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

ROZŠÍŘENÝ ABSTRAKT DISERTAČNÍ PRÁCE
EXTENDED ABSTRACT OF PHD THESIS

AUTOR PRÁCE — Ing. LUKÁŠ ĎURFINA
AUTHOR

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

BRNO 2014

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.

Keywords

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

Citace

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

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, steal 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 extended abstract is following: The short review of well known decompilers is summarized in Chapter 2. The generic decompiler is described in Chapter 3. It covers the overall design, and the closer description of front-end. The experience with malware analysis on the output source code from the decompiler is presented in the Chapter 4. The next Chapter 5 studies the opportunities for the comparing of the output source codes and recognizing the specific behavior. Experimental results are shown in Chapter 6. There are the examples of C output from decompiler and the results of specific behavior recognition. The last Chapter 7 concludes the abstract. We assume that the reader is familiar with the basics of formal language theory and the theory of compiler design (see [1]).

Chapter 2

Decompilers

In this chapter, 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.

2.1 The dcc Decompiler

The *dcc* decompiler was developed by Cristina Cifuentes while she was a PhD student at the Queensland University of Technology. It was introduced in her dissertation thesis [3]. 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.

2.2 Boomerang

Boomerang[1] is an open source project. It was strongly inspired by UQBT—A Resourceable and Retargetable Binary Translator in 1996 [14], 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.

The Boomerang decompiler is probably the first attempt to create a retargetable decompiler by using domain-specific language for description of the target architecture. The SLED language, developed within the New Jersey Machine-Code Toolkit [12] project, was used for this purpose. This project exploits the SLED language [11] 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, the authors of the Boomerang decompiler have to use it together with the RTL-based semantics description language SSL [4]. 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.

2.3 REC Studio

REC Studio—Reverse Engineering Compiler [13] is a freeware, but not open-source, interactive decompiler, which is still under development. It reads a Windows, Linux, Mac OS X or raw executables (e.g., firmware), and attempts to

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

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 Mac OS. However, this software is unstable on several architectures (e.g., Windows), and it often crashes during decompilation.

The author wrote on his web page [13] 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 OFFs: 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.

2.4 Hex-Rays Decompiler

The *Hex-Rays Decompiler*[2] is the nowadays decompilation "standard". It is implemented as a plugin to the IDA disassembler[3]. The Hex-Rays Decompiler supports the x86 (i.e., not x86_64) and ARM target architectures. It also supports both major OFFs—ELF and PE. The output is generated as a highly readable C code; however, the output is not designed for re-compilation, only for more rapid comprehension of what the program is doing.

This software is commercial and distributed without sources. The first version of the x86 decompiler was released in 2007, and support of ARM decompilation has been added in 2010. 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.

[2] `www.hex-rays.com/products/decompiler/`
[3] `www.hex-rays.com/products/ida/`

Chapter 3

Lissom Decompiler

Decompiler is developed within the team, where the author of thesis is one of the team members. Author is mainly involved in the reasearch and development of 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 [8, 15–20].

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 [10], 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.

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

3.1 ISAC Language

The ISAC language was developed within the Lissom project at Brno University of Technology [9]. 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.

3.2 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. Nowadays, the use of LLVM spans over many different areas, including compilation (Clang, LLVM D Compiler, Trident Compier), 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), 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.

3.3 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 3.1. 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.

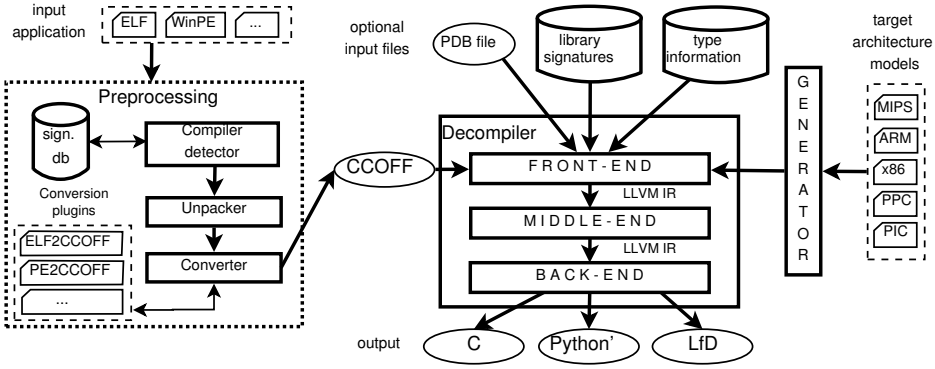

Figure 3.1: The concept of the retargetable decompiler.

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 [8]. 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 [7]. 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* [5], 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. [18]. 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 5.2). 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.

3.4 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 the thesis. The main part of the front-end is a static analysis of the decoded code before generation of final LLVM IR code. This part includes several specific analysis, some of them are architecture-specific. The cooperation and work flow of decfront is shown in Figure 3.2. This figure presents complete image of decfront architecture design.

The cursory descriptions of selected decfront analysis follow. It is important to note that all these analysis are static.

Overview of Selected 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. Selected methods are described in the following text.

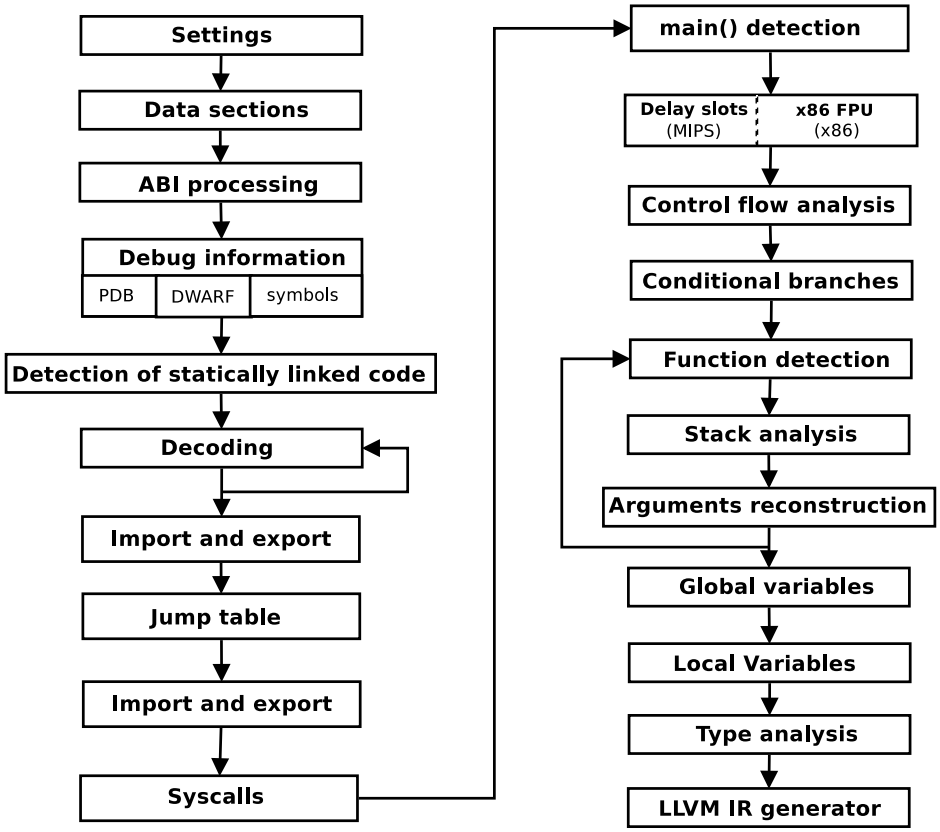

Figure 3.2: The architecture of decfront.

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 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 caller.

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.

Jump Table Analysis

Jump table analysis depends directly on 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 of this jump table. Our motivation is to have direct calls of linked functions. This point requires to connect addresses of 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.

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.

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 an additional method, which uses debugging information, but it is very simple method. It just creates the functions on the addresses that are gained from debugging information without any additional validation.

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.

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.

Local Variables Detection

One type of local variables is created by the stack analysis. But, the local variables may be represented also by 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.

LLVM IR Generator

LLVM IR generator is the last part of the front-end. The task of this part is to generate LLVM IR into a text 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. 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
```

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

Middle-end and Back-end

Middle-end and back-end is developed by Petr Zemek, therefore they are not described in this abstract. It is possible to read more details about them in the thesis.

Chapter 4

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 analysis of malware program, which is also published in [17].

4.1 Psyb0t – MIPS Malware

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* [2], which attacks network infrastructure devices (e.g. modems and routers) running MIPS processors with Linux-based operating systems. 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 [7] 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. The last part of the preprocessing phase is a conversion of the unpacked ELF file into an internal CCOFF format.

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.

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. 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.

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 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 [18] for details.

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.

Analysis of the Obtained Results

Psyb0t is an IRC bot, which reads the topic of the IRC channel after connecting to the server and gets commands from this topic. It scans devices in the network and tries to log in by default usernames and passwords or uses an exploit when the login fails. Once a shell of the vulnerable device is acquired, psyb0t downloads itself from a remote server by using the `wget` application into the victim's location `/var/tmp/udhcpc.env`. This new instance of psyb0t is executed afterwards. It supports classical malware actions like DDoS attacks, brute-force attacks on router passwords, download of files, visitation of web pages, or executing shell commands [6].

We have presented the decompilation process in a step-by-step way. 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. The most important parts of the `main` function are listed in Figure 4.1. The comments were added manually. Selected parts are listed separately with describing notes.

The first operation in `main` is opening of a file named `udhcpd.mtx` in a temporary folder. It is opened in the writing mode. The author of psyb0t followed good practice and checked the result of the operation.

```
uint32_t *file = fopen("/var/tmp/udhcpd.mtx", "w");
```

```
int main(int argc, char **argv) {
  //...
  uint32_t *file = fopen("/var/tmp/udhcpd.mtx","w");
  //...
  uint32_t fd = fileno((uint32_t *)file);
  //...
  uint32_t err_code = flock(fd, LOCK_EX | LOCK_NB);
  //...
  RSeed();
  //...
  Daemonize();
  //...
  system("/etc/firewall_start");
  system("iptables -A INPUT -p tcp --dport 23 -j DROP");
  system("rm -f /var/tmp/udhcpc.env");
  //...
  backup();           // Backup file /var/tmp/hosts
  //...
  function_404b1c(); // Prepare IRC nickname
  //...
  function_4056cc(); // Await for commands
  //...
  fclose(fd);         // Remove mutex file and quit
  //...
}
```

Figure 4.1: Simplified code of the `main` function by using the Lissom project retargetable decompiler.

```
var3 = (uint32_t)file;
if (file == NULL) {
    return 1;
}
```

Subsequently, there is the obtained file descriptor, which is checked for validity. If it is valid, the application tries to lock the file. After this operation, we can better understand the suffix `.mtx` in the name of the file, because it serves as a mutex. The lock is exclusive and it is does not block when the locking is done. The mutex is acquired only if there is no other running instance of psyb0t. Otherwise, the application is terminated.

```
uint32_t fd = fileno(file);
```

```
if (fd == -1) {
    var3 = 1;
    return 1;
}
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
```

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.

Chapter 5

Detection of Specific Behavior

Our retargetable decompiler creates a platform that processes the various types of binaries into unified representation—C language or Python'. 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 design a schema to find the similarity between two arbitrary binaries that can be processed by the decompiler. The idea is that we have a (malware) binary that is well-know for us and we compare it with unknown binaries to find similar (malware) binary. Following the problematic situation in 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 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 implemnted by the author of this thesis.

5.1 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.

5.2 Language LfD

LfD is abbreviation of 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 in the thesis for the complete listing. This form is required by 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.
- loops – are represented by the keyword `LOOP`.
- constants – can be strings, integers, or floating point numbers.

[1] `http://jplag.ipd.kit.edu/`
[2] `http://theory.stanford.edu/~aiken/moss/`
[3] `http://www.antlr.org/`
[4] `http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf`

An example of output is shown in Figure 5.1. 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 5.1: An example with the program in LfD.

5.3 Tool LfDComparator

This tool compares 2 inputs in LfD language and decides their similarity. The output is given by the 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 ANTLR grammar, so we are able to use ANTLR framework to generate lexer 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 graph and constants.

Chapter 6

Results

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

6.1 C Outputs

The example shows the fibonacci function in Figure 6.1. 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.

6.2 Detection of Similarity

In this part, we compare the results of JPlag, Moss, and LfDComparator in the detection of similarity. Firstly, we test the detection on the binaries from the different compilers. We have 10 testing C files. We produce 30 different binaries for each file, they are the 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 – 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 the generic detection on binaries from different architectures and file formats.

The results of Moss are presented in Figure 6.2. 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 6.3 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.

LfDComparator achieves better results as the previous tools, see Figure 6.4. The reason is usage of specialized language LfD 5.2 and also the 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 decompiler and LfDComparator.

For these kind of testing, it is important to not create false positives. LfdComparator has 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 bellow 10%.

Finally, we test the LfdComparator with real malware. There is used malware Aidra and Darlloz, 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 table with results is available in the thesis. The table with results is available in the thesis.

The disadvantage is a need of comparison between two inputs. This requirement causes the 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 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.

```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 func_401560(int32_t
    a1) {
    if (a1 < 3) {
        return 1;
    }
    int32_t v1 = func_401560(
        a1 - 1);
    while (a1 > 4) {
        a1 -= 2;
        v1 += func_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, func_401560
        (v2));
    while (v1 >= v3 + 1) {
        v3++;
        printf("Input number:
            ");
        scanf("%d", &v2);
        printf("fibonacci(%d)
            = %u\n", v2,
            func_401560(v2));
    }
    return 0;
}
```

(b) Output from decompiler.

Figure 6.1: Comparison for `x86 - PE` with strip and optimization `-O2`.

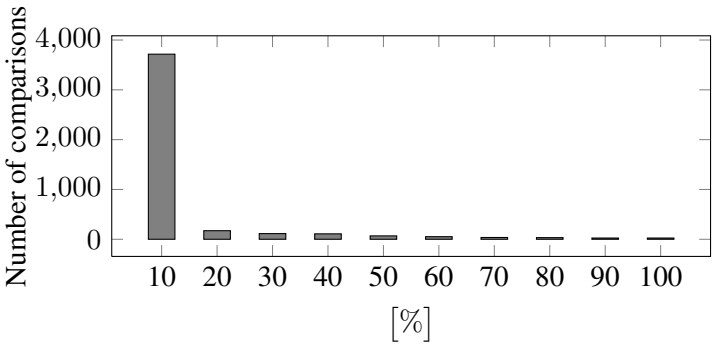

Figure 6.2: Results of comparison from Moss.

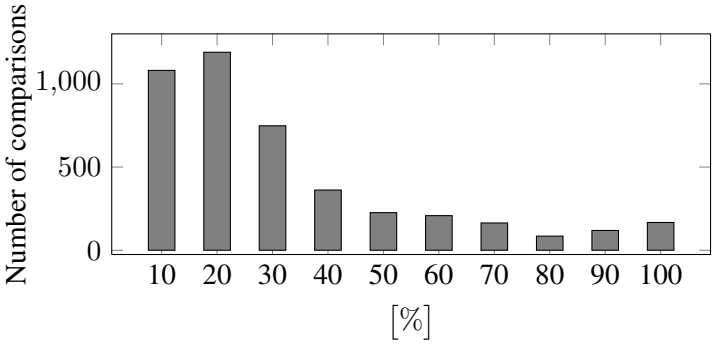

Figure 6.3: Results of comparison from JPlag.

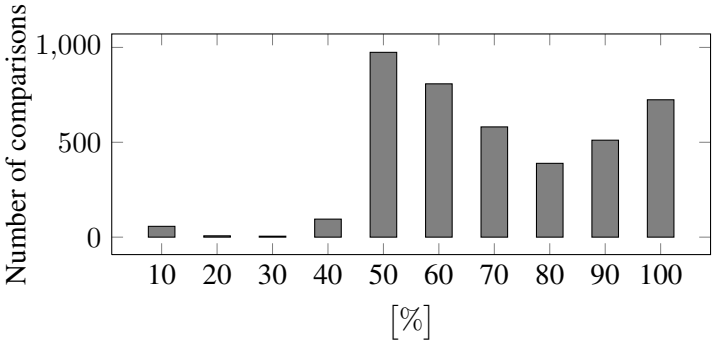

Figure 6.4: Results of comparison from LfDComparator.

Chapter 7

Conclusion

This extended abstract describes how to recognize specific behavior by the generic reverse compilation. This issue is divided in two separated tasks – 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 process performed by 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. 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.

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 extended abstract introduces new simple language LfD (Language for Decompilation). It is 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. Model case is known malware for one architecture and the detection of this malware between unknown binaries for the other architectures. The similarity between each pair is calculated by the tool LfDComparator.

In the results, we firstly presented the output of the decompiler in 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 extended abstract presented an innovative approach for the analysis of executable's behavior, which is capable to be applied in the generic way. The 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 stored them in database and usage of database engine for recognition of the similar parts.

Bibliography

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

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

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

[4] 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.

[5] 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.

[6] 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`.

[7] 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).

[8] 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.

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

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

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

Curriculum Vitae

Ing. Lukáš Ďurfina

Faculty of Information Technology
Brno University of Technology
Božetěchova 2
Brno 61266 Czech Republic

Phone: +420 608 546 603
email: idurfina@fit.vutbr.cz
URL: http://www.fit.vutbr.cz/~idurfina/

Born: November 19, 1986—Nitra, Slovakia
Nationality: Slovak Republic

Education

2010–now	PhD study in Computer Science and Engineering, FIT BUT
2008–2010	Ing. in Information Systems, FIT BUT
2005–2008	Bc. in Information Technology, FIT BUT
1997–2005	Secondary grammar school, Nitra

Professional carreer

`2010-now` freelancer programmer, Brno
`2011-now` AVG Technologies, developer, Brno

Language skills

English fluent written and spoken
Slovak native speaker

Interests

Football, reverse engineering, software development, wine, sports, reading