N is the set of basic blocks, E is the set of directed edges between these basic blocks, and h is the entry point of the program and $h \in N$.
- *Call graph* [12]: Let $P = \{p_1, p_2, ...\}$ be the finite set of procedures of a program. A call graph is a triple (N, E, h), where N is the set of procedures and $n_i \in N$ represents one and only one $p_i \in P$, E is the set of edges and $(n_i, n_j) \in E$ represents one or more references of p_i to p_j, and h is the `main` procedure and $h \in N$.
- *Memory place* is an addressable place that can be written or read. It could be register, place on the stack, or place in the memory. The memory place is *defined* if its content is modified (it is assigned a new value). The memory place is *used* if it is referenced (its value is used).
- *Definition-use chain* [12] for a definition d at instruction i is the set of instructions j, where d could be used before being redefined.
- *Use-definition chain* [12] for a use u at instruction j is the set of instructions i, where u was defined.

- *Obfuscation* [49] is a process when some instructions of the original code are replaced by program fragments that are semantically equivalent but more difficult to analyze, or additional instructions are added to the program and they do not change its behavior.

- *Dead code* [18] refers to computations whose results are never used. The notion of results not used must be considered broadly. For example, if it is possible for a computation to generate exceptions or raise signals whose handling can affect the behavior of the rest of the program, then we cannot consider that computation to be dead. Code that is dead can be eliminated without affecting the behavior of the program.

- *Malware* [53] is any software that is developed for the purpose of doing harm to computers or via computers. Malware can be classified in several ways, including on the basis of how it is spread, how it is executed and/or what it does. The main types of malware include worms, viruses, trojans, backdoors, spyware, and rootkits. Worms and viruses are computer programs that replicate themselves without human intervention. The difference is that a virus attaches itself to, and becomes part of, another executable (i.e., runnable) program, whereas a worm is self-contained and does not need to be part of another program to replicate itself. A trojan, or trojan horse, is software that is disguised as a legitimate program in order to entice users to download and install it. A backdoor (usually written as a single word) is any hidden method for obtaining remote access to a computer or other system. Spyware is software that is installed in a computer for the purpose of covertly gathering information about the computer, its users and/or or other computers on the network to which it is connected. A rootkit is software that is secretly inserted into a computer and which allows an intruder to gain access to the root account and thereby be able to control the computer at will.

Chapter 3

Malware

In this chapter, we describe a new area known as Internet of Things, where malware is causing serious troubles. We continue with a problem with modifying executable files to obstruct reverse engineering of such files, this process is known as the obfuscation and it can be done by more techniques which are explained in the following section.

3.1 Internet of Things

According to the company Gartner Inc., the Internet of Things is the network of physical objects that contain embedded technology to communicate and sense or interact with their internal states or the external environment. By these physical objects we can imagine all electronic devices that use to be denoted as intelligent, e.g DVD players, fridges, or toasters too. It is becoming more interesting with increasing usage of protocol IPv6, which provides a possibility for each such a device to have own IP address. In another words, such a device can be easily targeted and attacked. This result of this attack may be that all your friends receive the spam messages from your fridge.

From the view of malware analyst, the complication is the large number of several different architectures. These new devices may come with new generations of architectures and processors, what will put bigger demands on the analysts. Already now, there are used MIPS, ARM, PowerPC, or SuperH architectures and malware starts to aim at them. This is a new approach, because in the close history there was a trend to attack computers with x86 architectures, or mobile phones usually on ARM architecture. This new trend brings requirements for new ways how to detect malware.

There is an estimation from Cisco [14] that there are more than 12 billions of connected people, processes, data and things to the internet. Cisco expects the number of things to reach 1.8 trillion in 2020. Based on this we can expect that this is going to be very interesting field for malware creators. Example of such a malware is a worm Linux.Darlloz, which was discovered by Symantec [51]. We have analyzed this worm also by our decompiler and the analysis is described later in 5.8.2.

3.2 Obfuscation

The obfuscation is a thoughtful process of modification with aim to hide information without causing any damage to this information. We know several types of obfuscation and they are shown in the following text. This section is created from the article [67].

3.2.1 Types of Obfuscation

The general division dictates two main areas:

- binary files obfuscation
- source files obfuscation

Another division can be made according to a purpose:

- hide new algorithm, technology (protect intellectual property)
- hide well-known algorithm, technology (prevent unwanted detection)

In some point of view, we can say that obfuscation wants to achieve security through obscurity, what can be successful only in some particular cases.

3.3 Obfuscation of Binary Files

The base for this obfuscation is knowledge of an instruction set of aimed architecture, because it is natural that obfuscator for ARM executable would damage executable for the x86 architecture or other incompatible platform. The advantage is that well-written obfuscator can be usable for more executable formats of the given architecture. It is common to see obfuscator for both PE and ELF format on the x86 architecture [1].

We can take a look on the most used techniques [34]:

- Dead-code insertion

  The idea is same as inserting instruction `NOP`, but a single instruction would not be very helpful, because it can be simply filtered. Dead code does nothing useful at all, it only decreases performance and confuses the code. It can consist of various complex algorithms, which unbend attention from real object of the inspection. We use AT&T syntax.

```
push %eax
push %ebx
push %ecx
;some magic with eax, ebx, ecx
pop %ecx
pop %ebx
pop %eax
```

- Code transposition

  It is based on a finding of independent pieces of codes, and their mutual exchange. Transposition can be made on two instructions, but it can also be done on whole blocks. It depends on the skills of author how precisely he can determine independent blocks of code. Another way of code transposition is adjusting jumps and calls, and reassembling blocks of such code.

```
mov (%ecx), %eax
mov $10, %ebx
mul %ebx
```

```
mov $10, %ebx
mov (%ecx), %eax
mul %ebx
```

- Register realignment

  It is simple method, when we exchange the certain number of registers. The code works with other registers, so the bytes in binary code, which represent used registers, are different, but an algorithm is still the same one. In another words, we create different admissible permutation of registers for the given code.

```
mov %ebx, %eax
xor %ecx, %ecx
test %ecx, %eax
```

```
mov %edx, %ecx
xor %eax, %eax
test %eax, %ecx
```

  The important note is that there are some restrictions. It is definitely not be a good idea to exchange register `esp` on the x86 architecture, because it causes a corruption of stack. Such a corruption ends with the crash of application.

- Instruction substitution

  Instruction set of the x86 architecture is very wide, so the single action can be performed by more combinations of instructions [31]. This fact is used for substitution technique. We can distinguish 3 subcategories according to the change of code size:

  – code expansion - new code is formed by more instructions than original

```
add $4, %eax
```

```
add $2, %eax
add $2, %eax
```

  – code shrinking - new code has less instructions than original

```
add $100, %eax
mul $0
inc %eax
```

```
mov $1, %eax
```

  – code alternating - new code has same size as original; the following three blocks do the same

```
mov $0, %ebx
mov %eax, %eax
```

```
xor %ebx, %ebx
mul $1
```

```
sub %ebx, %ebx
add $0, %eax
```

A very popular approach is exchanging instructions, which have different semantics, but with clever updates they have the same result. We present it on the replacing of `push` and `call`:

```
push %eax
```

```
sub $4, %esp
mov %eax, (%esp)
```

```
call sub_count                          push $0x401020
                                        jmp sub_count
```

In the first example, we decrease stack pointer, and after that we store value from `eax` to stack. If we use in the following code `pop` the beginner could be confused due to no `push`. The second example uses the fact how the instruction `call` works, it stores the address, where the control should be returned after finish of called subroutine. This approach has a great advantage. You can set an arbitrary address for continuing after return from function.

The binary form of files provides another opportunities for code obfuscation, the good example is function call dispatching, what is nicely implemented in PEScrambler [27]. The technique is based on the redirections of all internal and external function calls to a single function, which acts as the dispatcher. The result is that all instructions `CALL` has the same operand and the reverse engineer does not know which specific function is called from the dispatcher.

3.4 Obfuscation at the Source Code Level

Source code obfuscation can be divided into two types according to the result of the obfuscation:

- obscure source code to make it more difficult to read and understand
- obscure compiled executable to make more difficult to understand its disassembly

In this thesis we are interested in the second type.

Editing source code is essentially different from editing binary code. There is no possibility to use some introduced techniques from the previous section, for example register realignment.

On the other hand, we can easily hide the data in program. We can imagine that all strings can be written in an encrypted form, and at the moment of use there will be called a decryption function, so the base function will get the data in correct form, but in the source and also in the data section of executable, the strings will be illegible. The encrypted form and the decryption function can be changed for each compilation, so every released version could have different data section and also different code for the decryption routine. This method could be utilized by botnet owners for better hiding of bots. The majority of bots support self update and by this way they could be updated regularly by new version.

We can also approximate function call dispatching, the function calls will be redirected by the single dispatcher, but from the code it will not be easy to recognize, which function is really called. This is usually applied for the linked functions from the external libraries. On the other hand, this method can hide the imports of variables from such libraries too. The trick is done by using WINAPI functions `LoadLibrary` and `GetProcAddress`[1]. The following example obfuscates the call of `CreateFileA`.

```
HMODULE hDLL = LoadLibrary("Kernel32.dll");
if (hDLL) {
    fCreateFile = GetProcAddress(hDLL, "CreateFileA");
    // call by fCreateFile
}
```

[1] `http://msdn.microsoft.com/en-us/library/ms123401.aspx`

After successful running of this code, we can use the variable `fCreateFile` in the same way as WINAPI function `CreateFileA`. Still it is not so great. In spite of the fact that we do not see a direct call to WINAPI function, the function name is stored in the data section as a string. Anyway, this can be solved by the string encryption, which encodes the both strings in this example, so the name of the function and the library will be completely hidden. The problem is a recognition of appropriate functions and loading the correct library for the each one. The solution is a database of libraries and corresponding functions, which would be complex and it has to be periodically updated.

Chapter 4

Reverse Engineering

Reverse engineering is a process of an examination, not change or replication [11]. We take reverse engineering as a process, which has a goal to understand properties, states, and architecture of an examined object, e.g. software. The result of such a process should be information how we can build the object again. The process can be divided into several subgroups according to a level of abstraction.

Redocumentation is the most simple one. It aims only on recreating of a product documentation. The extraction of a design is more complicated. The process covers the creation of design according to available documentation, personal experience or skills from the product area. The most complicated process is a re-engineering. It includes an upgrade or enhancement of an existing product.

4.1 Software Reverse Engineering

Reverse engineering is widely spread in the software engineering. In Figure 4.1 there is shown a relation between software engineering and reverse engineering. Cryptoanalysis is a discipline, where reverse engineering is used to reveal a weakness of security ciphers implementation. The producers of integrated circuits battle with reverse engineering, because the development is expensive and rivals can save a lot of money by copying the design of circuit.

We focus on reverse engineering in the area of software development. For the simple analysis of a program we can use a debugger. This program requires a knowledge of assembly language and it allows us to analyze only running program, so we do not cover the parts of code, which are not used during the execution. This analysis belongs to a dynamic analysis. The advantage is that we see the stored values in registers or in the stack.

The basic tool for the static analysis – analysis does not need to execute a program is a disassembler. It works in a opposite way to assembler, it converts machine code into assembly language. It is hard to read this code as it is for the debugger. The disassembler supports specific architecture, what is given by the differences between instructions sets of the architectures. The process of disassembling meets several important issues:

- division of the code and data – it touches mainly Von Neumann architectures and it is equivalent to Halting Problem [29]
- decoding of instructions – it affects architectures with variable size of instructions

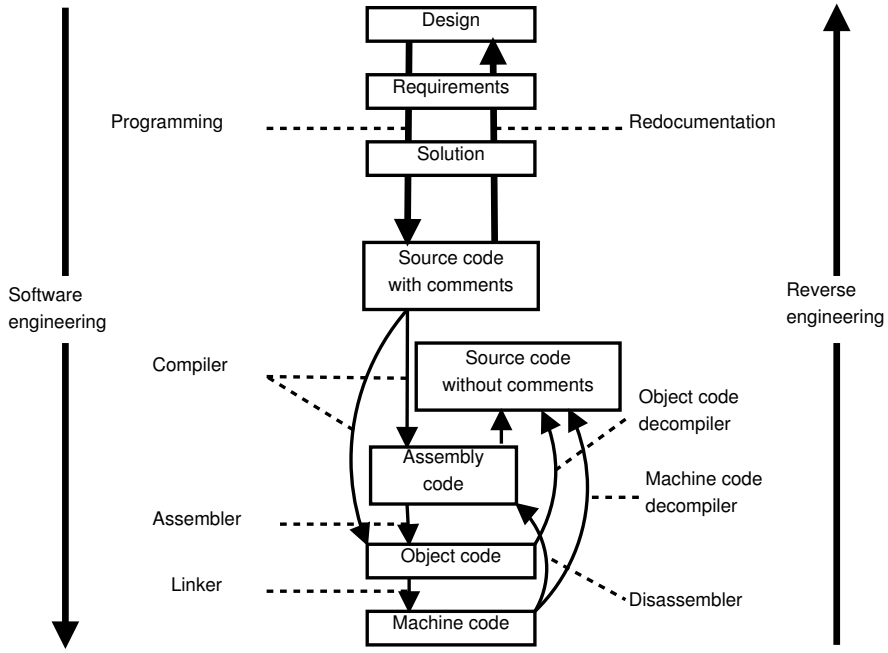


Figure 4.1: Relation between reverse and software engineering [39].

The mainly used disassemblers are: IDA Pro, objconv, Bastard, objdump, or Dissy that is practically a graphical interface for objdump.

This thesis is interested in decompilation. Decompilation is an advanced technique of reverse engineering. It is a process of transformation the binary executable code into higher programming language [12]. In another words, it is a reverse process to compilation. The Figure 4.2 illustrates both processes. The compiler parses the source code, creates an internal representation, runs analyses and optimizations, and finally produces binary code. The decompiler follows same principals, but it works in the reverse way.

4.2 History of Decompilation

The history of decompilers stretches back more than 50 years. Decompilers process various input formats and translate them into different types of the high level language (HLL) representations. This chapter is not intended to be an exhaustive list of all existing decompilers; we present only few milestones for each input-format category. A more detailed description can be found in [22].

4.2.1 Machine Code Decompilers

Machine-code decompilation has a surprisingly long history. Halstead [26] reports that the Donnelly-Neliac (*D-Neliac*) decompiler was producing Neliac (an Algol-like language) code from machine code in 1960. Decompilers at this time were based on pattern matching, and left more difficult cases for solving by the programmers. Barbe's *PILER System* was the first attempt to build a general decompiler. The system was able to read the machine code of several different target architectures, and generate code for several different HLLs. Only one input phase was completed (for the GE/Honeywell 600 machine) and only two output phases were written (Fortran and COBOL) [6].

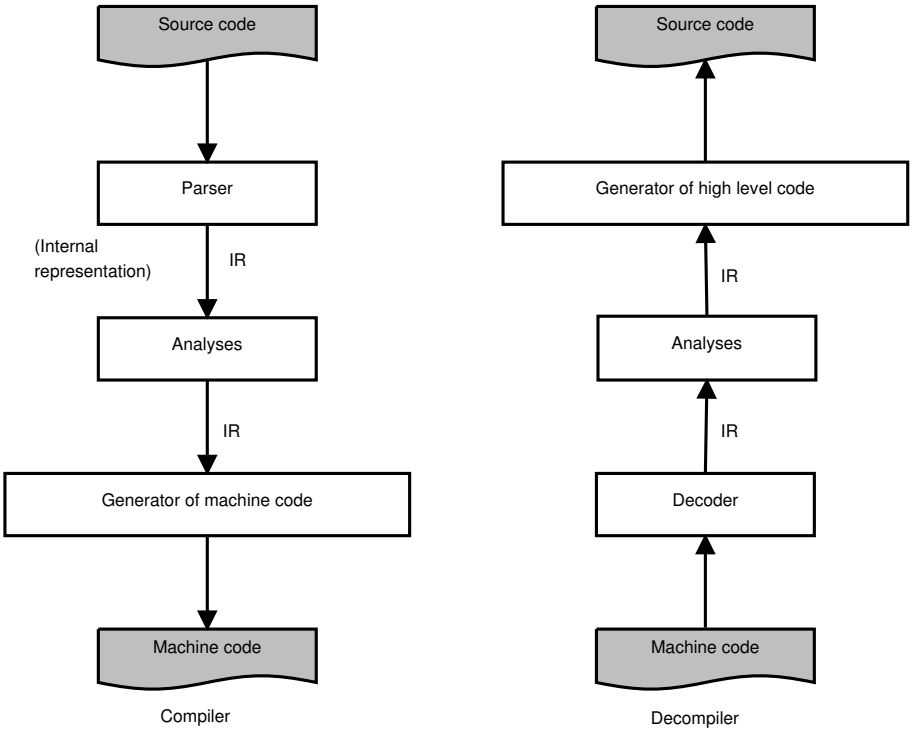


Figure 4.2: Workflow of the compiler and decompiler [22].

Exec-2-C was an experimental project by the company Austin Code Works, and it was not completed. Intel 80286/DOS executables were disassembled, converted to an internal format, and finally converted to C. Machine-related properties, such as registers and condition codes, were visible in the output. According to [28], only a very basic recovery of the C language code (e.g., conditional statements, loops) was performed. The University of Queensland Binary Translator (*UQBT*) uses a standard C compiler as the back-end. In other words, it 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 [65].

4.2.2 Object Code Decompilers

Object-code decompilers have several advantages over machine-code decompilers, but are less common, presumably because the availability of object code without source code is low.

Schneider and Winger in 1974 took a contrived grammar for a compiler and they inverted it to produce a matching decompiler [59]. This works only for a particular compiler, and only under certain circumstances; it was shown in [59] that Algol 60 could not be deterministically decompiled. It generally fails in the presence of optimizations, which are now commonplace. In 1988, there was written a quick decompiler *Decomp* for a specific purpose—to port a game from one platform to another, without having the original source code. Decomp was an object-code decompiler, and produced files that needed significant hand editing before they could be recompiled.

4.2.3 Assembly Decompilers

There was a pressing need to convert assembly language programs (second generation languages) to HLLs (third generation languages), but this is a somewhat easier task than machine code decompilation because assembler programs contain valuable information about symbolic names and linked functions as well as information about data and code separation, which are not available in executable programs. For example, it eliminates the problem of separating data from instructions in the parsing phase of a decompiler.

On the other hand, whenever an input assembly language code is obtained from executables (e.g., by disassemblers), all the previously mentioned benefits are missing and decompilation of such assembly code is roughly equal to decompilation of executables (e.g., the data versus code problem).

Zebra was a prototype decompiler developed at the Naval Underwater Systems Center [8]. It was an assembly decompiler, this time emitting another assembly language. The report concluded that decompilation to capture the semantics of a program was not economically practical, but it also noted that it is useful as an aid for program porting. University of London's *asm21toc* reverse compiler was created in 2000. This assembly language decompiler for Digital Signal Processor's (DSP) code was written in a compiler-compiler called *rdp*. The authors note that DSP is one of the last areas where assembly languages are still commonly used. This decompiler has to face several problems unique to DSP processors [33].

4.2.4 Decompilers for Virtual Machines

Virtual machine specifications (like Java or .Net bytecode) are rich in information such as names and types, making decompilers for these platforms much easier. However, good type analysis is still necessary for recompilability.

The Sable group at McGill University, Canada, have developed a framework for manipulating Java bytecode called *Soot*. The main purpose of Soot are optimizations of bytecode, but they have also built a decompiler called *Dava*[1] on top of Soot. With Dava, they have been concentrating on the more difficult aspects of bytecode decompilation, like typing of local variables, generating stack variables, or structuring. *JODE* (Java optimize and decompile environment)[2] is an open source decompiler and obfuscator/optimizer. JODE has a verifier, similar to the Java runtime verifier, that attempts to find type information from other class files. JODE is able to correctly infer types of local variables and to transform code into a more readable format, closer to the way Java is naturally written, than early versions of Dava.

4.3 Decompilers

In this section, we introduce the nowadays most popular machine-code decompilers and other projects related to decompilation. Description given in this section is mostly based on the official information presented by the authors of these tools. We will focus on supported target architectures, object file formats (OFFs), and other features.

[1] `http://www.sable.mcgill.ca/dava/`
[2] `http://sourceforge.net/projects/jode/`

4.3.1 The dcc Decompiler

The *dcc* decompiler was developed by Cristina Cifuentes while she was a PhD student at the Queensland University of Technology in 1991–1994. It was introduced in her dissertation thesis [12]. The dcc decompiler is distributed under the GPL license.

The structure of the decompiler resembles that of a compiler: a front-end, middle-end, and back-end which perform separate tasks. The front-end is a machine-language dependent module that reads in machine code for a particular machine and transforms it into an intermediate, machine-independent representation of the program. The middle-end (as known as the Universal Decompiling Machine or UDM) is a machine and language-independent module that performs 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 [61].

This decompiler was developed as a prototype for the Intel i80286 architecture and for DOS executables. dcc uses compiler and library signature recognition to decompile user routines only. The amount of signatures is very limited due to a narrow range of decompiled targets. The decompiler provides comments for each subroutine, and it has command switches to generate the bitmap of the program, call graph, output assembler file, statistics on the number of low-level and high-level instructions in each subroutine, and information on the control-flow graph of each subroutine [12].

4.3.2 Boomerang

Boomerang[3] is an open source project. It was strongly inspired by UQBT—A Resourceable and Retargetable Binary Translator in 1996 [65], and it was established in 2002. The original author is Mike Van Emmerik. Boomerang was originally released under a BSD-like license, however, it tends to be more GPL-oriented in its latest release (2006).

It works to ease the pain of reverse engineering by searching for patterns in machine code and replacing them with equivalent C constructs. It uses a series of algorithms that convert machine code to C code and then it makes automatic substitutions throughout. Ideally, all that is left for the reverse engineer is to rename the variable and function identifiers. Boomerang also accepts a set of hints that specify the names of known data structures so that the program can automatically replace those names as they are seen in the decompiled code [47].

Boomerang can decompile the code for several architectures: Intel x86, SPARC, and PowerPC. It supports commonly used object formats like UNIX ELF, Windows PE, and Apple Mach-O. The major limitation is missing recognition of statically linked code. This causes complications for decompiling programs that were compiled with the option `-static` and, because, 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.

The Boomerang decompiler is probably the first attempt to create a retargetable decompiler by using a domain-specific language for description of the target architecture. The SLED language, developed within the New Jersey Machine-Code Toolkit [56] project, was used for this purpose. This project exploits the SLED language [55] for compact description of instruction syntax and coding. However, this language does not support description of instruction semantics. Therefore, this language itself cannot be used for generation of tools like compilers or decompilers. Therefore,

[3] `http://boomerang.sourceforge.net/`

the authors of the Boomerang decompiler have to use it together with the RTL-based semantics description language SSL [13]. According to the 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 disassembled and it is expanded into the respective SSL semantics. When `jump` or `call` instructions are encountered, the decoding is performed by following the execution paths; this approach of disassembling is called *recursive traversal* because the program flow is followed. During decoding, sequential instructions are grouped together into basic blocks, which are connected together afterwards, according to flow transition instructions, in order to construct the control-flow graph. The instructions belonging to the same basic block satisfy the property that all of them are always executed before all the subsequent ones.

Once a code fragment is completely decoded and its control-flow graph is built, it is transformed into a static single assignment form (SSA) [22]. The particularity of this representation is that every definition of a program variable gives a rise to a new variable, thus the same variable is defined only once. For example, the first definition of the variable A generates the variable A_1 and the j-th definition generates the variable A_j. The main advantage of the SSA form is that it allows to enormously simplify the data-flow analysis process because it makes explicit the relation between the use of a variable (when it appears inside an expression) and its definition (when it appears on the left-hand side of an assignment). A trick has to be used when more than one definition reaches the same use, but through different paths: a special statement, called ϕ-statement, is inserted at the beginning of the basic blocks that uses these definitions, and it is used to define a new variable which indicates the use of multiple concurrent definitions. For example, if both $A_i := 0$ and $A_j := 1$ reach the instruction $B := A + 1$ of a block k, the instruction $A_k := \phi(i, j)$ is inserted before B's definition, which is then translated into $B := A_k + 1$ to explicitly specify that the assignment uses either A_i or A_j [9].

4.3.3 REC Studio

REC Studio—Reverse Engineering Compiler [57] is a freeware, but not open-source, interactive decompiler, which is still under development. It reads a Windows, Linux, OS X or raw executables (e.g., firmware), 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. The software is available for mainly used platforms: Windows, Linux (Ubuntu), and OS X. However, this software is unstable on several architectures (e.g., Windows), and it often crashes during decompilation.

The author wrote on his web page [57] that the disassemblers used in REC were taken from various sources. Due to this fact, we estimate that it is very complicated to add support for a new architecture. Also, it is a considerable amount of code from different origins, what makes it hard to maintain. REC has loaders for more object file formats (OFF): PE, ELF, COFF, and Mach-O. We can estimate that there is unique code for each loader. Its author also claims that debugging information is also supported and the decompiler can process the DWARF format and, partially, the PDB format.

The architecture of the decompiler is presented in Figure 4.3. The input executable is processed by a file reader, which supplies data to the symbol table and disassembler. Procedure finder uses the symbol table and code walker to detect functions. Then, the procedure analyzer is run over

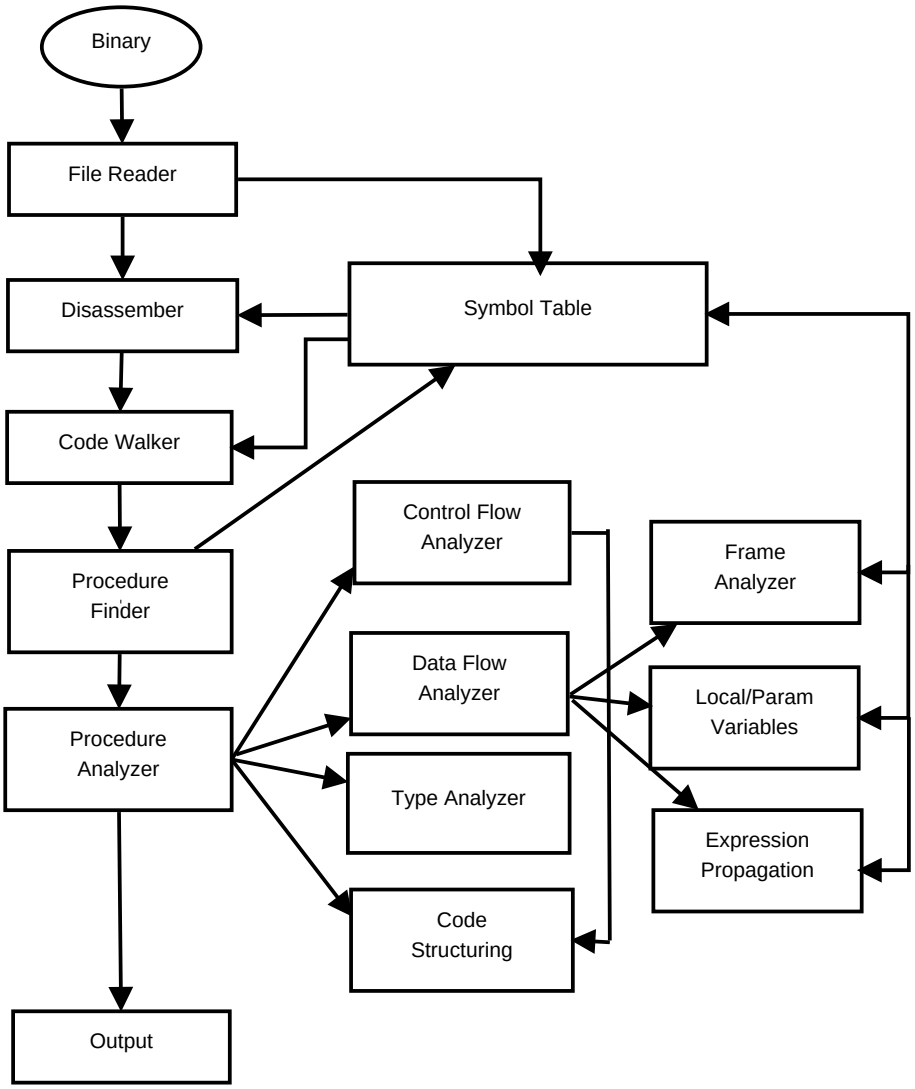


Figure 4.3: The architecture of REC Studio decompiler [57].

the detected functions and it cares about the coordination of control flow, data flow, type analyzer, and code structuring. Data-flow analysis has specialized analyzers for frames, local variables, parameters, and expression propagation. Finally, the result of the procedure analyzer is taken by the output module to print the HLL code.

4.3.4 Hex-Rays Decompiler

The *Hex-Rays Decompiler*[4] is the nowadays decompilation "standard". It is implemented as a plugin to the IDA disassembler[5]. The Hex-Rays Decompiler supports the x86 (also x86_64) and ARM target architectures. It also supports both major OFFs—ELF, PE, Mach-O. 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.

[4] `www.hex-rays.com/products/decompiler/`
[5] `www.hex-rays.com/products/ida/`

This software is commercial and distributed without sources. The first version of the x86 decompiler was released in 2007, support of ARM decompilation has been added in 2010, and support for x86_64 has been added in 2014. The current version is 1.7, and there is no plan for additional supported target platforms. 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 which gives access to the decompiler's internals and one can easily create new plugins or scripts (using Ruby or Python). It also has a 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).

4.3.5 Decompile-it.com

The author of this project is Naftali Schwartz. Up to the day of this document creation, there is no published article about this project, therefore, its description is very brief.

This project is tightly linked to the Valgrind[6] 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) and it is distributed under the GPL license.

The project has not yet been released and it is available only as a web interface[7] which is limited to the decompilation of x86/Linux executables. It seems that debugging information is mandatory. The official site claims 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 author's note [60], the MIPS and ARM architectures are also supported as well as C++ generated executables. The license of this tool is unknown..

4.3.6 SmartDec

The original name of this decompiler is *TyDec* and was firstly presented in [64]. In present, its license is unknown and it is distributed as a demo application without sources.

Later, it was renamed to *SmartDec*[8]. The work on the decompiler was based on a research related to automatic type reconstruction in disassembled C programs [19]. This is the reason why SmartDec takes assembly code as the input.

[6] `http://valgrind.org/`
[7] `http://decompile-it.com/`
[8] `http://decompilation.info/`

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.

SmartDec performs decompilation in several steps:

1. Parsing of the input assembly listing and creation of program representation as a sequence of assembly instructions. SmartDec currently handles the GNU objdump and Microsoft dumpbin output formats.
2. Building of the control-flow graph and isolation of functions.
3. Transformation of assembly instructions into platform-independent program representation.
4. Analysis of functions:
   (a) joint reaching definitions and constant propagation analysis;
   (b) dead code elimination;
   (c) reconstruction of local variables, function arguments, and return values;
   (d) reconstruction of integral and composite types;
   (e) structural analysis, including the reconstruction of compound conditions and loops.
5. HLL code generation, optimization and output.

4.3.7 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 [17]. 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 recompilable in Delphi.

The *Jakstab project* [37] is a static analysis framework written in Java which currently supports the x86 architecture and 32-bit PE or ELF executables. Its purpose is not decompilation, but translation of machine code to 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 [38], 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 part of Johannes Kinder's PhD thesis [36].

4.3.8 Comparison of Decompilers

In Table 4.1, we summarize the base features of the decompilers and knowledge obtained from our tests. The features marked with an asterisk (*) are claimed by the authors but are not included in any publicly available release.

From this table, we can see that the most popular file formats are Windows Portable Executable (PE) and UNIX/Linux ELF formats, as well as the Intel x86 architecture. The C language is the most

	dcc	Boomerang	REC Studio	Hex-Rays Decompiler	decompile-it.com	SmartDec
Supported architectures	x86	x86 SPARC PPC	x86 SPARC MIPS	x86 ARM x86_64	x86 MIPS* ARM*	x86
Supported OFFs	DOS-MZ	ELF PE	ELF PE COFF Mach-O	ELF PE Mach-O	ELF	none
Input	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	commer-cial	GPL?	unknown
DWARF dbg support	✗	✗	✓	✓	✓	✗
PDB dbg support	✗	✗	partial	✓	✗	✗
Interactive interface	✗	✓	✓	✓	✗	✗
Statically linked code detection	✓	✓	✗	✓	✗	✗
Retargetability	no	yes	unknown	unknown	unknown	unknown
Documentation	yes	yes	partial	yes	no	no
Quality of output	not tested	middle	middle	high	not tested	low

Table 4.1: Overview of features and tests of existing decompilers

common output language. Other features are supported less often. Unfortunately, we have to state that none of actively developed decompilers is available with its source code under a non-restricting license. Moreover, most of these tools tend to be proprietary software.

We have presented and commented the results of four decompilers. According to our judgement, Hex-Rays Decompiler is the best one. Boomerang takes the second place. REC Decompiler is on the third place, where the main reason are failures for function detection. The last one is SmartDec.

Except SmartDec, other decompilers are able to recognize and use strings from data section, which helps a lot for code understanding. Conditions handling is very important and fortunately it is implemented by all decompilers. SmartDec is not able to reconstruct loops, other compilers are able to do it, but they generate a `do-while` or `while` loop. Moreover, REC Decompiler uses an endless loop with `break` inside of `if` for loop termination. Hex-Rays works with arrays better than the other decompilers. It declares a `char` array instead of an `int` array, but it is the single decompiler that generates at least some declaration. Boomerang generates an access to an array without declaring the array, REC Decompiler and SmartDec do not recognize arrays and use pointer access. Hex-Rays handles a lot of widely used idioms. Boomerang decompiles idioms code partially so it is hard to decide. REC Decompiler does not handle idioms and it generates code directly according to the output of the compiler, SmartDec moves code related to idioms to separated functions which are not called from `main`. Function detection is quite successfully done by all decompilers except for REC Decompiler. The program for calculation of Ackermann function is correctly decompiled only by Hex-Rays Decompiler.

4.4 Future of Machine-Code Decompilation

There is a large group of challenges for machine code decompilation, which are waiting for more competitive resolution. Briefly, we can mention the following challenges.

- **Executables produced by a C++ compiler**. The C++ language supports much more complicated constructions than the C language and with the new standard ISO C++11, it becomes more and more powerful. With a growing power of the language, the complexity of its compilers grows too. This is, of course, reflected in complexity of the generated code. So, binary code of the executable is complicated and the original constructions are also complicated, and usually hard to transform into an understandable C source code.

  A short list of new features from the last C++ standard, which may cause complications:

  - lambda functions and expressions
  - multithreading memory model, including thread-local storage
  - garbage collection

- **Structures and unions**. These constructs are hard to decompile because the members of a structure can work as independent variables, and from machine code, any context connection could be invisible. This problem is quite well solvable when debug information is available and this information exactly tells which structures are used and which variable belong to it. On the other hand, in terms of malware, debug information is almost never available, so this can be in practice used only for self-testing. Now, we see a potential in type information of the standard library functions. If there is used, e.g., `fopen`, the decompiler can know that

it returns a pointer to `struct FILE`, and it can work with this knowledge later, when the pointer is used.

- **Obfuscated code**. Obfuscated code is used for hiding the meaning of code. Obfuscation is also used by malware, which wants to prevent the inspection of its code. The decompilation of malware would be very beneficial for analytics from anti-virus companies. Obfuscated code is specific for its usage of a wide variety of different code constructions which differ a lot from usual constructions generated by compilers. An obfuscator may divide functions in more separated blocks, which are located on more places through the executable. This makes it very hard to recognize such functions and the same applies for the reconstruction of arguments. Arithmetic expressions are changed to explore more difficult operations. Furthermore, dead code may be inserted on arbitrary places. This issue is taken into account in the Lissom decompiler.

- **Executables produced by compilers for functional languages (e.g., Haskell)**. Functional programming is completely different in comparison with procedural or object-oriented programming. Also, the generated executable has different structure, which is non-standard for decompilers aimed on processing executables from procedural language and producing the output in (imperative) procedural languages.

- **New processor architectures**. New processor architectures are presented much more often. More powerful and productive methods allow to develop new processors faster and cheaper. We can expect that new processors will come to market regularly. Such a situation requires decompilers to be retargetable; otherwise, they would not be able to react fast enough to support new architectures.

- **Parallel computing, multithreading**. This is also related to GPU (Graphics Processing Unit), which is still more and more used for various computing tasks. There are several different forms of parallel computing: bit level, instruction level, data, and task parallelism. A decompiler will have to reveal what is run in parallel and generate the output code according to that.

Chapter 5

Lissom Decompiler

Decompiler is developed within the team, where the author of thesis is one of the team members. Author is mainly involved in research and development of the front-end. Therefore this part is described with more details. However, other parts are also described for providing the overall view on the decompiler framework. This chapter is based on articles [42, 68, 70–75, 77].

We present an overview of a retargetable decompiler. Our approach is not tied to any particular target platform. The primarily utilization of this tool is a static platform-independent malware analysis. With its help, it is possible to inspect malware code on a much more abstract and unified form of representation, while preserving the functional equivalence of the code. Therefore, malware analysts do not need to have a deep knowledge of the target platform (i.e. instruction set and processor architecture) and they can fully focus on the malware analysis.

The retargetable decompiler is based on exploitation of the ADL ISAC [44], which is intended to be used for designing new application-specific instruction set processors (ASIPs). However, we use this formalism for the description of existing platforms. The front-end of the decompiler uses generated instructions semantics from this description. The decompiler core is based on the LLVM Compiler System[1], which we use for a translation from LLVM IR code into HLL.

5.1 ISAC Language

The ISAC language [44] was developed within the Lissom project at Brno University of Technology [43]. The project has two basic scopes. The first scope is a development of an ADL for the description of Multiprocessor Systems-on-Chip (MPSoC). The second scope is a transformation of MPSoC description into advanced software tools (e.g. a C compiler, a simulator, etc.) or into a hardware realization of each processor. The ISAC language belongs into a so-called mixed ADL. It means that a processor model consists of several parts. In the resource part, processor resources, such as registers or memory hierarchy, are declared. In the operation part, processor instruction set with behavior of instructions and processor micro-architecture is described. Processor model can be written in two levels of accuracy—instruction-accurate or cycle-accurate. The retargetable decompiler currently uses the first one.

The *assembler* and *coding* sections capture the format of instructions in the assembly and machine language, so they define instructions in textual and binary forms. For the behavioral model, the

[1] `http://llvm.org/`

```
RESOURCES {                    // HW resources
  PC REGISTER bit[32] pc;      // program counter
  REGISTER bit[32] regs[16]; // register file
  RAM bit[32] memory {SIZE(0x10000); FLAGS(R, W, X); };
}
OPERATION reg REPRESENTS regs
  { /* textual and binary description of registers */ }
OPERATION op_add {             // instruction description
  INSTANCE reg ALIAS {rd, rs, rt};
  ASSEMBLER { "ADD" rd "=" rs "," rt };
  CODING { 0b0001 rd rs rt };
                              // instruction behavior
  BEHAVIOR { regs[rd] = regs[rs] + regs[rt]; };
}
```

Figure 5.1: Example of a ISAC language source code.

behavior section is used. In this section, a subset of the ANSI C language can be used. The behavior section defines the semantics of each operation. For example, a simple instruction with its behavior is described using the assembler, coding, and behavior sections, see Figure 5.1.

5.2 LLVM Compiler System

The LLVM Compiler System was originally designed as a compiler framework to support transparent, lifelong program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time and in idle-time between runs [3]. Nowadays, the use of LLVM spans over many different areas, including compilation (Clang, LLVM D Compiler, Trident Compiler), video decoding (Jade), signal processing (Faust), static checking (Calysto), and implementation of various programming languages (Unladen Swallow, Rubinius, Pure). The key features of LLVM include a universal, language-independent instruction set, type system, intermediate representation (LLVM IR [2]), many built-in sophisticated optimization algorithms and passes, link-time optimizations, just-in-time (JIT) code generation, and application programming interface for several programming languages.

Consider the C source code in Figure 5.2. This straightforward implementation of the factorial function can be directly compiled into the LLVM IR. The output of this conversion is shown in Figure 5.3. This example shows us some of the properties of the LLVM IR:

- The used RISC-like instruction set captures the key operations of ordinary processors, but avoids most of machine-specific constraints. Most instructions are in the three-address form—they take either one or two operands and produce a single result. The instruction set includes arithmetic instructions (e.g. `add`, `mul`), bitwise instructions (e.g. `shl`, `and`), memory access instructions (e.g. `load`, `store`, `alloca`), conversion instructions (e.g. `trunc`, `zext`), and other instructions (e.g. `icmp`, `call`). Furthermore, every basic block ends with a terminator instruction (e.g. `br`, `ret`) which explicitly specifies its successor basic blocks.

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

Figure 5.2: A simple implementation of the factorial function in C.

- According to the presence of the `phi` instruction in Figure 5.3, the virtual registers are in the Static Single Assignment (SSA) form (see [16]), where each variable is assigned exactly once. The use of this form results in a simplification of many compiler optimizations.
- A language-independent type system is used. Every instruction and SSA register has an associated type and all operations obey strict type rules. This enables several optimizations which otherwise would not be possible (at least not in such a straightforward way). Primitive types include void, boolean, variable-sized integers, and floating-point types. Derived types include pointers, arrays, structures, and functions. The `cast` instruction can be used for type conversions (other ways of type conversions are not possible). Address computation and address arithmetic is done by the `getelementptr` instruction.
- The LLVM IR can exist in the following three forms: textual (as in Figure 5.3), binary (compiled textual representation), and in-memory (compiler internal representation). All of these representations are equivalent—that is, one can be transformed to the others without any loss of information.

5.3 Design of a Retargetable Decompiler

The objective of the decompiler is a static analysis of a binary code and its transformation into a HLL. It is important to preserve the functional equivalence of the transformed program; otherwise, further code analyses will be inaccurate. This is a very difficult task because we have to deal with missing information in the input code (e.g. because of compiler optimizations, malware obfuscation, etc.). The usage of the retargetable decompiler requires from user to describe the target architecture in the ISAC ADL. Then, the front-end of the decompiler can be automatically generated by a tool-chain generator based on this description. After that, it is possible to reversely translate binary executables for this architecture.

The toolkit consists of two main parts—the *preprocessing part* and the *decompiler core*, see Figure 5.4. The structure of the decompiler core is similar to a classical compiler. It consists of a front-end, a middle-end, and a back-end. The only platform-specific part is the front-end. For this purpose, the binary coding and semantics of each processor instruction is extracted from the architecture model in ISAC. This is a major difference against other retargetable decompilers, because it is not necessary to manually reconfigure the decompiler for a new architecture. It should be noted that in present, there is no other competitive method of automatically-generated retargetable decompilation.

```
define i32 @factorial(i32 %n) {
entry:
    %0 = icmp eq i32 %n, 0
    br i1 %0, label %bb2, label %bb1

bb1:
    %1 = add i32 %n, -1
    %2 = icmp eq i32 %1, 0
    br i1 %2, label %factorial.exit, label %bb1.i

bb1.i:
    %3 = add i32 %n, -2
    %4 = call i32 @factorial(i32 %3)
    %5 = mul i32 %4, %1
    br label %factorial.exit

factorial.exit:
    %6 = phi i32 [ %5, %bb1.i ], [ 1, %bb1 ]
    %7 = mul i32 %6, %n
    ret i32 %7

bb2:
    ret i32 1
}
```

Figure 5.3: The generated LLVM IR code from the code in Figure 5.2.

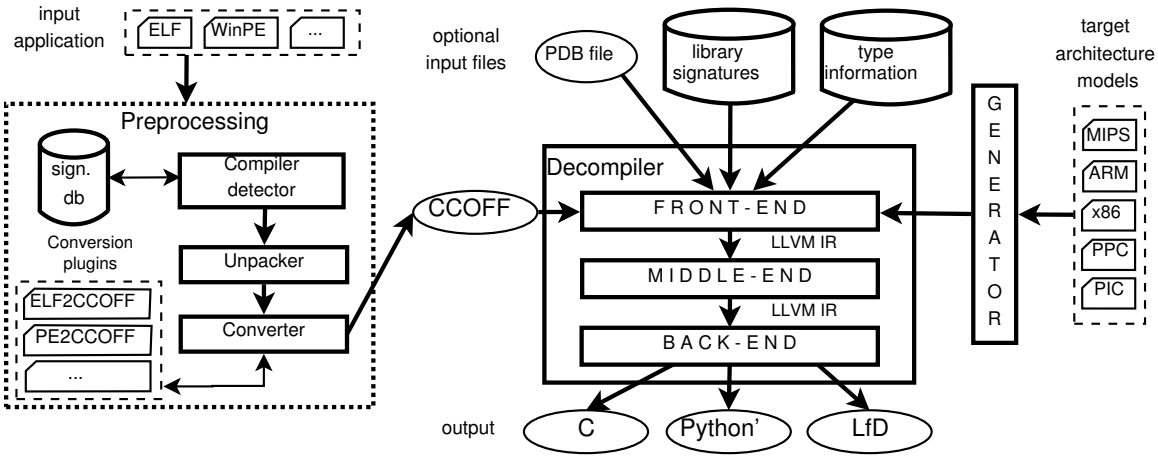


Figure 5.4: The concept of the retargetable decompiler.

The preprocessing part analyses the input application to detect the used file format, compiler, and, if the file was packed, the used packer. After that, it unpacks and converts the examined platform-dependent application into an internal object file format CCOFF (Codasip Common Object File Format). The conversion is done via our plugin-based converter [42]. We support conversions from Windows PE, UNIX ELF, Apple Mach-O, and other formats. Non-standard file formats can be supported via a direct implementation of the appropriate plugin, or via an automatic plugin generation based on the format description in our object-file-format description language [40]. Afterwards, such CCOFF files are processed by the decompiler core.

The decompiler core is built on top of the LLVM Compiler System. The LLVM assembly language, LLVM IR, is used as an internal code representation of the decompiled applications throughout the decompilation process. The core of our decompiler consists of three basic parts—a *front-end*, a *middle-end*, and a *back-end*, described next.

The unified CCOFF files are firstly processed by the front-end, which is the only platform-specific part of the decompiler because its instruction decoder is automatically generated based on the target architecture model in the architecture description language (ADL). The ISAC model is transformed by a *semantics extractor* [30], which transforms the semantic description (i.e. snippets of C code) of each instruction into a sequence of LLVM IR instructions, which properly describe its behavior. The extracted semantics and binary encoding of each instruction is used for an automatic generation of an *instruction decoder*. The decoder translates the application's machine code into sequences of LLVM IR instructions, which characterizes its behavior in a platform-independent way. This intermediate program representation is further analysed and transformed in the static-analysis phase of the front-end. This part is responsible for eliminating statically linked code, detecting the used ABI, recovering of functions, arguments, etc. [73]. When debugging information or symbols are present in the input application, we may utilize them to get a more accurate result. Although this may be useful during source recovery or code migration, this type of information is almost never present in case of malware, so we do not rely on it.

Afterwards, the LLVM IR program representation is optimized in the middle-end by using many built-in optimizations available in LLVM and our own passes (e.g., optimizations of loops, constant propagation, control-flow graph simplifications).

Finally, the back-end part converts the optimized intermediate representation into the target high-level language (HLL). Currently, we support three target HLLs: C, Python-like language, and LfD (specific language described in 6.2). The second one is very similar to Python, except a few differences—whenever there is no support in Python for a specific construction, we use C-like constructs. The conversion itself is done in a several-step way. First, the input LLVM IR is converted into another intermediate representation: *back-end intermediate representation* (BIR). During this conversion, high-level control-flow constructs, such as loops and conditional statements, are identified and reconstructed. After that, the obtained BIR is optimized, and finally, it is emitted in the form of the target HLL.

Apart from the target HLL, we are able to produce the call graph of the decompiled application, control-flow graphs for all functions, and an assembly representation of the application.

5.4 Preprocessing

Preprocessing part covers the convertion of the executable file into CCOFF file and gathering additional information about the executable file. Firstly, we describe the conversion issue and then, the the tool for inspection of executable and its output.

In present, we can find a variety of commonly used object file formats. Some of them are proprietary formats (e.g. Symbian E32Image, Apple Mach-O, Android DEX), others are open standards (e.g. UNIX ELF, COFF, PE). Some of these formats are open and well documented (ELF), and there are existing converters for them, while other formats are still being analyzed by reverse engineers (DEX). In the Lissom project, a specialized COFF-based file format called CCOFF is used for internal code representation. CCOFF is used by a complete set of retargetable tools that are

Supported architectures	x86 ARM + Thumb MIPS PowerPC
Supported OFFs	ELF PE COFF Mach-O
Input	binary
Output language	C, Py', LfD
Distribution	web-service
License	unspecified
DWARF dbg support	✓
PDB dbg support	✓
Interactive interface	✗
Statically linked code detection	✓
Retargetability	yes
Documentation	yes
Quality of output	high

Table 5.1: Overview of Lissom Decompiler features. Compare with 4.1

automatically generated in Lissom project (instead of decompiler, also retargetable disassembler or simulator).

CCOFF was designed in reference to independency on any particular architecture, universality and good readability. Therefore, it is possible to describe architectures with different types of endianity, byte sizes, instructions lengths, or instruction alignments. It is also possible to store executable, object, or library code within the CCOFF format.

The CCOFF structure is similar to the COFF format – basically, it has one header, followed by section headers, sections, and symbolic information (symbols, relocations, and debug information). The section's content is characterized by section flags. The format of CCOFF is textual; therefore, it is possible to study its content without any additional tools, see Figure 5.5.

Our solution is designed as a plugin-based system, where each plugin implements a conversion of one or more object file formats. The main converter (i.e. the host application) handles the user interface, manages conversion plugins, and provides the common functionality for plugins (e.g. CCOFF file manipulation) [45].

The conversion is done by plugins. Currently, we provide plugins for convering ELF, PE, E32Image, Mach-O, DEX, and their derivates. We use several existing third-party libraries for file formats manipulations. For handling ELF, PE, and Mach-O, the Binary File Descriptor (BFD) library is used. It is used for unified format manipulation because it supports a lot of object file formats [10]. During the conversion, the content of an input file is mapped to a BFD internal canonical structure, uniformly

```
AgT62kG9y7 //magic string
32    // word bit-size
4     // bytes per word
0     // byte order, 0-little, 1-big
...   // flags, entry point, etc.
30    // section count
1     // symbol table count
...   // information about sections
.text // *** section header *** - name
1     // is address absolute?
143654972   // section address
T     // section flags
      // section data follows
00111111110000001111110000000101
00000000000000000000000000000000
```

Figure 5.5: Example of the CCOFF format.

characterizing items like sections, section flags, symbols, and architecture type. Afterwards, this form is transformed to a structure suitable for the CCOFF format, and finally CCOFF file is saved.

The information about executable file is gathered by the application called *fileinfo*. It obtains the same information as the readelf utility, but independently on the used target format (i.e. it supports ELF, Windows PE and other common formats). Another its advantage is a built-in packer/compiler detector. The detection algorithm is based on pattern matching of the entry-point instructions with an internal signature database.

Example of signature can look like `5589E583EC18C7042401000000FF15--------E8`, where each character represents a nibble of instruction's encoding. Some code parts are variable as a target address in the call instruction. Such variable parts must be skipped during matching. In the signature they are represented by a wild-card character `-`.

Application *fileinfo* provides information for generation of XML configuration file for the front-end. The configuration of front-end allows setting a lot of options, therefore a XML file is generated by several scripts that cooperate with fileinfo.

The most important items in XML configuration are:

- `targetArchitecture` – architecture of input file (detected by fileinfo)
- `entryPointAddress` – address of entry point (detected by fileinfo)
- `toolName` – name of detected compiler or packer (detected by fileinfo)
- `versionInfo` – version of detected compiler or packer (detected by fileinfo)
- `semantics` – semantics of architecture
- `signatures` – files with signatures of statically linked code
- `types` – files with type information for functions
- `abis` – files with ABI descriptions
- `inputFile` – input file
- `outputFile` – output file

5.5 Front-end

The objective of the front-end is a translation from the CCOFF file into a sequence of low-level LLVM IR instructions. We use a name *decfront* for the front-end. As it was said before, it is a single part of decompiler, which is platform-independent and therefore it is generated according to architecture description. To be precise, an instruction decoder is generated, and the others analysis are generic. So, they are same for all architectures.

The large part of decfront is detection of statically linked code. This feature helps to decrease time of decompilation and also to improve the result. It is described closely in 5.5.1. The main part of the front-end is the static analysis of the decoded code before generation of final LLVM IR code. This part includes several specific analyses, some of them are architecture-specific. The cooperation and work flow of decfront is shown in Figure 5.6. This figure presents complete image of decfront architecture design.

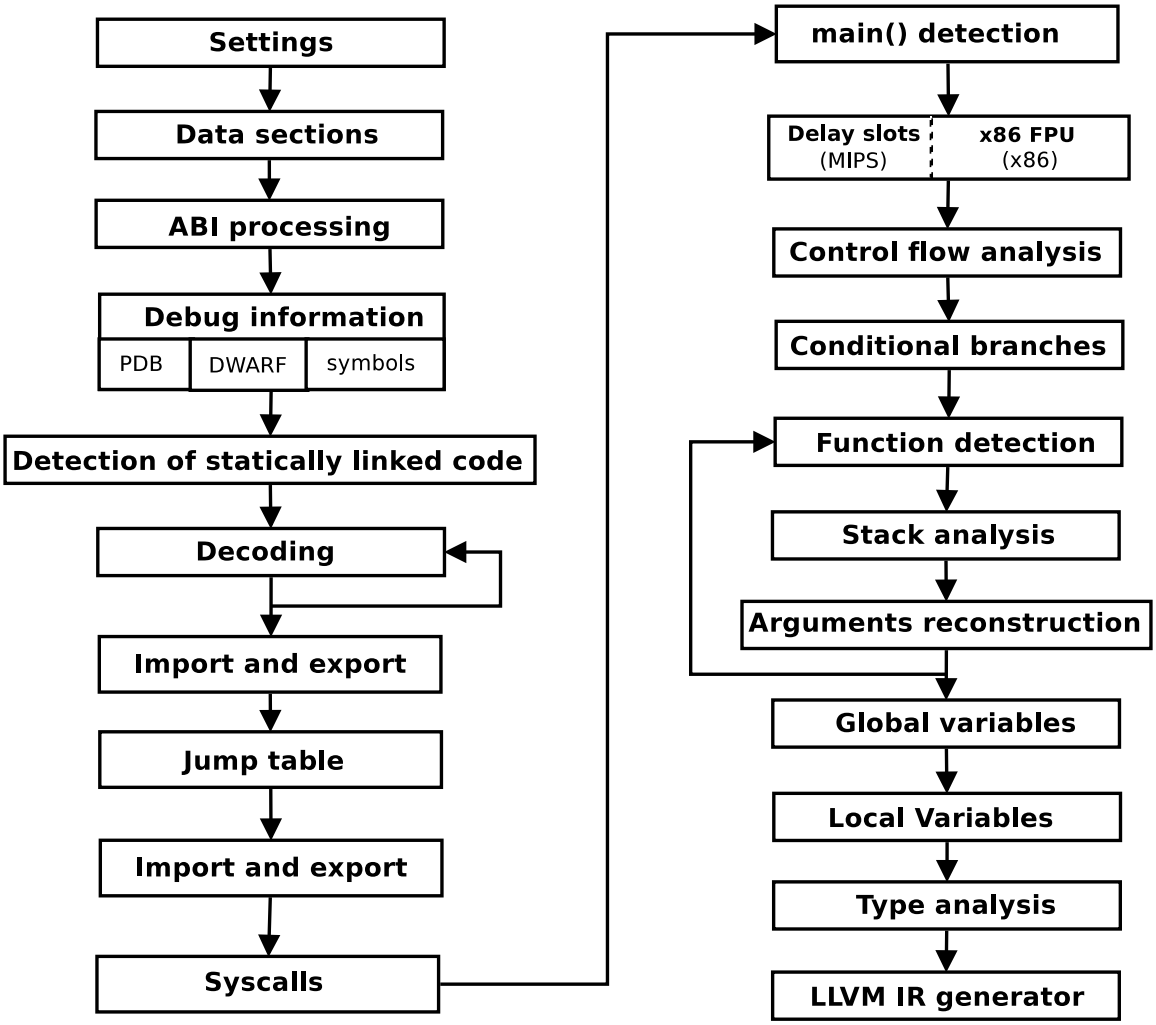


Figure 5.6: The architecture of decfront.

Firstly, detection of statically linked code is described. It covers also support tools and description of used file formats, which are utilized during the whole process. The detailed descriptions of decfront analysis follow. It is important to note that all these analyses are static.

5.5.1 Detection of Statically Linked Code

This section is based on the article [68]. The author of the thesis is the author of the toolkit for a detection of statically linked code. The inspiration for this toolkit is taken from FLIRT [23]. This detection is important for static binary analyses as a decompilation. Main aim is to eliminate such a code to save a time for a process of decompilation and also for an analysis of the results from the decompiler. The second addition is a delivery of valuable information about recognized code. We can directly mark some piece of the code as a concrete function. The significance of statically linked code removal for decompilation is also described in the article [75].

Linking of static libraries is available for all widely used platforms and compilers, therefore the process should be generic. Naturally, we cannot assume that we will recognize same version of library on different architectures by a single signature. The goal is to have a single tool for same action on libraries from different architectures and file formats. This is achieved by usage of unified object file format (CCOFF). Libraries are transformed to this format and the next tool continues by processing libraries in this format.

The signatures are not directly created from transformed libraries. From libraries we extract patterns, where each module from library is described by a single pattern. Finally, the patterns are processed into signatures and if there are conflicts, they are stored into separated files. This method allows easily to join patterns from more libraries into a single signature.

The signatures assign the name of functions for recognized code, but for the decompilation it is very important to know the type of arguments and return values. This is covered by type information. Our decompiler is based on LLVM, therefore the C types are directly transformed into LLVM types and they are used in that type information.

There are two mainly used object file formats: ELF and PE. The first tool, which touches the libraries, has to handle at least these two formats to get closer to generic purpose. The second and better option is to convert library from different formats into single common format. This is used in our solution. We use tool bintran [42], which converts object files from ELF, PE, or Mach-O formats into own CCOFF file format developed by the project Lissom. Thence, we can have tool for processing only this unified format. The library is an archive of object module files. Bintran extracts these module files from library, converts each module to the CCOFF format, and finally stores them all into own archive format.

Pattern Files

The second step is ensured by the tool *ccoff2pat*. This tool takes a converted library and generates a pattern file. The file contains a header and one pattern for each module from library. The pattern from module is taken only if it has at least 128 non-variable bits. The variable bits are used on the place of references. We do not know exact value of variable bit, because there is usually encoded an address, which is updated by the linker. According to the our experience, the lower number of non-variable bits causes too many false positives.

The header is formed by four lines, on the first line there is an identifier (magic number) for this file format `R14kdP0a7q`. Then, there is a size of byte in bits. The minimal size of instruction in bits is placed on the third line, and the last one is the number of lines with patterns (one pattern is on one line). The minimal size of instruction is important, because of usage for different architectures. For example, on the MIPS platform all instructions have same length 32 bits. This length forms minimal compare unit for signature creation and also for later searching in executables.

```
R14kdP0a7q
8
32
6
[256 bits] 04 9405 00F8 1 0000 accept 3 0024 _psp_descriptormap 0048 _errno 0078
    sceNetInetAccept [tail bits]
[256 bits] 30 F26D 0074 1 0000 atest 0 [tail bits]
[256 bits] 0C 05D8 00EC 1 0000 asprintf 4 0060 vxprintf 0078 realloc 00AC malloc
    00C0 strcpy [tail bits]
[256 bits] 1C 4487 009C 1 0000 bind 3 0018 _psp_descriptormap 003C _errno 0060
   sceNetInetBind [tail bits]
[256 bits] 0C 5AC0 008C 1 0000 chdir 3 000C _psp_path_absolute 001C sceIoDopen
   002C sceIoDclose [tail bits]
[256 bits] 0C 634C 0058 1 0000 closedir 4 0010 free 0018 sceIoDC 002C
   _psp_set_errno 0034 _errno [tail bits]
```

Figure 5.7: A preview of a pattern file.

The example is shown in Figure 5.7. Sequences of bits are replaced by `[]` blocks. This pattern file is extracted from library for MIPS architecture, where all instructions have same length 32 bits, therefore the minimal length is 32 (the second line).

The first part of pattern is 256 chars, char is one of `0`, `1`, or `.`, where dot means variable bit. These 256 bits represent the first bits of module. If the module has less than 256 bits, the missing bits are also represented by dots. Then, there is a number of bytes used for calculation of CRC code and the CRC code. This number depends on the distance of the first byte with variable bit after first 256 bits. Such a byte determines the end of code, which is used for CRC calculation. If the module is smaller than 256 bits, the CRC will be obviously `0000`. We use the CRC16 algorithm.

Behind CRC code, there is a number of public symbols of that module, for each public symbol there is its address and name. There should be always at least one public symbol. The same form is used for references, the difference is that there could be no reference, then there is just a `0` (the case of the second pattern in example). The references can be used for resolution of modules, which have same other parts (starting bits and CRC code). Because of generic approach it is much more complicated to use references. Therefore it is not applied at this moment and it is mentioned in a future research. The last part is `[tail bits]`, which contains the bits sequence after bits used for CRC code. This part can be empty. Its size is not limited, so it is filled out with all remaining bits. The content is same as it is for the first part, it consists of `0`, `1`, or `.`. The tail bits are used if there is unequal bit at same position for two modules. In that case, we store information about position and value of that bit to distinguish between modules.

Signature Files

The next step is creation of a signature file from one or more pattern files. The main reasons for this transformation are detection of conflicts and finding of common first bits. A conflict is a state, when two or more patterns have equal first bits, CRC codes, and tail bits. Such patterns are excluded from signature file into separate file, called exception file.

The tool *pat2sig* is developed for this action. It loads all input pattern files. It takes the smallest minimal instruction size as the compare unit size. Imagine that it is 32. Now, the patterns are divided into groups. The first group is derived from the first 32 bits of the first pattern. All other

patterns are tested if they have same first 32 bits, and if yes, they are included into this group. This is done recursively for all patterns until each pattern is in some group. There could be only one pattern in the group if it has unique first 32 bits. This process continues in groups with at least two patterns, there are compared next 32 bits to create subgroups. Dividing into groups is stopped when there is no more first bits or each pattern is in own group or subgroup.

The next step is detection of collisions. Now, it is quite simple step due to division of patterns into groups. If there is more than one pattern in group, they are tested for differences in CRC codes, tail bits, or references. If they cannot be distinguished, they are moved out into exception file. According to the way how the module is recognized, there are three types of signature:

- N - normal type: module is recognized by first bits or CRC code.
- T - tail type: module is recognized by tail bit (its position and value).
- R - relocation type: module is recognized by the relocation (reference).

The better precision and performance of signature is achieved by sorting of signatures before writing them into output file. The idea is based on a fact that the signatures with larger number of bytes included in CRC calculation are more accurate, so they are written firstly. If the number of such bytes is same, the second sort is done by the size of the module. The bigger modules reduce more code, which is skipped by search with another signatures.

Header of the signature file is formed by four lines. The format is very similar to pattern file header. On the first line, there is a string for identification of the file type (`Sig14sd77x`). It is followed by the number of bits in byte, the size of compare unit and the number of lines with signatures. The header is followed by lines with signatures, where, on the single line, there is at least one signature.

```
Sig14sd77x
8
32
7
[256 bits] | 1 N FF E60F 0B0C 2 0000 permute 016C getopt
[32 bits] [32 bits] [192 bits] | 1 N FF 904A 0150 1 0000 memcpy
                    [192 bits] | 1 N 64 D4BA 0084 1 0000 memmove
                    [192 bits] | 1 N B4 6347 00E0 1 0000 memset
                    [192 bits] | 1 N B4 6347 00D4 1 0000 stpncpy
                    [192 bits] | 2 T 64 706F 008C 27 1 1 0000 strcpy T 0090 35 0
                         1 0000 stpcpy
                    [192 bits] | 2 R 18 8D32 0028 000C _free_r 1 0000 free R 18
                       8D32 0028 000C _pvalloc_r 1 0000 pvalloc
          [32 bits] [192 bits] | 1 N 5C AD03 007C 1 0000 strcmp
```

Figure 5.8: A preview of signature file.

Division of patterns into groups is used for creation a tree-like format. This format decreases a memory consumption of loaded signatures, and also, it helps to get more efficient search. In Figure 5.8, the module of function `getopt` has unique first bits, so they are all written. Other modules have same 32 bits from beginning, so these bits are written only once. The indentation ensures an inclusion of modules to same group. We see that on the following 32 bits there is a difference in module of function `strcmp`, which has listed own 32 bits, and other modules have common 32 bits. All remaining modules are different in the next 32 bits, so they are listed separately.

The format of line is related to the pattern format. Firstly, there are bits, but they can be divided into groups as it was described. Then, there is a separator `|`, its effect is clearly visual. The next number is count of described modules. The following letter designates the type of the concrete signature, the valid letters are `N` – normal type, `T` – tail type, and `R` – reference type. The next two numbers are related to CRC – a count of used bytes and the CRC code. Then, there is the size of module in bytes. The last part includes public symbols: their count, and for each symbol there is its offset in module and name.

The case when the tail type is used is a little more complicated. The count of described modules can be 1 or more. The additional information about important bit in tail bits is stored after module size. There is the offset of that bit and its value. The part with public symbols is same. If there are at least two described modules, the next one is introduced by letter `T`, it has same CRC code, so this is not written again. There is only the size of this module, the different bit information and listed public symbols.

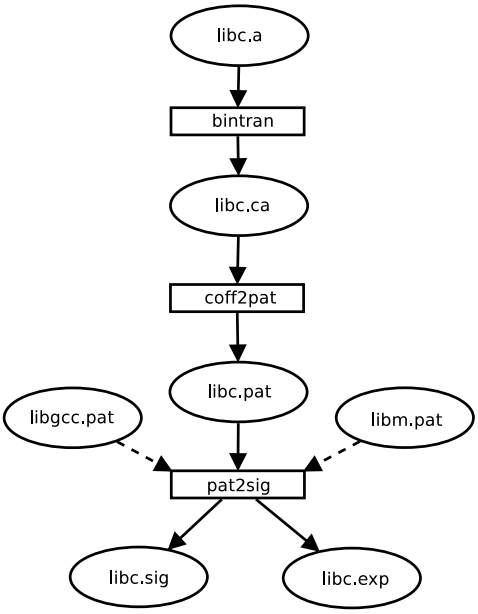


Figure 5.9: The process of signature file creation.

Whole process of the signature file creation is illustrated in Figure 5.9. The dash lines indicate an option that more pattern files can be transformed in a single signature. We remove more collisions between patterns by the transformation of more pattern files. Also, it is practical to transform patterns from more libraries of the single compiler. Than, we cover the produced code of this compiler by the single signature file.

Type Information

Type information brings additional data for recognized functions in the statically linked code. The core information are types of argument and return value. The advantage of separation of types and signatures is a possibility of usage also for functions, which are known from imports in executable files (in another words: which are not related to the statically linked code, because they are linked dynamically). The file with type information is shown in Figure 5.10.

```
TYP374INFO
2
%struct.IO_marker = type { %struct.IO_marker*, %struct.FILE*, i32 }
%struct.FILE = type { i32, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8*, i8
    *, %struct.IO_marker*, %struct.FILE*, i32, i32, i32, i16, i8, [1 x i8], i8*,
     i64, i8*, i8*, i8*, i8*, i32, i32, [40 x i8] }
10
setbuffer void 3 %struct.FILE*, i8*, i32 # void setbuffer (FILE *restrict, char
    *restrict, size_t)
fprintf i32 3 %struct.FILE*, i8*, ... # int fprintf (FILE *restrict, const char
    *restrict, ...)
printf i32 2 i8*, ... # int printf (const char *restrict, ...)
remove i32 1 i8* # int remove (const char *)
rename i32 2 i8*, i8* # int rename (const char *, const char *)
tmpfile %struct.FILE* 0  # FILE * tmpfile ()
tmpnam i8* 1 i8* # char * tmpnam (char *)
fclose i32 1 %struct.FILE* # int fclose (FILE *)
fflush i32 1 %struct.FILE* # int fflush (FILE *)
fopen %struct.FILE* 2 i8*, i8* # FILE * fopen (const char *restrict, const char
    *restrict)
```

Figure 5.10: A preview of file with library type information.

The file with type information is generated from one or more C header files. The tool is implemented as a plugin into the Clang compiler. The Clang is C front-end for LLVM and it allows to use plugins, which are able to connect to various states of C file or header file processing. The plugin is connected to part, which creates an abstract syntax tree. When the declaration of new function is parsed, the plugin adds information for this function – types of arguments and type of return value. If the type is a structure, it processes its declaration and creates a separated type information for this structure.

On the first line of file with type information there is magic string `TYP374INFO`. The file consists of two main parts – declarations of structures and declarations of functions. The number of declared structures is on the second line. Then, there is one structure declaration on each line. In our example the long declaration is divided into more lines to be able to show it. This part is followed by the number, which means the count of function declarations. And then, there is one declaration for each line. If there is used structure in structure, the declaration has to be listed for all of them. In `%struct.FILE` there is used a pointer to `%struct.IO_marker`, so its declaration is also listed.

All the types are directly transformed into LLVM types. The reason is usage in our decompiler, which has LLVM framework as its middle-end and back-end. This transformation is very straightforward due to usage of Clang. Clang as LLVM front-end supports a conversion of all C types into LLVM types. For example, `int` is converted into `i32`, or `char*` into `i8*`. Some information (e.g. sign type modifiers) are stripped, because their are not so needed in decompilation. Also, the modifier `const` is not transformed, because it is important only for the compiler.

The format for function declaration is quite simple. The first string is name of the described function. The type of return value follows. Then, there is a number of arguments and their types. Each type is divided by a comma. Behind the terminating symbol #, there is the original C declaration, which is used later by decompiler only for informative purposes.

Detection by Matching Signatures

The big amount of different libraries results into a lot of available signatures for the decompiler. The decompiler has to choose some subset of them to be applied. It is smart to sort signatures by various conditions, at least by an architecture and object file format. In the better case, we could also sort them by the concrete compiler or compiler version. This information could be included directly in the signatures, but that was rejected, because we want to avoid loading of each signature for the decision if it should be used or not.

The tool fileinfo provides the information about the architecture of executable, its file format, used compiler or packer. If the executable is packed, there is no need to use signatures, because the code is changed in many ways and it prevents finding of some library code. This tool helps the decompiler to set conditions for signature selection. For example, if an executable is for the `x86` architecture and it is the `PE` format, compiled by `Delphi`, we take only signatures for these specific parameters. It saves time and resources due to disposal of many signatures for other compilers as gcc or Microsoft Visual C.

Decompiler loads all selected signatures and sorts them by the same strategy as is used for signatures when they are written into a file. After load of the executable, the search can be started. The search is performed only on the bytes from code sections, because it is wasteful to look on data or other sections. From the loaded signatures the decompiler determines the size of minimal compare unit. Then, it reads and compares parts of code with this size. The tree-like structure is fully exploited, because we make only a single comparison between code and signatures, which have common starting part.

If there is a hit on the first bits, there is a control of CRC checksum or tail bit for tail type signature. After this all is correctly compared, the code is marked as statically linked code. Such a code is not more compared with other signatures and decompiler stores internally the addresses and the names of functions in this code. As a last step, the decompiler pairs the recognized functions with type information. This is done simply by matching the function names.

5.5.2 Overview of Front-end Analysis

The front-end part is basically responsible for translation of input platform-dependent machine-instructions into an independent code representation in the LLVM IR notation. However, it is necessary to apply several methods of static analysis, such as detection and recovery of functions and loops, data-flow and control-flow analysis. These methods are described in the following text.

Data Sections Analysis

Data section analysis manages data objects. It reads and stores the whole decompiled file, because we need access to both code and data sections. This analysis is used by other parts of the decompiler. Usually, in the case when there is a read of memory on some address. Motivation is to find out the value on that address. This analysis tries to investigate the bytes and determine the type of this part of memory and its value. This investigation can be supported by additional information provided by the caller.

Such additional information is a composite of an expected type and optionally size can be added. Without this information we try to recognize string value and if we are not successful, we mark it

as 4 bytes integer. It is possible to recognize a floating point number, but it has to be demanded by the hint. The reason is any special marks of such a number. We can compare it with string. If we find a sequence of bytes, which are printable characters and it is ended with zero, it is with a great probability a string. Such an assumption cannot be done for floating point numbers.

Each recognized data object is stored. Therefore, the next access on the memory address, which is covered by already created data object, gets this object. The advantage is the requirement to use hint from only one place and all other accesses have the recognized data object without providing of hint. We can illustrate this situation on a simple access of two different functions on same string. We have an address, which is used by functions `memcpy` and `strlen`. The second function indicates that there is a string on this address. This hint is used as a hint for data object recognition. As the result, the call of `memcpy` uses this information and it has a recognized string as an argument.

Application Binary Interface

Each architecture has its own specific application binary interface (ABI). This interface determines the way how the arguments are passed to functions, the value is returned from function, which register is used as a stack pointer and other features related to usage of registers.

We created a format for describing ABI. Each architecture, which is supported by the decompiler, has to be described in this format. An example for MIPS architecture is shown in Figure 5.11. The types are used in the LLVM IR syntax. Names and numbers of registers are given by the ADL description of the architecture. It consists of these parts (sections):

- `data` – it covers the mapping of types. The other sections are more simple due to this part. In the other sections, there can be the same rules for different types. So here the types are mapped into the common one and this one is used later.
- `stack-direction` – direction of storing arguments on the stack (available is right to left - RTL, or left to right - LTR).
- `stack` – information about stack. The register which works as the stack pointer, its value and the alignment.
- `jump` – place (stack or register), where the address for a return from function is stored.
- `return` – describes where is stored the return value from the callee.
- `passing` – describes where are stored the arguments for the callee.
- `flags` – describes the map between registers and flags (this section is optional).

Knowledge of this interface is essential for calling of linked (statically or dynamically) functions, which are linked according to the concrete architecture ABI. Also, for the reconstruction of arguments this information helps to decide if the detected place has bigger probability to be an argument, because it is typically used for this purpose. The writing into place given by section `jump` is used to detect the type of function call. If the function is called without storing the following address into this place, it is optimized version of call – *tail call*.

```
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 5.11: Description of ABI for MIPS architecture.

Exploitation of Debugging and Symbolic Information

In order to accelerate the decfront or provide more precise output, it is also possible to exploit debugging or symbolic information contained in the input executables or provided as an external file. It should be noted that debugging information is included only in a small amount of real-world executables, but we can easily generate it in our own testing executables. This part of decfront was implemented in a cooperation with Peter Matula and Jaromír Končický.

There are two common formats of debugging information—DWARF [21] and PDB [48]. DWARF is an open format. Therefore, the support is included without any major problems. PDB is a proprietary format created by Microsoft. It is more complicated to handle it and it required to create an entire tool for this task. We have previously analyzed its structure by using reverse-engineering techniques [41]. A brief overview of its structure is depicted in Figure 5.12. The information in the PDB file is divided into separated streams. Each stream contains data about a concrete scope.

Debugging information provides data about functions, their names, location, arguments, and return types. Similar information is available for global and local variables. Therefore, we are able to use the same names for functions and variables which were originally used by the developer of the decompiled executable.

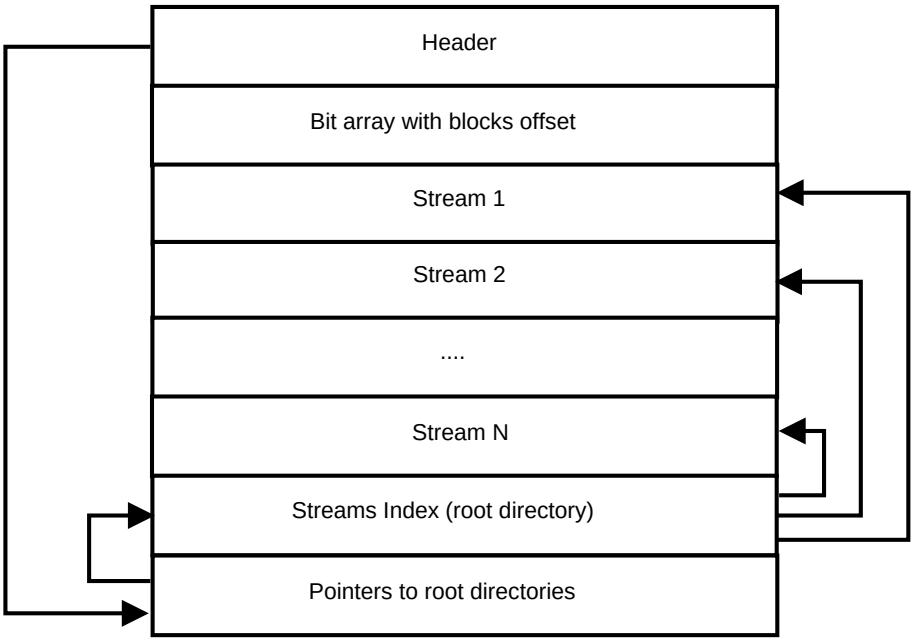


Figure 5.12: Structure of a PDB file.

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 function names and their positions in code sections. On the other hand, presence of symbolic information is more common in real-world applications.

All three types (DWARF, PDB, and symbolic information) are different, so they are unified in the common representation in the decfront. The information about each function is stored in the separated class. The robustness of available information is given by its source. If the source is just the symbolic information, the class has only the name and the start address of the function. On the other hand, if the source is PDB or DWARF, we have there complete data about arguments, return value, the end address of function, and also local variables.

Instruction Decoding

The necessary part of the translation process is *instruction decoding*. This part was implemented in a cooperation with Jakub Křoustek. It converts machine-instructions into a proper LLVM IR form. The *instruction decoder* for the particular architecture is automatically generated based on the extracted semantics and binary coding. The instruction decoder is responsible for translating architecture-specific binary machine code into an internal code representation as a sequence of low-level LLVM IR instructions (i.e., a block with several LLVM IR instructions for each input machine instruction). 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. This part has to deal with platform-specific features. For example, it has to support architectures with different endianness.

Another issue is decoding the code with variable size of different instructions. The problem can be caused by encoding the data between the instructions. This issue is related to architecture `x86`. The decoder decodes the data as instructions and this could signify the few wrongly decoded instructions

after the end of data. The decision if given bytes are code or data is equivalent to halting problem. So, we use two heuristics to avoid it. If there is the debugging information we have a proper start address of functions. This ensures that we do not have corrupted decoding on the starts of functions. The second heuristic is based on the checking the target address of jumps. If we find the address of jump inside of some instruction, we perform the decoding at this address again. We divide this instruction into two new instructions. If it is needed (the second instruction takes at least one new byte from the following one after new decoding), the following instructions are also decoded again.

Import and Export Tables

The analysis covers processing of two types of tables. The import table contains information about imported functions. These functions are imported from various dynamically linked libraries. The export table holds the public symbols, which are provided for public use. This table is usually filled in by the dynamic libraries. The user finds here a summary of functions, which are implemented by given library.

Each item of an import table is described by these fields:

- *address* – the function address
- *name* – name of the function
- *libID* – id of library. This is internal information and it points to an internal table with more information about the library
- *ord* – an ordinal number of function. Function can be recognized by its ordinal number in given library. In this case, the name can be unavailable. This usage is not recommended, because the number can be different in various versions of libraries. Nevertheless, we meet a lot of executables, which use this approach.

For usage in the decompiler, we need a name for each import. Imports without name are fixed due to knowledge of library and ordinal number. A part of decompiler is a collection of ORD files. ORD files have really simple structure. On each line there are the ordinal number and the name of function. There is a first 7 lines of file for library `ws2_32.dll`.

```
1 accept
2 bind
3 closesocket
4 connect
5 getpeername
6 getsockname
7 getsockopt
```

We have one file for one library. So we do not cover the case of different ordinals of different library version. The main reason is that we do not know a version of library that was used by an original author of decompiled binary. So we would not be able to determine which version of ORD file should be selected.

Exports are used mainly for function detection. The item looks same as for import, but libID is not used. We utilize only the address and the name. The address gives us the start address of function.

Jump Table Analysis

Jump table analysis depends directly on the import table. Usually, compilers do not call linked functions by direct jump on the address of import. They create a part of code with a lot of jumps on the imports. So, the user code jumps on the addresses in this jump table. Our motivation is to have direct calls of linked functions. This point requires to connect addresses in jump table with imports.

For ELF format the jump table is represented by `PLT` table in `.plt` section. Unfortunately, this may not be true for malware or non-standard binaries. Also, it is solved differently for other file formats. We manage to implement generic analysis. It goes through the code and it finds jumps to addresses from import table. It is important to check that it is only a jump and not proper call. We recognize this by watching if the return address is set before executing the jump. If it is not set, we take it as a jump. In the case that it is the proper call, it may be special case, when linked function is called directly by the address from import (jump table step is skipped).

Previous recognition is important, because we remove the code of jump table from decompilation result. The reason is that it is not a real user code and it would introduce messy calls of functions in output.

The following listing shows an example of jump table for architecture `x86` and file format `PE`.

```
4074a6:       90                      nop
4074a7:       90                      nop
4074a8:       ff 25 f0 b1 40 00       jmp     *0x40b1f0
4074ae:       90                      nop
4074af:       90                      nop
4074b0:       ff 25 ec b1 40 00       jmp     *0x40b1ec
4074b6:       90                      nop
4074b7:       90                      nop
4074b8:       ff 25 d8 b1 40 00       jmp     *0x40b1d8
4074be:       90                      nop
4074bf:       90                      nop
```

There are `nop` instructions to provide a better alignment of addresses. On other architecture, various instructions can be inserted there, e.g. for preparing arguments or similar. This fact makes situation more complicated. Imagine that jump on the address `4074b0` targets onto `scanf`. Now, we can call `scanf` by jump on 3 different addresses: `4074ae`, `4074af`, and `4074b0`. The analysis solves it by a search for the first previous jump. All visited instructions by this search are marked to represent the same function.

Syscalls

Syscall is a specific type of call, which provides a service of the operating system, e.g hardware access, process handling, or network communication. It is a low level of communication between the operating system and an application. The front-end transforms these syscalls into calls of well-known functions. Such a behavior is very tight with an architecture, so the analysis works a little bit different according to the architecture of a decompiled binary.

However, the call depends also on the operating system. Linux and OpenBSD have more than 300 different calls, NetBSD supports around 500 functions. According to our experience, it is enough to support the important subset of this calls, which is described by the POSIX standard. This standard includes widely used functions as `open`, `read`, `close`, or `fork`. The advantage is that operating

systems from Microsoft are also POSIX-compliant and so, we cover all commonly used operating systems by this approach.

Currently, we support syscalls on these architectures:

- `mips` – call is executed by the instruction `syscall` and the called function is determined by the value in register `v0`. The code in assembly language looks like this:

```
415110:       24020fa3        li v0,4003
415114:       0000000c        syscall
```

  When we find this instruction, we track the value in this register and make a lookup in built-in table to find the function. In this example it is `read()`[2].

- `arm` – situation is very similar to `mips`. The call is executed by the instruction `svc`. There is a difference, that the number of function is encoded in instruction code. E.g. function `unlink()` is called by:

```
160e4:       ef90000a        svc     0x0090000a
```

  It has number 10 and we can see it as the `a` in the code of instruction[3].

- `x86` – on this architecture the syscall is more complicated. It is executed as an interrupt by instruction `int 0x80`. The number of called function is in register `al`, but the structure of code is not straightforward (AT&T syntax):

```
805591f:       0f b6 c0                movzbl %al,%eax
8055922:       57                      push   %edi
8055923:       56                      push   %esi
8055924:       53                      push   %ebx
8055925:       55                      push   %ebp
8055926:       89 e7                   mov    %esp,%edi
8055928:       8b 5f 14                mov    0x14(%edi),%ebx
805592b:       8b 4f 18                mov    0x18(%edi),%ecx
805592e:       8b 57 1c                mov    0x1c(%edi),%edx
8055931:       8b 77 20                mov    0x20(%edi),%esi
8055934:       8b 6f 28                mov    0x28(%edi),%ebp
8055937:       8b 7f 24                mov    0x24(%edi),%edi
;this is jump on 8055959 (-> int)
805593a:       ff 15 40 81 05 08       call   *0x8058140
8055940:       5d                      pop    %ebp
8055941:       5b                      pop    %ebx
8055942:       5e                      pop    %esi
8055943:       5f                      pop    %edi
8055944:       3d 7c ff ff ff          cmp    $0xffffff7c,%eax
8055949:       72 0d                   jb     0x8055958
805594b:       f7 d8                   neg    %eax
805594d:       50                      push   %eax
805594e:       e8 62 f8 ff ff          call   0x80551b5
8055953:       8f 00                   popl   (%eax)
8055955:       83 c8 ff                or     $0xffffffff,%eax
8055958:       c3                      ret
8055959:       cd 80                   int    $0x80
805595b:       c3                      ret
805595c:       b0 3f                   mov    $0x3f,%al
805595e:       e9 bc ff ff ff          jmp    0x805591f
```

[2] `http://www.rdos.net/svn/tags/V9.2.5/watcom/bld/clib/h/sysmips.h`

[3] `http://lxr.free-electrons.com/source/arch/arm/include/asm/unistd.h?v=3.1;a%3Darm`

The key component for this analysis is an interpreter, which will be described later. By its help we are able to find out that the jump on the address `805595e` will continue by another jump on address `805593a` (here the data section analysis is called to find the target of call) finally to instruction `int`. So, we can track the previous set of register `al`, find the value 63 and recognize call of function `dup2()`[4].

Detection of main() Function

This feature is aimed on binaries without symbols or debug information. Such binaries are created with a tool `strip` and its usage is usual. The goal is to find out the address, where the `main` function is located. It is important for another analysis as function detection – it can start searching from this function. Also, we can avoid writing of some unwanted functions into the result, e.g. startup and exit routines that are generated by compiler. Another advantage is that we can follow a program logic easier. Without it, we can have many functions and we need to find `main` manually. It is possible to generate call graph and to estimate `main` function according to it, but it may be inaccurate and time-consuming.

Both file formats (ELF and PE) provide information about the *entry point*, but this is only a place, where the program starts its execution. The compiler puts there specific routines to prepare environment for running user code or other similar action. Each compiler solves this task by own attitude. Therefore some generic approach for analysis of the code, which is located on the entry point, to get address of `main` function is not possible.

Our approach is compiler specific. We use the information about detected compiler to select an algorithm, which is aimed on that specific compiler. One algorithm can handle compilers with same major and minor version. In a lot of cases, the compilers with different minor version can be covered by the same algorithm, but it has to be tested.

We distinguish three approaches how to describe finding out `main` function address:

- encoded in data – the address is stored in data section. It may be the same offset from the start of section or it may be found from a load on some specific place.
- jump – we know the address where should be a jump on the `main`. Then, we ensure that we have correctly decoded instruction on that address, check that it is the instruction of the jump and find out the target. Such a jump can be located by the offset of entry point or by the offset from the call of some very specific system function (which is linked dynamically, so we have its address from jump table analysis and we need to only find the call of this function).
- store to register or stack – the address may be placed as an argument for a function like `libc_start_main`. In that case, we find the call of this function, so we are able to find the instruction that does store of the address. According to the compiler we check for register or stack storage.

Some compilers are harder to describe, because various settings of compiler influence the way how the `main` is reached. An example is the MSVC compiler from Microsoft. An algorithm is built in the way try and see, because it is affected by the mode (release or debug), the state of incremental linking, and others.

[4] `http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html`

Delay Slots Handling

A *delay slot* is an instruction slot that is executed without the effects of the preceding instruction. The most common form is a single arbitrary instruction located immediately after a branch instruction, like in this example:

```
8900400:       0e24011c        jal     8900470 <printf>
8900404:       24845fe8        addiu   a0,a0,24552
```

This is, for example, used on the MIPS architecture. We need to update such code to follow the principles of other architectures without delay slots during the decompilation process.

In the ideal case, it could mean that all we need to do is to switch two instructions—branch instruction and the instruction in the delay slot, but this is not very correct, because it requires at least a change of instruction address to follow ascending attitude. The design of our decompiler allows to merge semantics of more instructions into one instruction. This technique is used for handling delay slots. The semantics of instruction in delay slot is inserted into a previous instruction of branch and its own body is leaved as empty. It is illustrated by this generated LLVM IR (variables with substring `delay_slot` were before in following instruction at address `8900404`, which is now empty):

```
;8900400        00001110001001000000000100011100        0e 24 01 1c
;JAL {35914012}
%call_8900400 = add i26 35914012, 0 ; unsigned, Signed: -31194852
%u0_8900400 = add i32 143655940, 0      ; Assign current PC
%_d_8900400 = add i32 4, 0
%add_8900400 = add i32 %u0_8900400, %_d_8900400
;Assign to return address register  @gpregs31, we filter it out
%u1_8900400 = add i32 143655940, 0      ; Assign current PC
%_h_8900400 = add i32 -268435456, 0
%and_8900400 = and i32 %u1_8900400, %_h_8900400
%conv_8900400 = zext i26 %call_8900400 to i32
%_k_8900400 = add i32 2, 0
%shl_8900400 = shl i32 %conv_8900400, %_k_8900400
%or_8900400 = or i32 %and_8900400, %shl_8900400
%call2_delay_slot_8900400 = add i16 24552, 0 ; signed, Unsigned: 24552
%u0_delay_slot_8900400 = load i32* @gpregs4
%shr_delay_slot_8900400 = sext i16 %call2_delay_slot_8900400 to i32
%add.i_delay_slot_8900400 = add i32 %u0_delay_slot_8900400, %
    shr_delay_slot_8900400
store i8* getelementptr inbounds([19 x i8]* @.str_arg_1, i64 0, i64 0), i8** %
    local_15
%arg5_8900400 = load i8** %local_15
%local_16_tmp_6 = call i32 (i8*,...)* @printf(i8* %arg5_8900400) nounwind
store i32 %local_16_tmp_6, i32* %local_16

;8900404        00100100100001000101111111101000        24 84 5f e8
;ADDIU {4}, {4}, {24552}
; nop()
```

We need to have information that an instruction is in a delay slot. That is solved in two ways—an instruction can be directly marked that it is in a delay slot, or we could have information that the architecture uses delay slots and the instruction is a branch instruction. Then, we know that the next instruction after branch instruction is in a delay slot.

x86 FPU Analysis

The floating point unit (FPU) is on architecture x86 designed with 8 working registers (`st0 - st7`), which work as a stack and state registers. There is a pointer on a register with a value from 0 to 7. Each register has size of 80 bits. Access to these registers is not direct by giving a number of specific register, but it is implemented as an offset to pointer.

This approach requires specific handling by the decompiler. If we want to know which register is touched, we have to know a value of the pointer. Pointer value is stored in the state register, therefore we are able to watch all operations with the pointer value and follow it. Such a following has an advantage that we can process each decompiled function separately.

Analysis works in three separated steps:

1. Process all instructions and find operations with registers. If the register is an FPU working or state register, we replace the current operation with a new specific operation, that will be updated in next step. There are 4 specific operations: load and store for working register, and load and store for a state register.

2. In this step, the operations created in the previous step are updated. The code is processed according to the control flow and there is set the number of register for each operation. We start at `main()` with pointer value 7. The first operation should be load from the state register and we update it by setting of value 7 in it. Following the operation of load or store into the working register, we know the current pointer value, so we add offset and update this operation with specific number of register. The pointer may be decreased by 1 and saved by store operation into the state register. This process sets the concrete numbers of working registers in all load and store operations. Therefore, we are able to generate LLVM IR with correct register numbers. The algorithm is built in a way that we can start with an arbitrary value from 0 to 7, and decompiled code is correct.

3. The last step is oriented on the returns from functions. If a function returns a floating point value, it is stored in the working register that is determined by the pointer value. We can identify this value after the second step is finished and also, we need to have complete control flow analysis to know which jumps are returns from functions.

Separation of access to state register has one more advantage. We can easily ignore it when LLVM IR code is generated. Handling of a pointer value is not a part of original source code, therefore without ignoring we would get the output with added unwanted code.

Control-Flow Analysis

Control-flow analysis (CFA) is the most important analysis of the whole decompiler and many other analysis depend on its results. It means that wrong output of this analysis will have bad impact on the output of other analysis and, therefore, on the whole result of decompilation. The aim of CFA is to divide the code into basic blocks, which are later used for, e.g., function detection or reaching definitions analysis.

Control flow is affected by branch instructions. Firstly, we need to recognize branch instructions and then mark them according to their purpose. The tricky part is that the purpose of a branch can be changed whenever we get more knowledge, i.e., branch on some address can become a tail call

when we find out that this address is an address of a function. We use the following flags for branch instructions:

- **branch** – every branch instruction is marked by this flag. If no other flag is used, it is an unconditional jump inside of a function.
- **conditional branch** – a branch that is taken only if a condition is true. Assignment of this mark is easy because these branches are specific. Indeed, it is enough to check if there is a condition associated to the branch.
- **unknown branch** – is a flag for a branch with an unknown target address. It is used for jumps based on a register or a memory place when we are not able to calculate the value of the register or memory place, and therefore, we are not able to decide the target address. This type is usually later change into another type as switch or pointer call.
- **defined function call** – indicates that a branch calls a function that is defined in the decoded code. This mark is assigned dynamically because it depends on function detection.
- **linked function call** – indicates that a branch calls an externally defined function, so its body will not be in an output of decompiler. This means that the function is dynamically or statically linked (a statically linked function is removed by static code recognition, so now it is comparable to a dynamically linked function in a question of visibility to decompiler).
- **return** – is a flag for branch, which returns from current function. The form of that branch differs a lot on different architectures. On MIPS, it is a branch by the specific register, on ARM or x86, it is a branch by a value stored on the stack.
- **tail call** – is used in a combination with a function or an external function call when we detect that the compiler made an optimization and created a call of a child function, which will return to the parent function of the current function. Shortly, a function `A` calls `B`, `B` calls `C` by a tail call, and the return in `C` will return directly to `A`.
- **pointer call** – is the flag for a branch that calls a function indirectly by a pointer. That could be, e.g., a jump by a register and we are able to find out that the address of some function is stored in the register.
- **switch** – similarly as pointer call, but we find out that the target address is taken from jump table and all target addresses are inside of the current function.

Another task of CFA is a calculation of instruction successors and predecessors. This is used intensively by an interpreter (will be described later), which has to know which way to go. The knowledge of basic blocks is a main part for this calculation. The instructions inside of 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. Zero number of successors will be used for the last basic block of a function (it is finished by the return).

Conditional Branches

Design of conditional branches is different between the architectures. We do not need to implement a demanding analysis for the MIPS architecture, because the condition is part of the encoded instruction, e.g.:

```
8900374:       1080000d        beqz    a0,89003ac
```

From this instruction we know how to build the condition. It is a compare of value in register `a0` with `0`, and if yes, the branch is taken. Without any additional updates we generate directly correct LLVM IR (truncated):

```
;BEQ {4}, $ZERO, {13}
%u0_8900374 = load i32* @grepgs4
%_c_8900374 = add i32 0, 0
%phitmp_8900374 = icmp eq i32 %u0_8900374, %_c_8900374
br i1 %phitmp_8900374, label %pc_89003ac, label %pc_8900378
```

Situation is more complicated on architectures ARM and x86, because the conditional branches use flags registers for decision if branch should be taken. Also, the names of registers for given flags are different on these architectures, therefore we use optional part of ABI description to map registers of given architecture to specific flags. For ARM it looks like:

```
section flags
    OF flagv
    SF flagn
    ZF flagz
    CF flagc
```

The names of flags are taken from architecture x86. It is evident from the listing of same section for x86:

```
section flags
    OF of
    SF sf
    ZF zf
    CF cf
```

Now, we present the analysis on the example of x86 code. We have this assembly code:

```
8048530:       83 7d 0c 00             cmpl    $0x0,0xc(%ebp)
8048534:       75 18                   jne     804854e
```

Without any deeper analysis we are able to generate the LLVM IR (for `jne`) directly according to its semantics:

```
%u0_subinst_70_8048534 = load i1* @zf0
%_d_subinst_70_8048534 = add i1 1, 0
%_b_8048534 = xor i1 %u0_subinst_70_8048534, %_d_subinst_70_8048534
br i1 %_b_8048534, label %pc_804854e, label %pc_8048536
```

If we compare it with LLVM IR for conditional branch on the MIPS architecture, we see that there is not any comparison between values, but the branch is taken according to the result of calculation with `xor` operation, where value in register `zf0` (represents zero flag ZF) is included.

The analysis has to handle 2 detached tasks:

1. Reconstruct the comparison from the equation with flag registers.
2. Find an instruction where the flags are set to reveal the operands for the comparison.

Reconstruction of the comparison is based on the creation of a postfix equation and search for such an equation in an internal map with equations for all valid comparisons. Equation from our

example belongs to comparison *not equal*. It is described in internal map as (we have there more combinations):

```
1 ZF xor
ZF 1 xor
```

This is one of simple equations, because it is defined as `ZF = 0`, but e.g. great than is defined as `ZF = 0 and SF = OF`, so the map for it looks like:

```
ZF SF OF xor or 1 xor
ZF OF SF xor or 1 xor
SF OF xor ZF or 1 xor
OF SF xor ZF or 1 xor
```

The equations in the map are built according to the semantics of instructions and there are used operators `xor` and `or`.

The last step is finding the instruction that sets the flag registers. There is used control flow graph, which is created by CFA. And finally, in this instruction we identify the variables, which are used in calculations to set the values in flag registers. From all these information we can substitute operations in conditional branch instruction to get much more nicer LLVM IR code (truncated):

```
%condFromFlags_8048534 = icmp ne i32 %u6_8048530, %u5_8048530
br i1 %condFromFlags_8048534, label %pc_804854e, label %pc_8048536
```

And the compared values are set in the previous instruction:

```
%u4_8048530 = add i8 0, 0 ; signed. Unsigned: 0
%u5_8048530 = sext i8 %u4_8048530 to i32
%u6_8048530 = load i32* %stack_var_8        ; value from stack
```

Function Detection

There are two main methods of function detection. Both methods were implemented in a cooperation with Břetislav Kábele. 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. These two methods are interconnected and they form an iterative, bidirectional function search algorithm. There is a third method, which uses debugging information, but it is a simple method. It just creates the functions on the addresses that are gained from debugging information without any additional validation.

Top-Down Detector

The top-down analysis starts with a single block containing the whole program. This block is subsequently divided into smaller blocks until there remain only detected functions. After that, the functions are divided into basic blocks. For simplicity, an assumption is used that every instruction in the program has the same width and there are no gaps between instructions. The first assumption can be reached by an abstraction and the second one by renumbering addresses of instructions. A division of blocks represents a common operation for functions and basic blocks. It needs several other operations for its work—mapping of an instruction address to a block, splitting a block into two blocks by a given address, and transformation of a set of blocks to a different set of blocks via the previous operation.

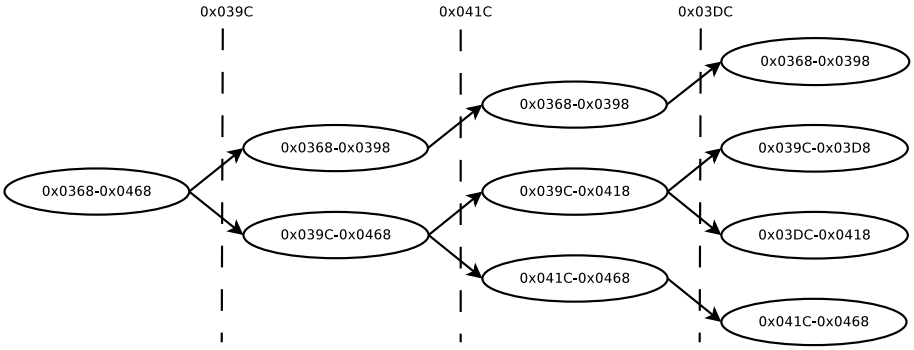


Figure 5.13: An example for the top-down analysis.

The main splitting operation works recursively. In each iteration, it picks and removes an address from a set of addresses and splits the block containing this address. The picked address becomes the first address of the block and the previous address becomes the last address of another block. The set of addresses usually contains addresses of jump instructions and a creation of this set depends on a target-architecture-specific calling convention. The definition of the split operation is architecture independent. However, its platform-independent implementation is problematic because of filling the set of addresses. Therefore, the detection by this approach must be robust and it must handle all types of call conventions and call instructions. An example is presented in Figure 5.13, the set of addresses contains `0x039C`, `0x041C`, and `0x03DC`. The first block `0x0368–0x0468` contains the whole program and in each iteration it is divided by an arbitrary address from the set. This approach is also used in the PROPAN system introduced in [35].

Bottom-Up Detector

The bottom-up analysis, introduced in [62] and tested on TriCore and PowerPC ELF executables, is an opposite of the top-down analysis. In the first step, every instruction is considered to be a block. Then, the blocks are iteratively being joined, until basic blocks are created. Finally, basic blocks are connected to form functions.

A basic operation of this analysis is joining 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 5.14.

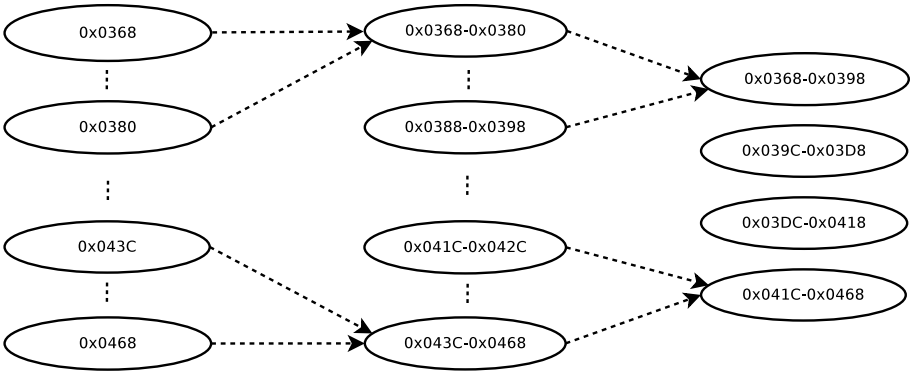


Figure 5.14: An example for the bottom-up analysis.

The most difficult phase is deciding which blocks should be joined. Similarly to the top-down analysis, we have to find all instructions which call functions. This is usually solved by heuristics dependent on the environment, which is a part of CFA. Two sets of jumps are created. The first set, J_d, is formed by decidable jumps. CFA has to assure that jumps that do not call functions are not included in this set. The second set, J_u, contains undecidable jumps. An undecidable jump is a jump with an unknown target. The ideal state is J_u being empty and the set J_d containing only jumps that are calls of functions. Unfortunately, it is not possible to always reach such a state [35].

Data-Flow Analysis

Data-flow analysis (DFA) is based on the memory places. If we have a set of registers and flag registers, R, and the set of all places for storing values in memory and stack, M, then the memory place $l \in R \cup M$ can contain a value of a variable. Every function uses memory places for input and output arguments. The DFA computes these arguments from instructions that access the stack or registers. Input arguments are stored in the analyzed function and output arguments are passed to the called function in a point of a call so not all of them are considered. After the computation, real arguments are recognized as the intersection of the input and received (output from parent) arguments. The return address is a memory place containing a return value, i.e., the value of the program counter (PC). The analysis is looking for storing the PC to a memory place and transmits this place to a proper function. The last step is a recognition of return values. We use the same principle as for function arguments.

This algorithm is advantageous due to no dependency on call conventions. Indeed, it can handle custom call conventions. An exception is statically or dynamically linked code of standard libraries, which follow call conventions of the particular architecture (described in the ABI). Therefore, the DFA utilizes this fact for functions from such libraries.

Function Arguments Recognition

A recognition of arguments and return values is made by analysing function bodies. This analysis was implemented in cooperation with Břetislav Kábele. If we know the used application binary interface (ABI), this task is much easier. Indeed, it is enough to check instructions that work with registers or the stack according to the ABI description. However, attention has to be paid for 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 the ABI use and, therefore, it is useful mainly for speeding up the recognition.

For the Intel x86 architecture, a detection by a prologue and epilogue is normally used. It is a special case of ABI detection. This method recognizes only arguments; it is not able to recognize return values. It is based on a fact that passing arguments by stack is so specific that it is easier to analyze it, but it requires to use the frame at the beginning of the function. If the frame is not used, the problem is partially solved by heuristics aimed on local variables. A modification detecting both arguments and return values was published in [5] and tested on the Intel x86 architecture.

Another method compares the number of passed values. It works on a pair caller–callee. First, memory places for both the caller and callee are detected. These places are compared, and in the case of a match, an argument is identified. An import role is played by the order of the write and read instructions. If there is a write before a read in the callee, it signals an occurrence of a local variable. An enhancement of this method is realized by a propagation of arguments through a call graph.

Arguments are propagated from functions located in leaves to the root. An advanced modification of this approach was introduced in [80] and implemented in the ITA binary translation framework which translates binary code of the IA-64 architecture.

Stack

The storage of local variables is ensured by the stack. The stack analysis aims on the accesses to memory, which belongs to stack accesses and they describe the usage of local variables. We create a local variable for each accessed offset. But, to get the offset we need to identify the load or store to memory, which is based on the value of stack pointer and value added to this one.

This analysis is run separately on each function, because local variables have validity inside of the function body. At the start, we need to know which register represents the stack pointer. This knowledge is earned from the ABI description (`section stack`). Subsequently, we seek for the loads and stores to memory, where the address depends on the value in such a register. By the current value of stack pointer and the value added to it, we resolve the offset of the local variable. Except these operations, we monitor the operations with stack pointer register. There are possible situations, when the value from the original stack pointer register is copied to another register, and in this moment, there are two valid stack pointer registers.

Interpreter

An important role is played by an interpreter of intermediate code. An essential step of decompilation is a recognition of jump targets during the static analysis. This step is not trivial for indirect jumps, where the target of a jump is stored in a specified operand. For the sake of simplicity, in what follows, we use the term jump even for call instructions.

Jump instructions can be found in all supported architectures:

- MIPS:
  ```
  jalr t9
  ```
- ARM:
  ```
  mov pc, r2
  ```
- x86:
  ```
  jmp eax
  ```

If we want to find a function which is called by an indirect jump, we have to resolve the value of the argument. The resolution is done by static interpretation of the code and tracking the value of the argument. The interpreter uses a control-flow graph 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. The processing is terminated if there are more predecessors and these predecessors differ in the modification of the tracked object. At this time, this is resolved by a calculation and usage of use-def chains.

The interpreter is also used for the calculation of the jump target. Often, it is not given by a direct number, but by the sequence of operations which calculate the address (usually left shift and addition). This is easier case, because we can do that without created control flow graph.

The usage through whole decompiler raises the requirement for a good performance, therefore the interpreter is able to cache results to save unnecessary calculations, when the same task is to be interpreted again. Moreover, it detects a dead lock when control flow forms a loop and interpretation could run forever in this case.

Global Variables Detection

This analysis has a goal to reveal the global variables and the global constants too. The difference is that the constants cannot be touched with a write operation. Similarly, as for the function detection, we are able to utilize debugging information to get exact addresses, names, and types of global variables.

A property of global variables is that they are located in the memory, usually in some section of executable. The constants use to be put e.g. in the section `.data`. We search for all write and read operations from memory (that are not stack related). We mark the addresses with flag if the address is written or read. Finally, we process these addresses and if the address is accessed only by the read operation, we create a constant, otherwise a variable. The type is derived from the following usage of read value, or how it is prepared for the write. The value from their global variable is stored in floating point register, so we know its type:

```
@glob_var_80487f8 = internal global double 0x4002a3d70a3d70a4 ; value: 2.33
```

The type can be found out very precisely, if the value is used in some linked function and we know the types of arguments. For example, it is straightforward to set type `i8*` for value, which is used as the first argument for function `printf`:

```
@.str_arg_8048780 = internal constant [11 x i8] c"mul by \25f\0A\00"
```

Local Variables Detection

One type of local variables is created by the stack analysis. But, the local variables may be allocated in the registers. Mainly, if there are used optimizations in the compilation process. In our representation, the registers have a status like a global variable. Of course, it is valid to generate LLVM IR code with usage of registers as global variables. Unfortunately, it decreases the strength of optimizations in middle-end and back-end and also, it raises the running time of these optimizations. Therefore, we have a motivation to replace usage of register with a usage of locally declared variable. The replace operation is based on the definition-use and use-definition chains.

Currently, the analysis creates local variables instead of registers in two cases:

- **function argument** – it is a use of value in register. We find all associated definitions and replace them with a local variable.
- **function return value** – it is a definition of value in register. So similarly, we replace all associated uses with a local variable.

In both cases, we know also the type, so the local variable is declared with correct type. That removes the need of conversions, because if function returns a pointer, we would need to convert it to the same type as the register has (usually an integer type). It is event more important for architectures as ARM, where one class of registers is used for both integer and floating point value and the registers are declared as integers. In that case, the function returns the `float` value, and without usage of local variable, we have to convert it to integer type and we loose the precision.

Dead Code Elimination

Some code can become useless during various analyses. For example, we can calculate the target address of a branch and store that address. Then, code for calculation of the address is not needed anymore—it is dead. This analysis can help to maintain clean code, which should contribute to faster run of all analyses. Moreover, it has great impact of allocated memory. Debugging should be also easier due to a smaller amount of code to be processed.

This analysis processes code reversely. It marks used variables and if it finds an operation which defines a not used variable, it removes the operation. In the same way, it cycles over the instruction until no operation is removed.

Type Analysis

The type analysis decides which type is used and provides a propagation of this type. The simple part of this analysis covers the following base types: `char`, `int`, `float`, `double`, `char *`, and other pointers. The complex part will cover the following composite types: arrays, structures, and unions. The complex part is a work in progress and it is not described here, because it belongs to another PhD thesis. The simple part is much easier because it can be based on types of used registers, type information about functions, or debugging information.

The idea is more or less shown in the text about local variables. On that place, we take the type directly from its usage and apply it to a local variable. This analysis implements the same idea, but it updates the types for variables created in the stack analysis and global variables analysis.

LLVM IR Generator

The LLVM IR generator is the last part of the front-end. The task of this part is a generation LLVM IR in a textual representation. It has to generate the declarations of all linked functions, global variables, and constants. The next step is producing the IR code for the decompiled executable. This code is divided into functions, which are recognized by the function detection. The generator uses a basic indentation of code for better orientation, which is very needed for debugging during the development. An example of generated LLVM IR code for a single instruction:

```
;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
```

Finally, the generator emits metadata. These metadata contains all supporting information for the subsequent parts of the decompiler—the middle-end and back-end. It includes the number of decompiled functions or the real names for variables and arguments (this is acquired from debugging information):

```
!decomp.debug.local_var_names.main.u6_8048664 = !{!7}
!7 = metadata !{metadata !"a"}

!decomp.func_count = !{!34}
!34 = metadata !{ i32 3 }
!decomp.detected_compiler = !{!35}
!35 = metadata !{ metadata !"llvm (i686-pc-linux-gnu)" }
```

After this generation, there is a release of the allocated resources and the front-end exits to allow continue of the middle-end.

5.6 Middle-end

The text of this section is based on report [70] and paper [76]. The middle-end is developed by Petr Zemek, but its description is included for the complete view of the decompiler. We move to the middle-end and present methods for this part, which is responsible for optimizing the intermediate representation that is the output of the front-end. A general view on this part is shown in Figure 5.15.

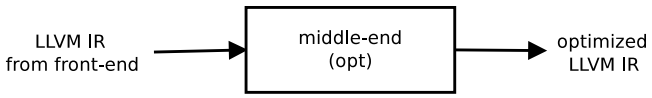


Figure 5.15: A general view on the middle-end part.

The middle-end is based on the `opt` tool[5] from the LLVM platform. This tool provides many built-in optimizations and analyses. Next, we give a brief overview of existing optimizations available in `opt`.

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 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, in many times, such optimizations are part of other optimizations that are useful for us. Therefore, we have to be careful which parts of optimizations we enable. In several cases, we will need to modify the optimizations to select precisely the modifications which are useful.

The following optimizations are (partly after modification) reused in the decompiler.

- **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

[5] `http://llvm.org/docs/CommandGuide/opt.html`

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` parameters, we may eliminate 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` parameter, 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 revert the re-construction of idioms that has been done in the front-end. 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 parameter to convert all occurrences of the `switch` instruction into a series of branches.

- **Reassociation of expressions**. The `-reassociate` parameter 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`.

- **Emission of control-flow graphs**. A control-flow graph is a representation, using notation from graph theory, of all paths that might be traversed through a program during its execution. By using the `-dot-cfg` parameter, we may print control-flow graphs of all the functions in the module. This can be utilized during debugging.

- **Emission of a call graph**. A call graph is a directed graph that represents calling relationships between functions in a program. Like with the emission of control-flow graphs, by using the `-dot-callgraph` switch, we may emit the call graph of the module.

- **Various debug prints**. The `opt` tool provides various methods of printing information that may be useful during the development. For example, when several optimizations are applied at once, it allows its user to print the module after each of them has run.

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 [50]), available in `opt` by using the `-scalarrepl` parameter. 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 makes 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.

To improve middle phase in terms of effectiveness, the LLVM IR code generated by our front-end is annotated. We utilize the fact that LLVM IR allows metadata to be attached to instructions in the program that can convey extra information about the code. The used annotations include information about application binary interface (calling conventions, system calls, etc.), names for variables, count of reconstructed funtions, and detected used compiler.

5.7 Back-end

The text of this section is based on report [70] and paper [76]. Back-end is developed mostly by Petr Zemek, but its description is included for the complete view of the decompiler. The author of thesis implemented the module for generating the output in LfD language. The goal of the back-end is to convert the input LLVM IR into the target high-level language (HLL).

A general structure of this part is displayed in Figure 5.16, and we now shortly discuss it.

Recall that by taking the analyzed requirements in [79] into account, we have decided to first convert the input LLVM IR into another IR, called BIR (*B*ack-end *I*ntermediate *R*epresentation). Then, based on the target HLL that the user of the decompiler has selected, we use an appropriate generator to generate source code in that language. The generators take BIR as their input. The generated source code is then the output of the decompiler.

In this section, we propose a more detailed design of the back-end with respect to methods that will be needed in there. We start by analyzing what parts of the back-end will be required. First, we need to read the input LLVM IR into memory. Then, we have to convert it into BIR while reconstructing high-level constructs, such as conditional statements (`if/else-if/else`) and loops (`for` and `while` loops), from the low-level constructs in LLVM IR. Apart from the actual code, we also have

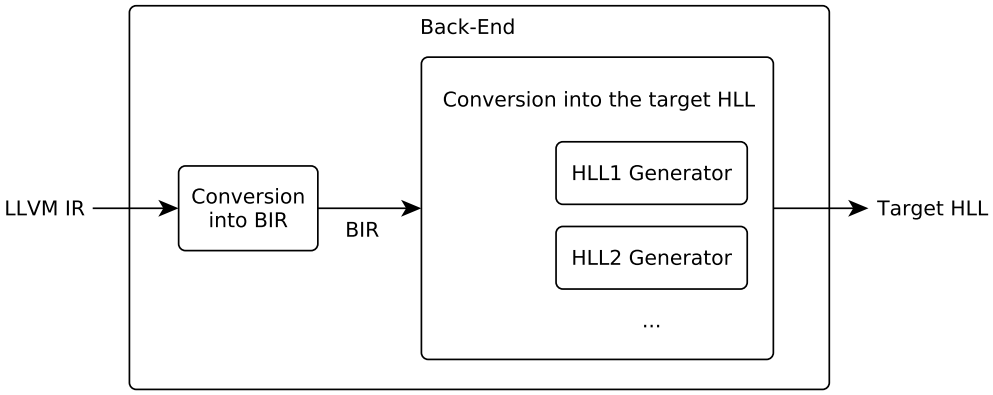


Figure 5.16: Proposed general structure of the back-end.

to obtain debugging information, which is stored as metadata in LLVM IR. After we have BIR with attached debugging information, we will require to 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 will be needed. After BIR has been optimized, we will need to rename variables to make the code more readable. In this phase, we may utilize the debugging information we have obtained earlier. Finally, we may use an appropriate generator to emit source code in the selected HLL.

Based on the analysis above, we design a detailed structure of the back-end, shown in Figure 5.17. We have included two new phases: generation of control-flow graphs (CFGs) and call graphs (CGs). The main reason of including both of these phases is that by they may considerably improve manual analysis of malware by analysts. Furthermore, as we will see, CGs and CFGs will be needed in some optimizations.

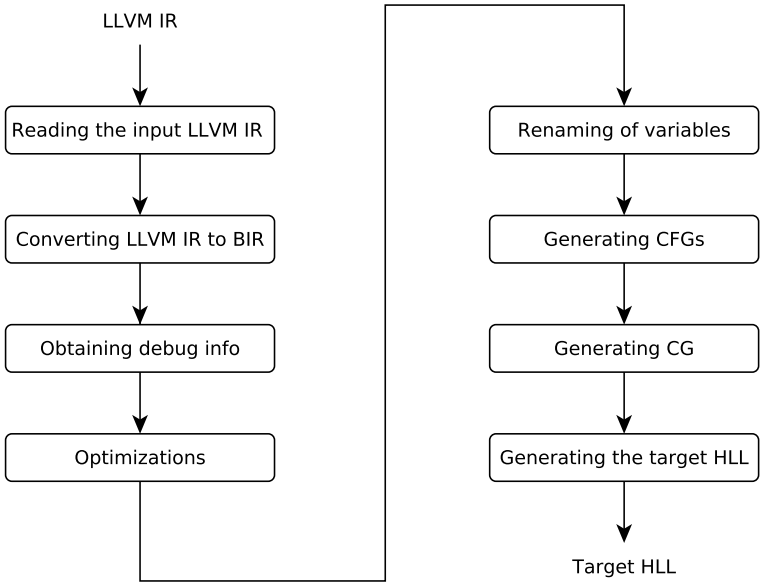


Figure 5.17: Proposed detailed structure of the back-end.

In the remainder of the present chapter, we will focus on the design of methods that will be needed in some of the eight parts of the proposed back-end's structure.

This sections includes a discussion about reconstruction of high-level constructs, like conditional statements and loops, from low-level constructs, such as jumps between basic blocks. It proposes an analysis to differentiate between signed and unsigned integral types. Subsequently, it discusses optimizations over BIR that are needed to get as readable resulting code as possible. During some optimizations, we need to know the variables that are read or modified in function calls. To improve the generated code and incorporate debugging information, renaming of variables has to be done. As the last, there is an analysis that removes redundant brackets from the emitted code.

5.7.1 Reconstruction of High-Level Constructs

As has been said in the introduction to the present chapter, LLVM IR is a fairly low-level representation, in the sense that instead of using conditional statements, like `if/else-if/else` clauses, there are only conditional jumps. However, conditional statements and loops are easier to read than a spaghetti code full of `goto` statements [25]. To this end, we have to design a method of reconstructing high-level constructs from low-level constructs.

Figure 5.18 shows several types of regions in a control-flow graph (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.

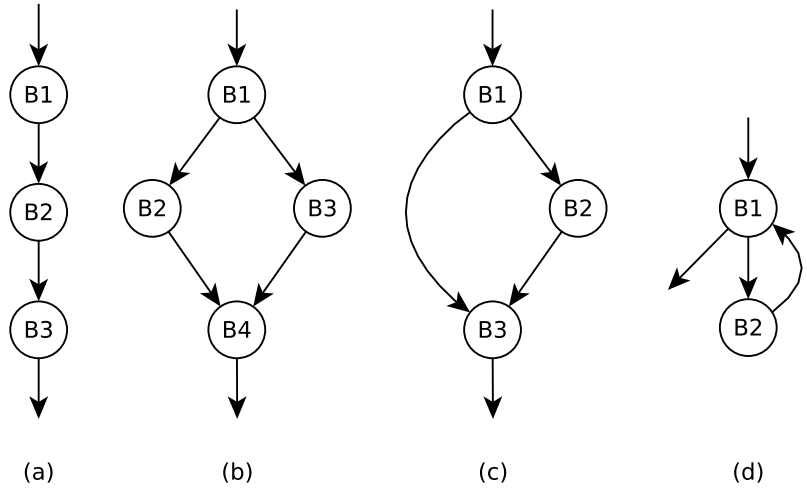


Figure 5.18: Several types of regions defining high-level constructs.

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 propose the following method. For every function, 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 5.18. 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 (see [79]). 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.

5.7.2 Analysis of Signed and Unsigned Integer Types

In LLVM IR, there is no distinction between signed and unsigned integers in terms of types—that is, there is only a single integer type, with varying bit width. However, several programming languages, like the C language, make this distinction. Sometimes, the behavior greatly depends on whether a variable is of a singed or an unsigned type. Consider the following snippet of C code:

```
char c = -1;
if (c > 0) {
    // ...
}
```

In this piece of code, the value of the condition `c > 0` depends on whether `c` is a signed character or an unsigned one[6]. Therefore, it is of major importance for us to correctly recognize the actual type of `c`.

Even though LLVM IR does not provide any distinction of signed and unsigned integers in terms of types, several instructions come in two variants—one signed and one unsigned. For example, consider the division instruction, `div`. If `sdiv` is used, then both of its operands are treated as signed numbers. On the other hand, if `udiv` is used, they are considered to be unsigned. Other ways of obtaining the information about the signess of a variable are by utilizing cast and comparison instructions.

The proposed method of finding out whether a variable is signed or unsigned consists of finding such instructions and setting an appropriate type to the variable. For example, if an integer variable is always used as an operand of "unsigned instructions", we can make it an unsigned integer. Furthermore, assignment instructions may be used to propagate the information on signess between more variables.

5.7.3 Obtaining Used Variables in Function Calls

In many optimizations, discussed later, we will need to have sets of variables which may be read or written in a function call. For example, consider the following piece of code:

```
a = 10
func()
return a
```

When we know that the variable `a` is not read or modified in the call to `func()`, we may optimize the code to

```
func()
return 10
```

[6]The C standard [24] says that whether `char` is signed or unsigned is implementation-defined, so it varies from compiler to compiler.

However, if `a` is read or modified in the call, we cannot perform this optimization because it would change the behavior of the code.

To be more specific, for every function call, we will need to have the following sets of variables:

- *Variables that may be read in the call.* These are variables which sometimes may be read but sometimes not—that is, we do not know for certain whether such variables will be read in the call.
- *Variables that may be written in the call.* This is an analogy with the previous point in terms of written-into variables.
- *Variables whose value is not changed in the call.* These variables may be read or modified in the call, but their value is never changed in the call. In other words, the call may modify the variable, but after it returns, the variable will always have the same value is it had before the call.

We implemented the following method of computing such sets of variables. First, for every function and function call in the module, we initialize all the sets of variables to the empty set. Then, we keep iterating over function bodies until none of the sets changes, and during every iteration, we do the following. We make a pass through all function bodies, and during a pass, we update the sets of variables for both the function and the calls that are inside the body. As there is a finite number of variables in the module and during every iteration, we never remove anything from a set, this algorithm always halts. After that, we have successfully computed all the needed sets of variables.

We speedup the computation by the following method. We begin by constructing a call graph (CG) of the decompiled program. In this CG, we see what functions are called in which functions, i.e. we see function dependencies. Then, we compute all strongly connected components (SCCs) in the graph. After that, we create an empty list named `order`. In this list, we will store functions in an order that will speedup the computation of used variables in function calls. We initialize the list by inserting all functions that do not call any other functions. The order in which such functions are inserted does not matter. Now, repeat (1) and (2), given next, until we have included all functions in the list.

(1) If there is a function that calls just the functions that are already in `order`, insert such a function at the end of `order`.

(2) Insert at the end of `order` all functions from an SCC that contains the greatest number of functions that are in `order`. If there is a match between several SCCs, insert the functions from the smallest one.

5.7.4 Optimizations in Back-end

Even though the main place for optimizing the code is the middle-end, there will still be a need for doing optimizations also in the back-end. As you will see shortly, the reason is that after converting LLVM IR to BIR and reconstructing high-level constructs, additional passes should be performed to improve code readability. In this section, we discuss three such optimizations.

Simplification of Arithmetical Expressions

After a direct conversion of LLVM IR into BIR and manipulation of BIR during other passes in the back-end, there may be a need for simplifying arithmetical expressions. For example, the following statement

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

may be simplified into

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

The new form is clearly more readable than the original one.

The following design of a method simplifies arithmetical expressions. We keep traversing the BIR representation of the decompiled program and try to identify constructions that can be simplified. As we have seen above, such constructions may include conversion of `n + -1` into `n - 1`. If the current module has been changed during an iteration, we perform another iteration, until the module is not changed. In this way, we simplify even expressions that need more than a single pass over BIR to be properly optimized.

Copy Propagation

The second optimization is used to reduce the number of assignments in the module. As LLVM IR resembles assembly languages, there are a lot of load and stores from memory. These operations, when converted into BIR, results into assignments into temporary variables. To make the code more readable, we should eliminate such auxiliary assignments that just copy expressions by propagating the expression through the code. For example, the following code

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

may be simplified into

```
return func()
```

First, we compute so-called *def-use* and *use-def chains* by using the standard data-flow algorithms (see [4]). They provide information on definitions and uses of variables so at every program point, we know where the variables used in the statement were lastly modified. We then use the information from both of these chains during the actual optimization, described next.

The copy propagation optimization will work as follows. First, we compute the above-mentioned chains. Then, we keep iterating over BIR and at every assignment of the form `a = expr`, we check how many next uses the variable `a` on the left-hand side of the assignment has. This can be done by using the computed chains. After that, if the replacement of `a` for `expr` in all of the uses is valid (i.e. we do not alter the behavior of the code), we can replace it. We keep traversing the BIR until the code does not change, which means that no further copy propagations can be made.

As a special case, if we find that a variable has no uses, we may eliminate it. Indeed, if a variable is never used, it has no sense of keeping it.

Conversion of Global Variables To Local Variables

The last optimization we will discuss deals with conversion of global variables into local variables. The rationale behind this optimization is that when using global registers, like many assemblers do, including LLVM, we may end up with a large number of global variables, even if some of them are used only locally. Moreover, global variables make the code harder to optimize and less readable. To this end, we propose the following method of converting global variables into local variables.

We begin by an example. The following code

```
a = 10
b = 9
c = 8

def func1():
    printf("%d\n", c)

def func2():
    global b
    b = 5
    printf("%d\n", c)
```

where `a`, `b`, and `c` are global variables, may be simplified into

```
c = 8

def func1():
    printf("%d\n", c)

def func2():
    printf("%d\n", c)
```

provided that certain conditions are met. The reason for converting `b` into a local variable is that the only function where it is used is `func2`. Furthermore, `b` in there is used only to store the value `5`. The global variable `a` has been removed completely because it is never used.

The conversion of global to local variables will work as follows. We keep iterating over BIR and during every iteration, we compute the sets of variables read and modified in every function (see Section 5.7.3). Then, based on the computed results, we check the following cases whether some of them applies. If so, them we perform an optimization.

(1) If a global variable is used only in a single function and it is not used in recursion, we may convert it into a local variable.

(2) If a global variable is used in a function just to store temporary results, we may also turn it into a local variable.

(3) If there is a global variable that is not used in any function, we may remove it.

We keep iterating over BIR until the code does not change.

5.7.5 Renaming of Variables

To make the resulting source code as readable as possible, we should give variables as meaningful names as we can. Of course, since the decompiler takes as its input a binary application where

there may be no variable names (a stripped binary file with no debugging information), this is not always possible. However, we have designed ways of renaming variables which improve the readability of the generated code. These ways are described next. However, before we delve into details, let us note that the usual way of variables naming is to use either some common prefix and an increasing sequence of numbers (`var1`, `var2`, ...) or the address of the definition of the variable (`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.

In the decompiler, we will consider the following cases when variables may be renamed.

- When the input binary was compiled with debugging information and we have successfully extracted this information in the front-end, we may use it. Then, we obtain a one-to-one correspondence between the names of variables that were in the original source code and the names of the variables in the code produced by the decompiler. This way, instead of

  ```
  def func1(var1, var2):
      var3 = var1 + var2
      return var3
  ```

  we may generate

  ```
  def add(a, b):
      sum = a + b
      return sum
  ```

  Of course, this is just a trivial example, but in real-world code, knowing the original variable names can make a huge difference.

- When there is no debugging information available or we simply have to name auxiliary variables generated by the compiler that produced the original binary file, we will produce more readable variable names: names of fruits. For example, instead of

  ```
  var1 = rand()
  var2 = var1 + rand()
  if var1 < var2:
      return var1 - var2
  ```

  we will generate

  ```
  apple = rand()
  banana = apple + rand()
  if apple < banana:
      return apple - banana
  ```

  The larger the code is, the more readable it becomes after we use such names rather than `var1`, `var2`, etc.

- Consider `for` loops. In this type of a loop, we iterate by using a so-called induction variable which is modified after the end of every iteration. Usual names of such variables are `i`, `j`, `k`, etc. By utilizing this programming habit, we may assign such names to induction variables of the reconstructed loops. For example, instead of

  ```
  for var1 in range(1, 11):
      for var2 in range(1, 11):
          var3[var1][var2] = var1 + var2
  ```

we will generate

```
for i in range(1, 11):
    for j in range(1, 11):
        banana[i][j] = i + j
```

- When there is a variable which always stores the result that is returned from a function, we will name it `result`. For example, instead of

```
var10 = rand()
if var10 < 255:
    var10 = 255
return var10
```

we will generate

```
result = rand()
if result < 255:
    result = 255
return result
```

- Finally, if the entry function of the decompiled binary takes two parameters, then according to the tradition of the C language, we name such a function `main` and its two parameters `argc` and `argv`. That is, instead of

```
def entry_func(var1, var2):
    # ...
```

where `entry_func` has been detected as the entry function of the application, we will generate

```
def main(argc, argv):
    # ...
```

5.7.6 Elimination of Redundant Brackets

As BIR is of the form of an abstract syntax tree, there is no explicit notion of brackets around expressions. Instead, brackets are emitted during the generation of the target HLL. This brings us a need to distinguish situations when brackets are necessary to be emitted and when we can omit them. For example, the following code

```
a = ((((b) + (c)) + (d)) + (e))
```

can be simplified into

```
a = b + c + d + e
```

by utilizing the priority and associativity of operators in our HLL. Of course, since our goal is to make the resulting code as readable as possible, we appreciate if there are only brackets that improve readability. To this end, we use the following method. For every target HLL, a table prescribing the priority, associativity, and commutativity of operators have to be given. Then, before emitting brackets in an expression, we check whether brackets are needed in the current situation.

To improve the readability even more, the table has to be able to force brackets in some places. For example, consider the following piece of code in our HLL, which resembles the Python programming language:

```
if a & b | c:
    # ...
```

A question that an analyst might ask is: "Is `a & b` going to be evaluated before `b | c` or vice versa?" The problem is that some operators are less commonly used so it may be not clear what is their precedence. By forcing the decompiler to emit brackets around `a & b`, the order of evaluation becomes clear:

```
if (a & b) | c:
    # ...
```

5.8 Malware Decompilation Experience

Decompiler is currently available as an online service and it is free to use at `http://decompiler.fit.vutbr.cz/`. We get a lot of feedback from whole world. Users decompiled a large number of various binaries from standard executables for Windows to specialities as firmware for MIPS routers. The result was a plenty of suggestions to improve, but fortunately, also compliments for our tool.

Online service has some limitations as maximal running time of decompilation. The fully functional tool is used in company AVG Technologies. It helps to better uncover the behavior of malware. This section introduces two analysis of malware programs, which are also published in [71] and [69].

5.8.1 Psyb0t – MIPS Malware

This section is based on paper [71]. The paper brought the bigger interest in our decompiler from security academical groups and it helped to form hackathon group on the given conference.

We present a step-by-step case study of malware decompilation by using the previously described retargetable decompiler. The target of our examination is a computer worm called *psyb0t* [7], which attacks network infrastructure devices (e.g. modems and routers) running MIPS processors with Linux-based operating systems. The following text describes all the major decompilation phases with illustrations.

Initial Recognition Using the Third-Party Tools

Our instance of psybot has MD5 hash `58f00c14942cae1e9f24b03d55cd295d` and the size of the examined binary file is 29,264 byte. This is the latest known version of this malware[7]. Tt marks itself as "`PSYB0T v2.9L`". The previous mentioned articles about psyb0t analysis were focused mainly on the older version `2.5L` [7].

The very first step of an initial analysis is detection of the file format and the target platform. The file starts with an identifier "'`0x7f'ELF`"'. In other words, it is the ELF file format [63] that is used on UNIX-based systems. Therefore, we can obtain additional information by using standard Linux tools like `readelf` or `objdump`. Relevant parts of the former one's output are shown in Figure 5.19.

[7] It should be noted that psyb0t has successors, like the Chuck Norris botnet [66].

```
ELF Header:
 Magic:   7f454c4601010100000000000000000
 Class:                             ELF32
 Data:                              2's compl.,little endian
 OS/ABI:                            UNIX - System V
 Type:                              EXEC (Executable file)
 Machine:                           MIPS R3000
 Entry point address:               0x106828
 Start of program headers:          52 (bytes into file)
 Start of section headers:          0 (bytes into file)
 Number of program headers: 2
 Number of section headers: 0
 Section header string table index: 0
```

Figure 5.19: Information about the (packed) executable file gathered by using the `readelf` utility.

According to the output, it is an executable file for the 32-bit MIPS architecture and it uses the little-endian encoding (this architecture is explicitly called MIPSel). Its entry point address (i.e. address of the first instruction executed during the application run-time) is atypical 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 comparison with Windows, the number of Linux packers is very limited (e.g. gzexe, Elfcrypt, UPX, and HASP). A detection of the used packer is difficult because the existing packer detectors (e.g. PEiD, ProtectionID, Exeinfo PE) do not support the ELF format and its packers, see [40] for details. In the classical approach, presented in [7, 20, 32], 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. A detailed description of the decompilation process follows.

Preprocessing Phase

The analyses done in the previous subsection are usually used when inspecting malware manually. We do not need any of the above-mentioned third-party software to perform such analyses. Indeed, our decompiler performs them automatically by itself so no manual intervention is needed. In a greater detail, the first part of the decompilation process begins at our file-information-gathering application called `fileinfo`. It obtains the same information as the `readelf` utility does in Figure 5.19 but independently on the used target format (i.e. it supports ELF, Windows PE and other common formats). Another its advantage is a built-in packer/compiler detector.

The detection algorithm is based on the pattern matching of the entry-point instructions with an internal signature database. The sequence of the entry-point instructions for the psyb0t malware starts at file offset `0x6828` and it contains sequence "`e00011040000f7272028a4000000e6ac0080 0d3c`"; its translation to the MIPS machine code is illustrated in Figure 5.20.

```
Address       Hex dump    MIPS instruction
-----------------------------------------
0x00006828    041100e0    bal 0x00006bac
0x0000682c    27f70000    addiu s7,ra,0
0x00006830    00a42820    add a1,a1,a0
0x00006834    ace60000    sw a2,0(a3)
0x00006838    3c0d8000    lui t5,0x8000
```

Figure 5.20: Entry-point instructions of the UPX packed code (little-endian encoding).

This sequence is matched with the internal signature for the MIPSel/ELF UPX packer[8] of the shortened little-endian form "----11040000f7272028a4000000e6ac". Symbol '-' denotes a variable part—in this case an immediate value of the conditional branch instruction `bal`.

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 [40] for details. The UPX unpacking plugin is trivial—it simply invokes the UPX packer with argument `-d`. This argument switches UPX's behavior to unpacking mode. However, this input file was manually modified to disable this form of unpacking.

The psyb0t's author wiped out (i.e. replaced by zero bytes) the four parts used by UPX to detect packed binaries. The first (file offset `0x0078`), second (`0x6803`, near the entry point), and fourth (`0x722c`) part consists of the string "`UPX!`" and are mandatory for detection by UPX. The third 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. $
```

The unpacking of such a file can be done in two ways. (1) Execute the application in a MIPS emulator and break execution after the UPX decompression routine is done and the original entry point is hit. Afterwards, dump the memory content to disk and reconstruct the ELF file by using this memory dump. Every step of this process can be done automatically without a user interaction. Retargetability can be preserved via the concept of a retargetable simulation, presented in [54]. (2) Manually patch the three missing "UPX!" strings and use UPX for unpacking.

The first method is marked as our future research but unavailable yet. Therefore, we have to manually modify the file by using the second method. This is the only manual interaction needed during the complete decompilation process of this file. The modified file can be easily unpacked via the `upx -d` command. 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 CCOFF based format. This is done by using our another plugin-based application as illustrated in Figure 5.4.

[8] `http://upx.sourceforge.net/`

The front-End Phase

Next, the unpacked psyb0t application in the CCOFF format is processed in the front-end phase. At first the instruction decoder has to be automatically generated based on the MIPS architecture model in the ISAC language. The model is relatively simple—about 4000 lines in this ADL. After that, the instruction decoder translates the MIPS machine-code instructions stored in CCOFF into LLVM IR platform-independent representation that is further processed by the following analyses.

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. This analysis is simple and just stores the pairs with the address and name of each symbol, see Table 5.2 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.

Function address	Function name
0x404c20	main
0x402da0	cgen
0x40450c	ddos
0x406f44	IrcPrivmsg
0x40eb14	rsgen
0x4156ac	rscan
0x4162cc	backup
0x41646c	spoof
0x416b98	kill_all

Table 5.2: Shortened list of functions extracted from symbols.

The analysis ensures that it checks the latency of the current instruction, and if it is the instruction followed by the delay slot, it will take the following instruction and incorporate its semantics into the current instruction. Finally, it inserts a `nop` (i.e. instruction that does nothing) instruction instead of an instruction that was in the delay slot. After this analysis, we can work with code without taking delay slots into account.

The next analysis is aimed on creating a control-flow graph (CFG). It examines all branch instructions, tries to get the target addresses and resolve the type of branches. The goal is to recognize

```
Address      MIPS instruction
-------------------------------------
0x410534:    lui     gp, 0xfc0
0x410538:    addiu   gp, gp, -30884
0x41053c:    addu    gp, gp, t9
...
0x4105c0:    lw      t9, -32268(gp)
0x4105c4:    nop
0x4105c8:    jalr    t9
```

Figure 5.21: Example of code interpretation.

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 ensured by our internal static-code interpreter, which uses a partially created CFG. It goes backwards in the CFG and searches for a store of a value in the tracked register, see [73] for details. We illustrate how the interpreter works on the piece of psyb0t code listed in Figure 5.21.

On address `0x4105c8`, there is a call of a function whose address is stored in register `t9`. Therefore, we call the interpreter to track this register and find its value. The interpreter goes backwards in the CFG and identifies the write of a value into `t9` on address `0x4105c0`. The written value is read from memory on offset `-32268` from value of the `gp` register. Next, the interpreter has to get the value of `gp`. This register is written on the beginning of the function. Therefore, it is not a problem to find it by traversing the CFG. The value is given by the following expression: `0x0fc0 << 16 - 30884 + t9`. The current function is called by register `t9`, so the interpreter uses the address of the current function in this expression. It has the value of `gp`. It subtracts `32268` from this value and the result is address of memory, where the final value is stored.

After control-flow analysis, we can detect functions. As we mentioned before, the executable under decompilation 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. The detection of functions is realized by our algorithms that were presented in [73].

Psyb0t often uses the `snprintf` function, which is used to build commands for an IRC server. This function has a variable number of arguments and it would be very eligible for us to know the accurate number of arguments and their types. This is solved by a variadic-function analysis. It takes a look on a call of such a function, and if we can get the formatting string, which is the only fixed argument, we can continue. The following arguments depend on that string and by processing the string, we find out the missing arguments. For example, given string `,,%s %s :%s``, we know that there are three more `char*` arguments.

At the end, we generate LLVM IR code, which is processed by the middle-end, described next.

The 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 5.22, 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 5.22: A block generated by the front-end for the instruction `jalr t9` on address `0x41a088`.

As described in paragraph 5.8.1, `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 5.22 after optimizations can be seen in Figure 5.23.

The back-End Phase

The back-end part takes as input optimized LLVM IR, and produces code in the specified target language (C, Python' , or LfD). More specifically, the following actions are performed:

1. The input LLVM IR is converted into BIR, which is the internal representation used throughout the back-end. During this conversion, high-level constructs, such as conditional statements or loops, are identified and reconstructed.
2. The obtained BIR is optimized by various optimizations, like conversion of global variables to local variables (when possible), constant and copy propagation, conversion of `while` loops to `for` loops, simplification of arithmetic expressions, restructuring of compound statements, etc.
3. Variables are given more readable names. When debugging information is available, we use the names from there. Otherwise, we try to rename the variables to have as readable names as possible. For example, instead of `var1`, `var2`, . . . , we name variables by fruit names.
4. If requested, the call graph or control-flow graphs are constructed and emitted.

```
%res0_41a088 = tail call i32 @usleep(i32 %arg1)
store i32 %res0_41a088, i32* @gpregs2, align 4
```

Figure 5.23: The block from Figure 5.22 after optimizations.

5. The target code in the specified language is emitted by converting BIR into a text representation in the requested language.

As a special feature, not present in other decompilers, we are able to reconstruct some symbolic names of constants passed to various functions from the standard libraries, such as `socket`. Even though the mapping of constants into their symbolic names is often implementation-defined, by using the information provided by the preprocessing phase, we were able to detect the version of the linked standard library. Therefore, we know the implementation-defined mapping of constants to their symbolic names, and we are able to utilize it to improve the readability of the generated code.

For example, consider the following call to `socket`, done by psyb0t:

```
var12 = socket(2, 3, 255);
```

The first parameter specifies the address family to be used with the socket. In the statically linked library, `2` corresponds to `PF_INET`, which is the IP protocol family. The second parameter specifies desired type of communication. For `3`, this is `SOCK_RAW`, which indicates that the communication is directly to the network protocols. The last parameter is the particular protocol to be used with the socket, which, in our case, maps to `IPPROTO_RAW` (raw IP packets). Hence, we just generate

```
var12 = socket(PF_INET, SOCK_RAW, IPPROTO_RAW);
```

Moreover, we utilize the information that we are calling a function from the standard library by assigning a more meaningful name to the variable storing the result. Since `socket` returns socket file descriptor (upon successful completion), a more appropriate name is `sock_id`. Therefore, in the very end, we generate the following piece of code:

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

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 execution of shell commands [32].

There are two known versions of psyb0t. We have decompiled the newer one, which identifies itself as `[PRIVATE] PSYB0T 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 GNU `gcc` compiler.

In this section, 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 section, 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 at the call graph. It is good for a fast detection of relations between functions. A part of that graph is shown in Figure 5.25. A complete call graph is omitted due to space constraints. The most important parts of the `main` function are listed in Figure 5.24. 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(); // Wait for commands
  //...
  fclose(fd);         // Remove mutex file and quit
  //...
}
```

Figure 5.24: Simplified code of the `main` function by using the Lissom project retargetable decompiler.

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 a 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 a 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

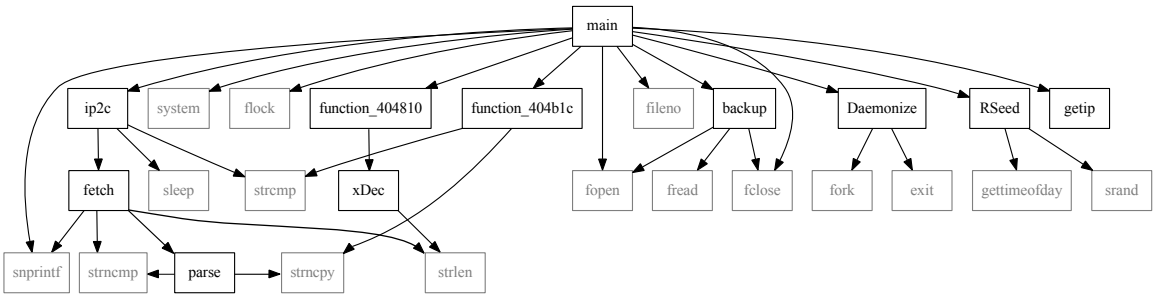


Figure 5.25: A part of the call graph for `main`. Nodes in black are user-defined functions while grey nodes denote external functions.

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;
}
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[9] (inside `function_404b1c`). The last nine symbols are generated randomly as an index to string `,,0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ``` using the previously initialised pseudo-random generator.

The data section analysis 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.

[9] `\[NIP\]-[A-Z0-9]{9}`

```
const char *STRINGS[] = {
  "mode", "login", "logout", "_exit_", "sh",
  "tlist", "kill", "killall", "silent", "getip",
  "visit", "scan", "rscan", "sleep", "sel", "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 the 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 an 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 section).

Finally, in Table 5.3, we provide some statistics about the output from the decompiler. The result in the Python' language is shorter because it does not use types. The size of both Python' and C files is quite large. In the future, we plan to improve our optimization algorithms in the back-end part to remove even more code and produce more readable output.

Feature	Value
Internal functions count	91
External functions count	57
Function calls	1278
C output size	553 kB
Python' output size	453 kB
LfD output size	15 kB

Table 5.3: Statistics about the decompiler output for psyb0t.

We have given a step-by-step case study of decompiling the psyb0t worm, targeting modems and routers with MIPS processors, by using the Lissom project's retargetable decompiler. We can conclude that by using our decompiler, we are able to speedup the analysis of malware because we deal with high-level code (cf. [20, 32], where only the output from a disassembler is used, which requires many additional analyses to be done).

5.8.2 Aidra and Darlloz – Linux Worms

This section is based on article [69] on the AVG blog. At the end of 2013, a new worm that targets small Internet-enabled devices was discovered[10]. The worm, called Linux.Darlloz[11], is capable of infecting a wide range of Internet-of-things devices, like routers, security cameras, and entertainment systems that are increasingly equipped with an Internet connection.

The Linux.Darlloz worm is interesting also from another point of view. When it is executed on an infected device, it first checks if another malicious worm, Linux.Aidra[12], runs on that device, and if so, it removes the competing worm from the device. This kind of war between malicious-software writers is not something that we see very often, and it is assumed that in the future, we may see more of these fights over the control of Internet-of-things devices.

Overview of the Samples and Initial Analysis

We have analyzed over twenty samples of the Linux.Darlloz and Linux.Aidra worms. All the samples were in the Linux ELF file format, but they were built for different architectures. More specifically, we had samples of Linux.Darlloz for the MIPS, ARM, PowerPC, and Intel x86 architectures, and samples of Linux.Aidra for MIPS, ARM, and PowerPC. We have not seen a version of this worm for Intel x86. Moreover, we had samples of Linux.Aidra for the SuperH architecture, but we did not analyze them because the decompiler does not support this architecture at the moment.

Some of the Linux.Aidra samples were compiled with debugging information, which the decompiler utilized to give functions and variables more meaningful names and types. Moreover, almost none of the samples were stripped, so we had available also symbols for functions. On the other hand, all the samples of Linux.Darlloz were stripped so no symbols or debugging information were available. We have only detected that the Linux.Darlloz binaries were built by using GCC 4.1.2. Finally, none of the binaries were packed by a packer or protector.

Analysis of Linux.Aidra

This worm is unique due to the fact that its source code is freely available. From the words of its author, it is a mass-tool commanded by IRC that allows scanning and exploiting routers to make a botnet. In addition to this, one can perform attacks with TCP flood.

Due to the openness of its source code, a highly detailed analysis is possible to be done. However, during our analysis, we have found that the binary samples we had available differ with some respect

[10] `http://www.symantec.com/connect/blogs/linux-worm-targeting-hidden-devices`

[11] `http://www.symantec.com/security_response/writeup.jsp?docid=2013-112710-1612-99`

[12] `http://www.symantec.com/security_response/writeup.jsp?docid=2013-121118-5758-99`

to the provided source code, even though the samples report the same version as the source files: lightaidra 2012. This means that the worm was modified before it was put out in the wilderness. In what follows, we explicitly point out these differences.

When the worm is started, it performs the following actions:

- calls user function `deamonize()`, which calls Linux function `fork()`. As the function name suggests, this makes the process running as a daemon.
- writes its process identifier (PID) into file `/var/run/.lightpid`. This can be seen from the following piece of code that our decompiler generated:

```
fd = fopen("/var/run/.lightpid", "a+");
if (fd != NULL) {
    v3 = getpid();
    fprintf(fd, "\%d", v3);
    v4 = fclose(fd);
}
```

As we will see later, Linux.Darlloz utilizes this file to kill his enemy. If there already runs an instance of the worm on the infected system, it is killed and replaced with a new instance. This can be utilized when a new version of the worm intrudes the device. It tries to connect to IRC servers whose addresses are encoded in the binaries. In the original sources, the list of servers is encoded by a substitution cipher.

We have discovered that the samples contained hard-coded addresses. One of such address was `94.23.X.Y:6667`. The name of the IRC channel is hard-coded in the binaries (originally `#chan`, but some samples use different names, such as `#drogs`). By calling user function `connect_to_irc()`, it connects to an IRC server under a name generated by `getrstr()`. This name has always a fixed prefix, depending on the architecture. This prefix was originally [a], [m], [s], [p], and [x] for ARM, MIPS, SuperH, PowerPC, and other architectures, respectively. However, in the samples, this prefix is fixed (for example, [PrEd0ne] or [falcon]). Finally, after the prefix, there is a sequence of 10 random characters. Sometimes, a password is present (for example, SHTDDoS), but not in all samples.

The most important function is `irc_requests()`. In there, a connection to the server is kept open by replying `PONG` to `PING` requests from the server. The commands to be performed by the worm are obtained by reading the channel topic `(TOPIC)` or receiving private messages `(PRIVMSG)`. The commands are received in function `pub_requests()`, where the received string is parsed, and then a matching function is called. For example, upon receiving PRIVMSG: `.exec`, user function `cmd_exec()` is called with a string specifying the command to be executed. This function calls Linux function `popen()`. In this way, the attacker may execute any command on the infected device.

Apart from waiting for commands, the worm tries to infect other devices. It does this by calling user function `cmd_advscan()`, which scans the given range of IP addresses. The scanning is done in a separate thread so the main process can wait for commands. More precisely, 128 threads are used for scanning.

There are two possible ways of intruding other devices:

- by *utilizing a vulnerability* of D-link routers. When a live IP address is detected, the worm tries to connect to port 80 (http) and sends the following POST request, which exploits a

vulnerability that is present in some D-link routers. If the exploit succeeds, the router returns its configuration file in an XML format. The worm parses the file to obtain a password for the root user. This password is then used when connecting to the vulnerable device through the telnet service on port 23.

- by *using a name and login* received from the IRC channel. In this case, it tries to connect to the received address directly, again by using the telnet service on port 23.

The connection through telnet is done in user function `cmd_advscan_join()`. If the login data are correct, it downloads script `getbinaries.sh` by executing

```
rm -rf /var/run/getbinaries.sh
wget -c \%s/getbinaries.sh -P /var/run && sh /var/run/getbinaries.sh &
```

where %s is substituted with the address of a remote server that hosts the script. This script then downloads binaries for all the supported architectures (MIPS, ARM, PowerPC, and SuperH) and executes them. One of them will eventually start. After that, the script erases itself to cover its tracks. Some samples have modified the above way of downloading the script, and execute the following commands instead:

```
rm -rf /var/run/getbinaries.sh
cp /usr/bin/-wget /usr/bin/wget
cp /bin/-wget /bin/wget
cp /usr/bin/wg /usr/bin/wget
cp /bin/wg /bin/wget
wget -c \%s/getbinaries.sh -P /var/run && sh /var/run/getbinaries.sh &
```

All these commands we see in the output C file as the system calls by function `system()`.

Analysis of Linux.Darlloz

The analysis of Linux.Darlloz was harder due to the fact that all the samples were stripped. This means that no symbols or debugging information were available. However, by analyzing system calls, we were able to reconstruct the names of several functions. Moreover, we have given more readable names to user-defined functions to make the results of the analysis more readable.

A simplified version of the worm's call graph can be seen in Figure 5.26. Nodes with full line are user functions and nodes with dashed line are Linux functions. In what follows, we describe all these functions in detail.

In user function `mask_as_httpd()`, the worm tries to mask itself as httpd (an HTTP server) by calling Linux function `prctl()`. This can be seen from the following piece of code:

```
prctl(PR_SET_NAME, (int32_t)"httpd", 0, 0, 0);
```

In user function `remove_aidra()`, the worm tries to detect Linux.Aidra, analyzed in the previous section, kill it, and prevent the device from being infected by Linux.Aidra again. This is done in the following several-step way. First, it loads modules for netfilter/iptables, which is a firewall typically used on Linux:

```
exec_sh_mod("ins_mod /lib/modules/uname -r" \
            "/kernel/net/ipv4/netfilter/ip_tables.ko");
exec_sh_mod("ins_mod /lib/modules/uname -r" \
            "/kernel/net/ipv4/netfilter/iptable_filter.ko");
```

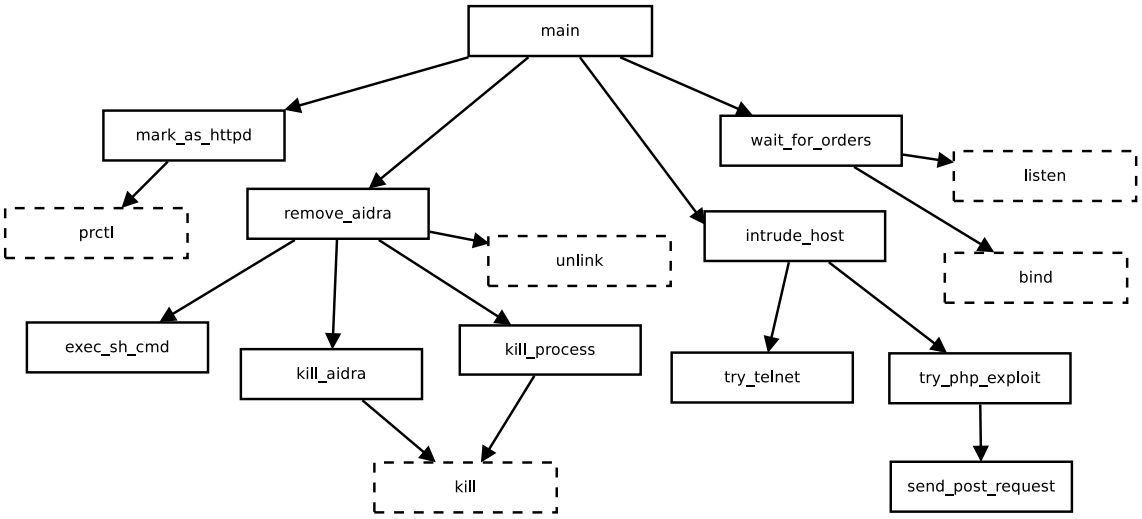


Figure 5.26: Simplified call graph of Darlloz.

The subcommand uname -r returns the version of the operating system. Then, it configures the firewall to drop packets from TCP port 23, which is the telnet service:

```
exec_sh_cmd("iptables -D INPUT -p TCP --dport 23 -j DROP");
exec_sh_cmd("iptables -A INPUT -p TCP --dport 23 -j DROP");
```

This prevents Linux.Aidra and remote users from connecting to the compromised device. After that, it tries to kill telnetd to make sure no telnet access is possible:

```
kill_process("telnetd");
```

and to kill its competing worm, Linux.Aidra:

```
kill_aidra("/var/run/.lightpid");
kill_aidra("/var/run/.aidrapid");
kill_aidra("/var/run/lightpid");
```

Finally, it tries to erase many files by calling Linux function `unlink()`:

```
unlink("/var/run/.lightscan");
unlink("/var/run/lightscan");
unlink("/var/run/mipsel");
unlink("/var/run/mips");
// ... (skipped)
unlink("/bin/wget");
unlink("/usr/bin/wget");
unlink("/usr/bin/-wget");
```

User function `exec_sh_cmd()` simply executes the given command through `/bin/sh`. In a greater detail, it calls Linux function `fork()`, and the execution of the command is performed by the created child process so the main process can continue. User function `kill_aidra()` works as follows. As a parameter, it takes a path to the file storing the PID of Linux.Aidra (see the analysis of Linux.Aidra). This is `/var/run/.lightpid`. It opens the file by calling user function `sys_open()`, reads the PID from the file by user function `sys_read()`, converts its textual representation into a number by C standard function `strtol()`, and calls Linux function `kill()` with the PID and SIGKILL as arguments. This kills the process. User function `kill_process()`

takes a single parameter, which is the name of the process. Upon calling, the function traverses all directories in `/proc`, and tries to convert their names into numbers by calling `strtol()`. In `/proc`, the system keeps the PIDs of all existing processes. Thus, the converted number represents a PID. Then, for every such directory PID, it tries to read `/proc/PID/stat`, which holds information about the process. After that, it parses the data to see if the process matches the name that was given to `kill_process()`. If this is so, then it calls Linux function `kill()`, which kills the process. The worm then starts to listen on port 58455 (hard-coded into the binaries) by using Linux functions `bind()` and `listen()`. After that, it waits for orders on that port in an infinite loop. We have also found out that first, it tries to communicate with IP addresses from the range `117.201.X.1 - 117.201.Y.254`.

For spreading it generates random IP addresses excluding some ranges. When a valid range is generated, the worm tries to intrude the remote host on the generated IP address by the following means: via telnet (port 23) and by exploiting a PHP vulnerability. In user function `try_telnet()`, if the worm successfully accesses TCP port 23. It tries some hard-coded combinations of login/password to access the host as `admin`/`admin`. When it succeeds, it executes the commands that create binary files in the ELF file format. When the above-mentioned telnet attempt fails, it tries to intrude the host by exploiting the `php-cgi` Information Disclosure Vulnerability[13] through a HTTP POST request. In user function `try_php_exploit()`, it tries to connect to the IP address and send a malicious POST request:

```
for (unsigned int i = 0; i < 5; i++) {
    sys_connect(&socket, a1);
    char *ch = cgi_bin_php[i];
    send_post_request(socket, a1, ch);
}
```

We show how the retargetable decompiler can be used for malware analysis. Its results help to get a better knowledge of malware internals and implementation.

[13] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-1823

Chapter 6

Detection of Specific Behavior

Our retargetable decompiler creates a platform that processes various types of binaries into an unified representation—C language, Python', or LfD. Due to this platform, we are able to compare binaries from different architectures on the higher level. This topic is also very extensive as the reverse engineering. Since, the development of decompiler was very demanding, this topic is not researched in such a depth.

We designed a schema to find similarity between two arbitrary binaries that can be processed by the decompiler. The idea is that we have a (malware) binary that is well-known for us and we compare it with unknown binaries to find a similar (malware) binary. Following the problematic situation in the area of Internet of Things, we could have malware for x86. This is analysed and described by the analysts, because they know this architecture quite well. And, there is a bunch of binaries for ARM, MIPS, or PowerPC that can be analysed and classified automatically as the same or very similar malware.

There is a possibility to use output to C and current tools for finding similarities for the C language. We introduce 2 tools that are aimed for revealing the plagiarism, but they are not so successful for this more specific goal. We have proposed a simplified language LfD, which is easier to analyse, therefore the similarity is found with higher precision. For this language, we have a tool LfDComparator, that is able to handle two input files in LfD language and decide the ratio of their similarity. The language LfD and tool LfDComparator are created and implemented by the author of this thesis.

6.1 C Source Analyzers

For our comparison, we use two robust tools *JPlag* developed on Karlsruhe Institute of Technology[1] and *Moss* developed on Stanford University[2]. The comparison on the source code for the malware is not a standard way, because naturally, the source code for it is not available. On the other hand, these tools solve a quite common issue if they detect similarities in source code to unveil the plagiarism.

[1] `http://jplag.ipd.kit.edu/`

[2] `http://theory.stanford.edu/~aiken/moss/`

6.1.1 Moss

Moss is an abbreviation of a Measure Of Software Similarity. It is developed from 1994. Its algorithm analyze code in many languages as C, C++, Java, Python, Javascript, Pascal, and much more. It is provided as an internet service. The authors declare that Moss is a significant improvement over other cheating detection algorithms. Therefore, we expected that it could be able to find similarities also over the output sources from the decompiler.

More details about this system are available in paper [58]. It introduces a class of local document fingerprinting algorithms. Authors developed *winnowing*, an efficient local fingerprinting algorithm. The main idea of this algorithm is defined as: In each window select the minimum hash value. If there is more than one hash with the minimum value, select the rightmost occurrence. Now save all selected hashes as the fingerprints of the document.

6.1.2 JPlag

JPlag is a system that finds similarities among multiple sets of source code files. It currently supports Java, C, C++, and some more languages. It is implemented in Java. More information about this tool is available in the paper [52].

JPlag's algorithm computes similarity in two phases:

1. All programs are parsed and converted to token strings.
2. These tokens are compared in pairs for determining the similarity of each pair. The used method is *Greedy String Tiling*. During each comparison, JPlag attempts to cover one token (string) with substrings (tiles) taken from the other as well as possible. The similarity value is given by the percentage of token strings that can be covered.

Naturally, this method starts to fail if there is used `for` cycle instead of `while`, or a statement `switch` instead of `if` sequence. For bigger change to be successful, we should avoid these issues by design of simple language—LfD.

6.2 Language LfD

LfD is an abbreviation for Language for Decompilation. It is designed to be really simple. It persists only the most important information, which are call graphs, constants and a part of control flow – loops. It is easy to extend, so it is widely open for the future research. For processing the files in LfD, the ANTLR[3] framework is used. The language is described by a context free grammar, which is expressed using Extended Backus-Naur Form[4], see Appendix A for the complete listing of the grammar. This form is required by the ANTLR framework.

The output code is quite austere, but the objection is aiming for the better similarity detection. There are three main parts:

- functions – the code is inside their bodies. The arguments are not defined, because they are used only if they are constant.

[3] `http://www.antlr.org/`

[4] `http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf`

- loops – are represented by the keyword `LOOP`.
- constants – can be strings, integers, or floating point numbers.

An example of output is shown in Figure 6.1. There are visible string constants and also, the control flow and call graphs with functions and loops is well understandable. This is a base for the following comparison of such a programs.

```
factorize() {
    printf("Prime factors of %d: ");
    LOOP {
        printf("%d x ");
    }
    printf("%d\n",);
}

main() {
    scanf("%d");
    factorize();
    LOOP {
        scanf("%d");
        factorize();
    }
}
```

Figure 6.1: An example with the program in LfD.

6.3 Tool LfDComparator

This tool compares 2 inputs in the the LfD language and decides their similarity. The output is given by a number between 0 and 100 – it means a percentage of the inputs similarity. LfDComparator is developed in Java and it uses the ANTLR framework. LfD is described by an ANTLR grammar, so we are able to use ANTLR framework to generate a lexical analyzer and parser for this language. LfdComparator uses these generated parts and it is built over them.

For deciding the similarity, the LfDComparator compares 2 areas:

- *control flow and call graphs* – the tool travels the call graph in both inputs and compares the loops, function calls and their arguments. There is important if the `main` function is present. If yes, the process is easier – the tool starts from this function. If there is not this function, the heuristics tries to find the functions that could be `main`. It is done by the search with an idea as a depth-first search. We want to find the functions that have the most deep call graph. If there are more such functions, we do more comparisons and the result with the highest score is taken.

  There are 2 types of functions: defined (there is their definition in the source) and linked (they are just called – such as `printf`). According to this type, the different action is executed for the processed function call. If the linked function is called, we compare the names in the both inputs, if there are the same linked functions (or very similar functions – this is going to be explained later). If there are defined functions, we make a comparison of their bodies

or if they were compared earlier, we use the stored result of their comparison. So for the defined functions, there is no impact of their names. This is logical, because the names can vary without any influence on the behavior.

- *constants* – During the traversal of both inputs the tool saves all constants and the functions that use them. After the comparison described in the previous point is finished, these saved constants are compared. It is a simple comparison of the value, but we again take in an account the very similar functions and we allow some special differences in the string literals. Therefore, the combinations of a constant and function given by these 2 calls are matched successfully:

```
puts("Catch me if you can")

printf("Catch me if you can\n")
```

The tricky part of the control flow comparison is the loop. There can be 3 different situations that are detected and taken as similar:

- no loop – this is usually caused by the compiler optimizations. The compiler finds a loop in the original source code. It also finds out that the number of loop iterations is constant and it is better to generate the body of the loop more times (equal to the constant).
- usage of loop – compiler generates the code as it is described in the source code. It may be the same code as in the previous case, but without usage of compiler optimizations, the loop is not removed.
- partial usage of loop – compiler can detect a part of loop as loop invariant. Such a part can be moved out of the loop if the optimizations are used.

These situations are not solved by the decompiler, because it does not try to optimize the output in this way. It creates the loop only if there is a branch in the code. Therefore, these special situations are used here. Moreover, there are more important for the comparison of the behavior similarity than for well understanding of the behavior itself.

For the comparison of the linked function calls we mention very similar functions. LfDComparator has internal database with functions that are considered as very similar from the view of behavior. The good example is a pair `printf` and `puts`. These pairs are not taken as 100% same. Therefore the sources, where the only difference is usage of very similar functions, produce the result a near under 100%.

As the last point, we can take a look at two LfD sources that are evaluated by LfdComparator as the almost same (the result is 97%):

```
compute() {
    printf("Calculation of %d: ");
    LOOP {
        printf("%d");
    }
}

main() {
    scanf("%d");
    compute();
    printf("Done\n");
}
```

(a) The first LfD source.

```
function_809324() {
    fprintf("Calculation of %d: ");
    fprintf("%d");
    fprintf("%d");
    fprintf("%d");
    fprintf("%d");
}

main() {
    scanf("%d");
    function_809324();
    puts("Done");
}
```

(b) The second LfD source.

Figure 6.2: Example of two LfD sources that have the similarity of 97%.

Chapter 7

Results

In this chapter, we present two kinds of results. The first result consists of a standard C output from the decompiler, where the output C sources are compared to the original input sources. The second result shows how the individual tools are effective in detection of similarity.

7.1 C Outputs

The decompiler was extensively tested. There are integration and night tests. During the night tests, decompiler is currently tested by 70118 different decompilations. Only in 83 cases, we generate invalid LLVM IR code, so the C output is not created. There is a binary suite for each supported architecture with pre-compiled binaries. The second part of night tests is composed of C source codes, which are compiled before the test, and the created binaries are then decompiled. Each source code is compiled for each architecture with different optimization level (`-O0`, `-O1`, `-O2`), file format (`pe`, `elf`), and other configurations (with or without debug symbols, strip). For some architecture, we also use more types of compilers (`gcc`, `clang`).

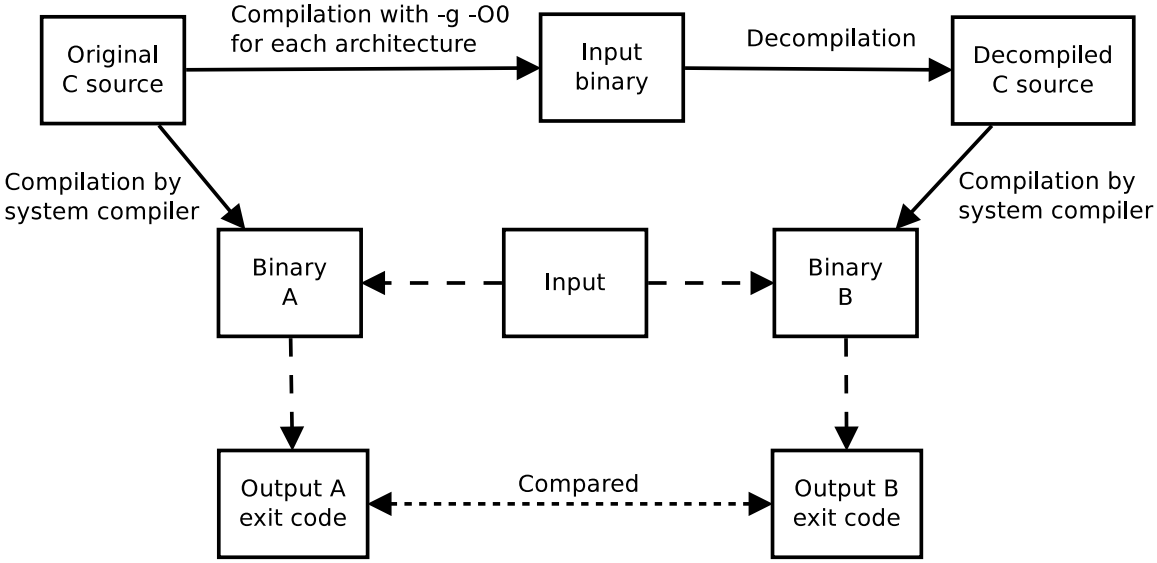


Figure 7.1: The integration testing schema.

Integration tests serve as a fast check of new changes in the decompiler source code. After each commit, the decompiler is built and these tests are run. Currently, there are 24 source code files for this test. Each file is compiled for each architecture with debug symbols and with the optimization level `-O0`. Then, the file is decompiled to C output. The aim of the decompiler is to produce compilable files, what is important for integration testing. Now, we compile the original source code into executable A, the decompiled source code into executable B. Both executables are executed with the same inputs. The outputs and exit codes are compared. If there is a change to worse results as it was in the previous run of integration tests, the author of a commit is immediately notified by e-mail. The schema of test is shown in Figure 7.1.

There are the examples of C outputs in Figure 7.2. We show output for the `MIPS` architecture with debug symbols and without optimizations. This code is very close to the original code, because we have available all important information due to compilation with debug symbols and binary code is not tweaked by the optimizations. Therefore, we can use same names for arguments and variables as it is in the original source code.

```c
int ack(int m, int 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[]) {
  int res = 0, x = 0, y = 0;

  scanf("%d %d",&x,&y);
  res = ack(x ,y);
  printf("ackerman( %d , %d ) = %d\n
      ",x,y,res);

  return res;
}
```

(a) Original source code.

```c
int32_t ack(int32_t m, int32_t n) {
    if (m == 0) {
        return n + 1;
    }
    int32_t v1 = m - 1;
    int32_t v2;
    if (n == 0) {
        v2 = ack(v1, 1);
    } else {
        v2 = ack(v1, ack(m, n - 1));
    }
    return v2;
}

int main(int argc, char **argv) {
    int32_t res = 0; // bp-32
    int32_t x = 0; // bp-28
    int32_t y = 0; // bp-24
    scanf("%d %d", &x, &y);
    int32_t v1 = ack(x, y);
    res = v1;
    printf("ackerman( %d , %d ) = %d
        \n", x, y, v1);
    return res;
}
```

(b) Output from decompiler.

Figure 7.2: Comparison for `MIPS` - `ELF` with debug symbols and optimization `-O0`.

In Figure 7.3, there is an example with calculation of the greatest common divider (gcd). The compilation is done without debug symbols (strip is not applied, so there is symbolic information). Therefore, the function `gcd` has the same name as in the original source code. But the arguments are recovered by the data-flow analysis of the decompiler and the names for them are generated. We see that the structure of algorithm is a little bit different, but this is normal phenomenon for usage of optimization (`-O1`).

Also, notice that the names of all local variables are generated. There is a difference, that decompiler does not use the local variable for original variable `res`, but directly put the call of `gcd` as the argument of `printf`.

```
int gcd(int a, int b)
{
  while( 1 )
  {
    a = a % b;
    if (a == 0)
    {
      return b;
    }
    b = b % a;

    if (b == 0)
    {
      return a;
    }
  }
}

int main(int argc,char *argv[])
{
  int a = 0, b = 0;
  scanf("%d %d", &a, &b);

  int res = gcd(a, b);
  printf("gcd %d %d = %d\n", a, b,
      res);
  return 0;
}
```

(a) Original source code.

```
int32_t gcd(int32_t a1, int32_t a2)
    {
    int32_t v1;
    while (true) {
        v1 = a1 % a2;
        if (v1 == 0) {
            return a2;
        }
        int32_t v2 = a2 % v1;
        if (v2 == 0) {
            break;
        }
        a2 = v2;
        a1 = v1;
    }

    return v1;
}

int main(int a1, char **a2) {
    int32_t v1 = 0; // bp-12
    int32_t v2 = 0; // bp-16
    scanf("%d %d", &v1, &v2);
    printf("gcd %d %d = %d\n", v1,
        v2, gcd(v1, v2));
    return 0;
}
```

(b) Output from decompiler.

Figure 7.3: Comparison for `ARM` - `ELF` without debug symbols and optimization `-O1`.

The last example shows the Fibonacci function in Figure 7.4. This case is for `x86` and `PE` file format. There is used compilation with `strip`, so all symbols are removed from the binary. Therefore, the function `function_401560` is recovered by the function detection. Hint for that is the name of function, which is now not preserved from the original source code, but it is generated according to address, where the function is recovered. The usage of advanced optimizations `-O2` results in usage of `while` loop in the decompiled source code, and also instead of two recursive calls, there is only the single one.

There is a difference in `main` function, where the loop `while` is used instead of original `for` loop. This is not a mistake, but it is caused by the fact that `for` loop is harder to reconstruct.

More results can be obtained on the decompiler website `http://decompiler.fit.vutbr.cz/decompilation/`, where are also prepared simple C source codes, but the users can load own C sources or binaries.

```c
unsigned fib(int x)
{
  if (x > 2)
  {
    return fib(x - 1) + fib(x - 2);
  }
  else
  {
    return 1;
  }
}


int main(int argc, char *argv[])
{
    int i, numtimes = 0, number = 0;
    unsigned value;

    printf("Input number of
        iterations: ");
    scanf ("%d", &numtimes);
    for (i = 1; i <= numtimes; i++)
    {
        printf ("Input number: ");
        scanf ("%d", &number);
        value = fib(number);
        printf("fibonacci(%d) = %u\n
            ", number, value);
    }
    return 0;
}
```

(a) Original source code.

```c
int32_t function_401560(int32_t a1)
    {
    if (a1 < 3) {
        return 1;
    }
    int32_t v1 = function_401560(a1
        - 1);
    while (a1 > 4) {
        a1 -= 2;
        v1 += function_401560(a1 -
            1);
    }
    return v1 + 1;
}

int main(int a1, char **a2) {
    int32_t v1 = 0; // bp-24
    int32_t v2 = 0; // bp-20
    printf("Input number of
        iterations: ");
    scanf("%d", &v1);
    if (v1 < 1) {
        return 0;
    }
    int32_t v3 = 1;
    printf("Input number: ");
    scanf("%d", &v2);
    printf("fibonacci(%d) = %u\n",
        v2, function_401560(v2));
    while (v1 >= v3 + 1) {
        v3++;
        printf("Input number: ");
        scanf("%d", &v2);
        printf("fibonacci(%d) = %u\n
            ", v2, function_401560(
            v2));
    }
    return 0;
}
```

(b) Output from decompiler.

Figure 7.4: Comparison for `x86 - PE` with strip and optimization `-O2`.

7.2 Detection of Similarity

In this part, we compare the results of JPlag, Moss, and LfDComparator in detection of similarity. Firstly, we test the detection on the binaries from different compilers. We have 10 testing C files (students projects, tool cat, and similar). We produce 30 different binaries for each file, they are a combination of these options:

- architecture – MIPS, ARM, x86
- file format – ELF, PE (PE is not generated in combination with MIPS)
- optimization – O0, O2

- additional information – debugging information (DWARF), symbols, stripped

Then, the decompiler is used to decompile all these binary files into C files. We have C files that are tested for similarity by JPlag and Moss. By this test, we simulate a generic detection on binaries from different architectures and file formats.

Each pair from these 30 C files is processed and its similarity is recorded. The number of comparisons is given by $\binom{30}{2} = 435$. We have 10 different inputs, so each tool produces 4350 results. In ideal case, we should be close to 100% similarity in the all comparisons. We decide to merge results in the 10 groups (step by 10%) for better presentation.

The results of Moss are presented in Figure 7.5. Moss provides the results as the 2 numbers for each compared pair, when at least some small similarity is found. These 2 numbers mean the ratio of similar part. The numbers can be lightly different according to the size of result, so we take average as the single result for the pair. We sort given results in the 10 groups. All unlisted pairs are put in the group `0-10%`. Overall results are really bad. This tool would not be usable for similarity detection.

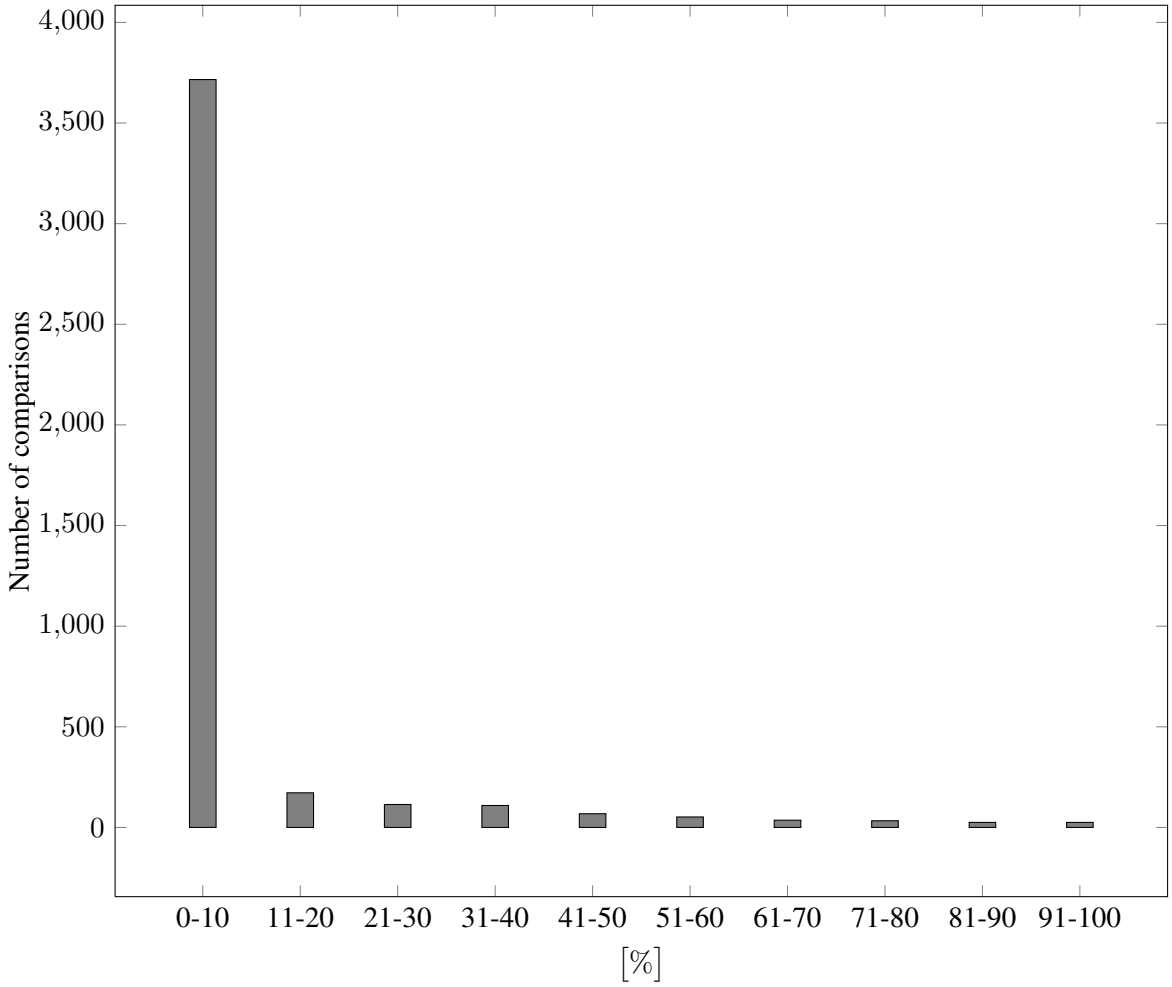


Figure 7.5: Results of comparison from Moss.

Figure 7.6 presents the results of JPlag. We see that the majority of results is below 30%, what is not very good result, but it is better than Moss. There is a little bit more detections over 70%.

Such a number can be considered as very good and it is a sign that the compared files have quite similar behavior. These detections of larger similarity are reached if debugging information is present. Unfortunately, this is not usual for real world binaries or malware. Overall, JPlag is also inappropriate tool for deciding the similarity of decompiled results.

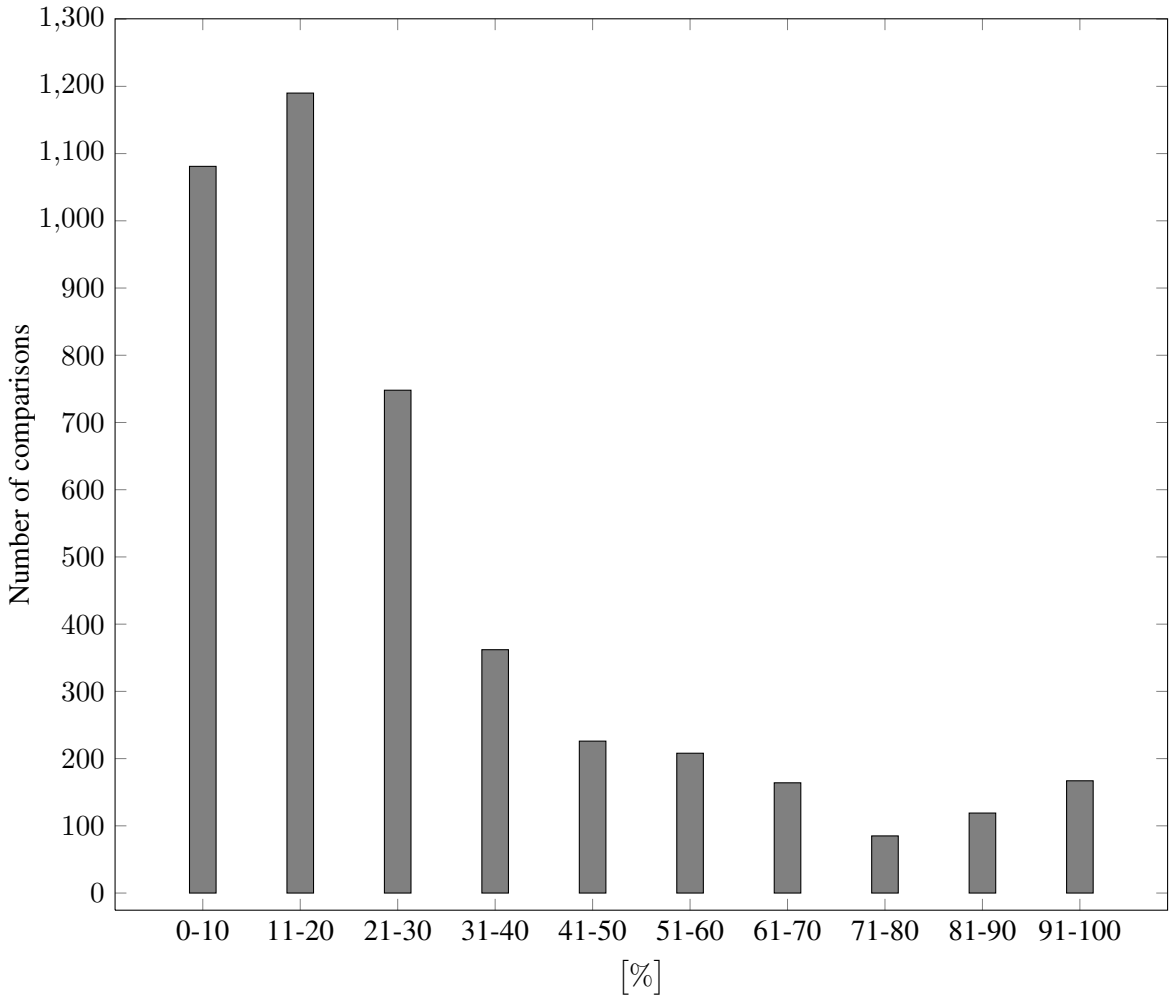


Figure 7.6: Results of comparison from JPlag.

LfDComparator achieves better results as the previous tools, see Figure 7.7. The reason is usage of specialized language LfD 6.2 and also a more specific aim of this tool. We still have some cases, where the results are bellow 40%. These cases could be improved by enhancements of the decompiler and LfDComparator.

For this kind of testing, it is important to not create false positives. LfdComparator has a good result also for this. We take one file in LfD for each tested file, so we have 10 different files to test. Except one comparison, where the result is 24%, all other comparisons are below 10%.

Finally, we test the LfdComparator with real malware. There are used previously described malware Aidra and Darlloz (see 5.8.2), because they target more architectures. So we examine real cases with malware from different architectures, what is a suitable test for verifying of generic comparison. The result is shown in Table 7.1.

The test was run on 20 binaries:

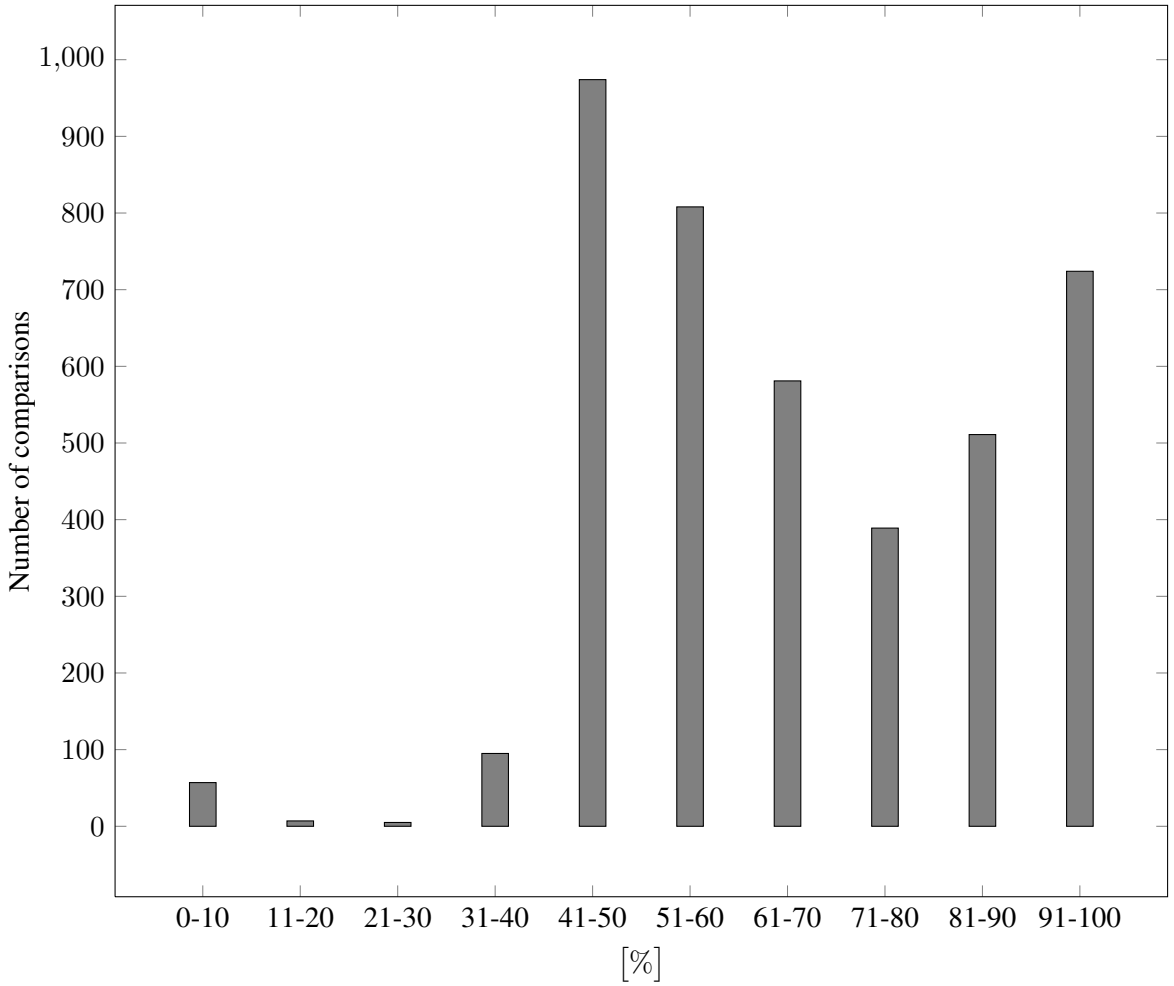


Figure 7.7: Results of comparison from LfDComparator.

- Aidra (9 binaries)
  - ARM (3)
  - MIPS (5)
  - PowerPC (1)
- Darlloz (11 binaries)
  - ARM (2)
  - MIPS (5)
  - PowerPC (2)
  - x86 (2)

The higher percentage of similarity is obtained for the binaries of same malware. In the left top part, there are the numbers for the Aidra binaries. The similarity of behaviour is almost 100% for pair ARM – PowerPC. The same result is get for some mips binaries, but there is not so great success, because it is the same architecture.

The results for Darlloz binaries are located in the right bottom part. The result of 99% is achieved only for binaries from PowerPC, but there are still the results over 90% between binaries from arm and x86. Also, positive fact is avoiding of false positives – the similarity of Aidra and Darlloz binaries is in all cases under 5%. Overall, the test with these two malware families can be considered as successful.

	aidra-arm-88b36c	aidra-arm-382c5c	aidra-arm-a3abee	aidra-mips-3ebb92	aidra-mips-91ac17	aidra-mips-239bc7	aidra-mips-acf08a	aidra-mips-c035ac	aidra-ppc-f895a9	darlloz-arm-8a5ccb	darlloz-arm-981989	darlloz-mips-1d0ffd	darlloz-mips-5ef7ac	darlloz-mips-9d9c01	darlloz-mips-19911c	darlloz-mips-b02d28	darlloz-ppc-304011	darlloz-ppc-b61b85	darlloz-x86-00a229	darlloz-x86-5b4321
88b36c	–	88	76	76	86	76	76	88	99	0	0	0	0	0	0	1	1	0	0	0
382c5c	88	–	99	78	78	77	78	78	99	3	0	1	2	2	2	3	4	2	2	2
a3abee	76	99	–	78	78	77	78	78	99	0	0	0	0	0	0	0	1	1	0	0
3ebb92	76	78	78	–	76	99	77	99	70	0	0	0	0	0	0	0	1	1	0	0
91ac17	86	78	78	76	–	76	75	76	70	0	0	0	0	0	0	0	1	1	0	0
239bc7	76	77	77	99	76	–	99	99	70	1	1	1	1	1	2	1	1	1	1	1
acf08a	76	78	78	77	75	99	–	99	79	3	3	1	1	1	1	3	3	2	2	2
c035ac	88	78	78	99	76	99	99	–	70	3	3	0	0	0	0	0	3	3	2	2
f895a9	99	99	99	70	70	70	79	70	–	2	3	3	0	1	0	1	1	2	2	2
8a5ccb	0	3	0	0	0	1	3	3	2	–	91	71	72	70	71	71	81	82	94	93
981989	0	0	0	0	0	1	3	3	3	91	–	71	72	71	72	71	81	82	94	95
1d0ffd	0	1	0	0	0	1	1	0	3	71	71	–	74	78	74	78	72	71	72	71
5ef7ac	0	2	0	0	0	1	1	0	0	72	72	74	–	74	76	74	72	71	71	71
9d9c01	0	2	0	0	0	1	1	0	1	70	71	78	74	–	74	78	71	71	72	71
19911c	0	2	0	0	0	2	1	0	0	71	72	74	76	74	–	74	70	71	70	70
b02d28	1	3	0	0	0	1	3	0	1	71	71	78	74	78	74	–	71	72	73	72
304011	1	4	1	1	1	1	3	3	1	81	81	72	72	71	70	71	–	99	76	76
b61b85	0	2	1	1	1	1	2	3	2	82	82	71	71	71	71	72	99	–	74	76
00a229	0	2	0	0	0	1	2	2	2	94	94	72	71	72	70	73	76	74	–	97
5b4321	0	2	0	0	0	1	2	2	2	93	95	71	71	71	70	72	76	75	97	–

Table 7.1: Results of comparison between malware binaries.

The disadvantage is a need of comparison between two inputs. This requirement causes a high number of comparisons, which can take a quite long time for bigger binaries. The solution could be extracting some preciously selected parts, storing them in a database and make search by database engine. This next step is not covered by this thesis, but it is going to be researched and developed in the continuing master thesis.

Chapter 8

Conclusion

This thesis describes how to recognize specific behavior by the generic reverse compilation. This issue is divided in two separated tasks – a generic reverse compilation (decompilation) and recognition of specific behavior (on the outputs from decompilation). According to presented results, we can consider both tasks as successfully solved. Of course, there is a great area for future research and a lot of enhancements, but the important part is a verification that this idea is valid and it can be used for e.g. malware detection.

The generic reverse decompilation is a process performed by the Lissom Decompiler. Author of this thesis is one of its developers and he is responsible for the front-end part. Nowadays, this tool is available as the online service on the `http://decompiler.fit.vutbr.cz/`. Moreover, it is used for the malware analysis in the company AVG Technologies [69]. The development of this tool was really time demanding. On the other hand, the quality of its output is very important for the following analysis for recognition of specific behavior. Also, it is a part that ensures the generic approach. The generic approach of decompilation is a requirement, which is not easy to provide. It is visible in the features preview of the other decompilers, see 4.1.

The Lissom Decompiler is able to process binaries from MIPS, ARM, PowerPC, and x86 architectures in the most used object file formats – PE, ELF, COFF, and Mach-O. We can label it as generic and retargetable, because its all parts are unified and the support of a new architecture is provided by supplying the semantics, which describes the architecture. As a part of the decompiler, a toolkit for a recognition and annotation of the statically linked code was created. The decompiler generates the output in three languages: C, Python', and LfD.

The thesis introduces a new simple language LfD (Language for Decompilation). It is the simple language for an analysis, which recognizes specific behavior. The specific behavior is recognized as the percentage of behavior similarity between two files, where the one of them is known and the second one is unknown. A model case is known malware for one architecture and the detection of this malware between unknown binaries for the other architectures. Similarity between each pair is calculated by the tool LfDComparator.

In the results, we firstly presented the output of the decompiler in the C language. Then, we showed that the finding of similarity on the C outputs is problematic also by very complex tools. The last part of results consists of the recognition of specific behavior on the files in LfD language by the LfDComparator. The detection is shown on selected testing files and also on the real malware binaries.

The thesis presented an innovative approach for the analysis of executable's behavior, which is capable to be applied in the generic way. Future research will be aimed for improving the decompiler to be more complex and robust tool, and finding the possibilities how to extract key parts of LfD sources to store them in a database and usage of a database engine for recognition of the similar parts.

Bibliography

[1] C. Kruegel A. Moser and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421 –430, dec. 2007.

[2] V. Adve and C. Lattner. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization*, pages 75–86, Palo Alto, US-CA, 2004.

[3] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. LLVA: A low-level virtual instruction set architecture. In *36th Annual ACM/IEEE International Symposium on Microarchitecture*, San Diego, US-CA, 2003.

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

[5] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *13th International Conference on Compiler Construction (CC'04)*, pages 5–23, Berlin, Heidelberg, DE, 2004. Springer-Verlag.

[6] P. Barbe. The PILER system of computer program translation. Technical report, Probe Consultants Inc, 1974.

[7] T. Baume. Netcomm NB5 botnet – psyb0t 2.5L. [online], 2009. Available on `http://www.baume.id.au/psyb0t/PSYB0T.pdf?info=EXLINK`.

[8] D. L. Brinkley. Intercomputer transportation of assembly language software through decompilation. Technical report, Naval Underwater Systems Center, Princeton, US-RI, 1981.

[9] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *IEEE Security and Privacy*, 5(2):46–54, 2007.

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

[11] E.J. Chikofsky and II Cross, J.H. Reverse engineering and design recovery: a taxonomy. *Software, IEEE*, 7(1):13–17, Jan 1990.

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

[13] 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.

[14] Cisco. Connections counter: The internet of everything in motion. [online], May 2014. Available on http://newsroom.cisco.com/feature-content?type=webcontent&articleId=1208342.

[15] J. Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, 1970.

[16] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[17] +DaFixer. Delphi Reverse Engineering - Adding functionality to a Delphi program. [online], 2012. Available on http://www.woodmann.com/fravia/dafix_t1.htm.

[18] S. K. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Trans. Program. Lang. Syst.*, 22(2):378–415, March 2000.

[19] 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.

[20] DroneBL. Network bluepill – stealth router-based botnet has been DDoSing DroneBL for the last couple of weeks. [online], 2009. Available on http://dronebl.org/blog/8.

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

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

[23] Fast Library Identification and Recognition Technology (FLIRT). http://www.hex-rays.com/idapro/flirt.htm, 2012.

[24] International Organization for Standardization. *[ISO/IEC 9899:2011] ISO/IEC. Programming Languages–C*. Geneva, CH, 2012.

[25] N. Gray. *A Beginners C++*, chapter 4. 2002. Available on http://www.uow.edu.au/~nabg/ABC/ABC.html.

[26] M. H. Halstead. *Machine-Independent Computer Programming*. Spartan Books, 1962.

[27] N. Harbour. Advanced software armoring and polymorphic kung-fu, 2008.

[28] History Of Decompilation. [online], 2012. Available on http://www.program-transformation.org/Transform/HistoryOfDecompilation2.

[29] R. N. Horspool and N. Marovac. An approach to the problem of detranslation of computer programs. *The Computer Journal*, 23(3):223–229, 1980.

[30] 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.

[31] Intel. Intel architecture software developers manual volume 2: Instruction set reference, 1999.

[32] M. Janus. Heads of the hydra. malware for network devices. [online], 2011. Available on https://www.securelist.com/en/analysis/204792187/Heads_of_the_Hydra_Malware_for_Network_Devices.

[33] A. Johnstone, E. Scott, and T. Womack. Reverse compilation for digital signal processors: A working example. In *33rd Annual Hawaii International Conference on System Sciences (HICSS'00)*, pages 316–325, Los Alamitos, US-CA, 2000. IEEE Computer Society.

[34] A. Karnik, S. Goswami, and R. Guha. Detecting obfuscated viruses using cosine similarity analysis. In *Modelling Simulation, 2007. AMS '07. First Asia International Conference on*, pages 165–170, March 2007.

[35] D. Kästner and S. Wilhelm. Generic control flow reconstruction from assembly code. *ACM SIGPLAN Notices*, 37(7), 2002.

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

[37] 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.

[38] 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.

[39] J. Křoustek. Analýza a převod kódů do vyššího programovacího jazyka. Master's thesis, FIT VUT v Brně, 2009.

[40] 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).

[41] 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.

[42] 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.

[43] Lissom. [online], 2013. Available on http://www.fit.vutbr.cz/research/groups/lissom/.

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

[45] P. Matula. Tools for executable file format conversions. Bachelor's thesis, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2011.

[46] A. Meduna. *Automata and Languages: Theory and Applications*. Springer-Verlag, London, GB, 2005.

[47] M. Melanson. Breaking eggs and making omelettes: Intelligence gathering for open source software development. [online], 2005. Available on `http://multimedia.cx/eggs/images/linuxtag-2005-re-paper.pdf`.

[48] Microsoft Corporation. Description of the .pdb files and of the .dbg files. `http://support.microsoft.com/kb/121366`.

[49] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430, Dec 2007.

[50] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, US-CA, 1997.

[51] D. O'Brien. The internet of things: New threats emerge in a connected world. [online], January 2014. Available on `http://www.symantec.com/connect/blogs/internet-things-new-threats-emerge-connected-world`.

[52] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *JOURNAL OF UNIVERSAL COMPUTER SCIENCE*, 8:1016–1038, 2000.

[53] The Linux Information Project. [online], February 2006. Available on `http://www.linfo.org/malware.html`.

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

[55] N. Ramsey and M. Fernández. Specifying representations of machine instructions. *ACM Transactions on Programming Languages and Systems*, 19(3):492–524, 1997.

[56] N. Ramsey and M. F. Fernandez. The New Jersey Machine-Code Toolkit. In *USENIX Technical Conference*, pages 289–302, 1995.

[57] Reverse Engineering Compiler (REC). [online], 2013. Available on `http://www.backerstreet.com/rec/rec.htm`.

[58] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 76–85, New York, NY, USA, 2003. ACM.

[59] V. Schneider and G. Winiger. Translation grammars for compilation and decompilation. *BIT Numerical Mathematics*, 14(1):78–86, 1974.

[60] N. Schwartz. New valgrind decompliation tool: Help wanted. [Valgrind-developers] `http://sourceforge.net/mailarchive/message.php?msg_id=29903738`, September 2012.

[61] The dcc Decompiler. [online], 2013. Available on http://itee.uq.edu.au/~cristina/dcc.html.

[62] 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.

[63] TIS Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, 1995. http://refspecs.freestandards.org/elf/elf.pdf.

[64] K. Troshina, A. Chernov, and Y. Derevenets. C decompilation: Is it possible? In *International Workshop on Program Understanding (IWPU'09)*, pages 18–27, 2009.

[65] UQBT - A Resourceable and Retargetable Binary Translator. [online], 2012. Available on http://itee.uq.edu.au/~cristina/uqbt.html.

[66] P. Čeleda and R. Krejčí. An analysis of the chuck norris botnet 2. [online], 2011. Available on http://www.muni.cz/ics/research/cyber/chuck_norris_botnet.

[67] L. Ďurfina and D. Kolář. C source code obfuscator. *Kybernetika*, 48(3):8, 2012.

[68] L. Ďurfina and D. Kolář. Generic detection of the statically linked code. In *Proceedings of the Twelfth International Conference on Informatics INFORMATICS 2013*, pages 157–161. Faculty of Electrical Engineering and Informatics, University of Technology Košice, 2013.

[69] L. Ďurfina, J. Křoustek, P. Matula, and P. Zemek. Linux.Aidra vs Linux.Darlloz: War of the worms. [online], February 2014. Available on http://blogs.avg.com/news-threats/war-of-the-worms/.

[70] L. Ďurfina, J. Křoustek, and P. Zemek. Design of merhods for retargetable decompilation. Internal document, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2012.

[71] 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.

[72] 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 2013)*, pages 57–62. IEEE Computer Society, 2013.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In HanneRiis Nielson and Gilberto Filé, editors, *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, pages 170–183. Springer Berlin Heidelberg, 2007.

[79] P. Zemek. Design of a language for unified code representation. Internal document, Brno University of Technology, Faculty of Information Technology, Brno, CZ, 2012.

[80] 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.

Appendix A

LfD ANTLR grammar

```
grammar Lfd;

file
    :    (func)+
    ;


func
    :    ID '(' ')' block
    ;


block
    :    '{' statement* '}'
    ;


statement
    :    func_call ';'
    |    loop_stmt
    ;


func_call
    :    ID arguments
    ;

loop_stmt
    :    'LOOP' block
    ;


arguments
    :    '(' argumentList? ')'
    ;

argumentList
    :    argument (',' argument)*
    ;

argument
```

```
    :   DecimalLiteral
    |   FloatingPointLiteral
    |   CharacterLiteral
    |   StringLiteral
    |   booleanLiteral
    |   func_call
    |   'NULL'
    ;

booleanLiteral
    :   'true'
    |   'false'
    ;

// Lex part, symbols start with big letter


DecimalLiteral : ('0' | '1'..'9' '0'..'9'*);

FloatingPointLiteral
    :    ('0'..'9')+ '.' ('0'..'9')* Exponent?
    |    '.' ('0'..'9')+ Exponent?
    |    ('0'..'9')+ Exponent
    |    ('0'..'9')+
    |    ('0x' | '0X') (HexDigit )*
         ('.' (HexDigit)*)?
         ( 'p' | 'P' )
         ( '+' | '-' )?
         ( '0' .. '9' )+
    ;

fragment
Exponent : ('e'|'E') ('+'|'-')? ('0'..'9')+ ;

fragment
HexDigit : ('0'..'9'|'a'..'f'|'A'..'F') ;


CharacterLiteral
    :    '\'' ( EscapeSequence | ~('\''|'\\') ) '\''
    ;

StringLiteral
    :    '"' ( EscapeSequence | ~('\\'|'"') )* '"'
    ;

fragment
EscapeSequence
    :    '\\' ('b'|'t'|'n'|'f'|'r'|'x'|'\"'|'\''|'\\')
    |    UnicodeEscape
    ;

fragment
UnicodeEscape
    :   '\\' 'u' HexDigit HexDigit HexDigit HexDigit
    |   '\\' 'x' HexDigit HexDigit
    ;
```

```
ID
    :    [_a-zA-Z][_a-zA-Z0-9]+
    ;

INT
    :    [0-9]+
    ;

WS
    :    [ \r\t\u000C\n]+       -> skip
    ;

LINE_COMMENT
    :    '#' .*? '\r'? '\n'     -> skip
    ;
```