GENERIC DETECTION AND ANNOTATIONS OF THE STATICALLY LINKED CODE

Lukáš Ďurfina*, Dušan Kolář**
*Faculty of Information Technology, Brno University of Technology, Božetěchova 1/2, 612 66 Brno, Czech Republic, e-mail: idurfina@fit.vutbr.cz
**Faculty of Information Technology, Brno University of Technology, Božetěchova 1/2, 612 66 Brno, Czech Republic, e-mail: kolar@fit.vutbr.cz

ABSTRACT

Detection of the statically linked code is one of the important steps in a process of decompilation. It restricts a code, which can be skipped by the decompiler. Type annotations provide an additional information about the number, types, and suitable names for arguments and return values of recognized functions in recognized statically linked code. This is important for generation of calls for these functions. The detection is based on the generic signatures, which are created from the static libraries. The signatures are composed of the first bytes of library modules, CRC codes, module sizes, public symbols, and optionally tail bits or references. A tree structure of signature improves performance by decreasing a count of compared bytes. Generic approach of detection is achieved by an usage of a common object file format. The process is not restricted on specific architecture or file format. However, there are situations when a conflict in the detection can be resolved only by an analysis in the decompiler. Impact of signature usage is verified by the tests with the decompiler.

Keywords: *statically linked code, signature, detection, type annotations, decompiler*

1. INTRODUCTION

Detection of statically linked code 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 decompiler. The second aim is a delivery of valuable information about recognized code. We can directly mark some piece of the code as a specific function. This tool-chain was developed for a usage in decompiler, which is created by our project Lissom [2]. The architecture of decompiler and the significance of statically linked code removal for decompilation was described in the article [3]. The motivation is to provide a better decompilation of binaries with the statically linked code. The detection of such a code makes the decompilation faster and more accurate due to better knowledge of called functions.

Linking of static libraries is available for all widely used platforms and compilers, therefore the process should be generic. Also, it should follow idea of the decompiler, which is built as generic and retargetable. Naturally, we cannot assume that we recognize same version of library on the different architectures by a single signature. The goal is to have a single tool for same action. The fact that the libraries are from the different architectures and file formats has to be irrelevant for such a tool. We achieve it by usage of unified object file format. Libraries are transformed to this format and the tool-chain starts the work with handling the files in this format.

The signatures are not directly created from transformed libraries. We extract patterns from them. Each module from the library is described by a single pattern. Finally, the patterns are transformed into signatures and if there are conflicts, they are stored into separated file. 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, return values, and suitable names for variables with these values. This is covered by type annotations. Lissom decompiler is based on LLVM [4]. Therefore the C types are directly transformed into LLVM types and these types are used in type annotations for more precise description.

In the Section 2 there is described the current state of this research field. The following Section 3 presents complete process how the signature is created. For meaningful usage the decompiler needs more information about functions from libraries. This is ensured by type annotations, what is written down in Section 4. Section 5 brings a closer look how the signatures and type annotations are used inside of decompiler. The experimental results are shown in Section 6. The paper is closed by Section 7 with idea for future research. This article is an extended version of the paper [1].

2. THE STATE OF THE ART

The detection of statically linked code is solved on different levels in various decompilers. The decompiler dcc [5] is aimed only at MS-DOS executables, so it can detect linked static libraries for that operating system. It uses a signature generator, which generates signatures with the first 23 bytes of standard *lib* files. It uses wild-card byte `F4` (the opcode for `HALT` instruction), because it occurs very rarely [6]. For each triplet compiler vendor, memory model, and compiler version there is created own static library signature.

The decompiler Boomerang [7] supports decompilation for several architectures and this is probably reason why it does not support recognition of statically linked code. It has only some simplified header files with function declarations and typedefs. These files are used for dynamically linked functions that are recognized from imports in executables.

REC decompiler [8] is in similar situation like Boomerang. It has some special signatures for recognition of used compiler, but these signatures cannot be used

for some complex elimination of statically linked code. It has also header files, but these files are not simplified, but are taken in original state. They contain the declarations of structures, but they do not seem to be used.

The commercial Hex-Rays Decompiler [9] uses own F.L.I.R.T technology for recognizing standard functions [10]. It stands for Fast Library Identification and Recognition Technology. The technology tries to reduce false positives and to require a minimum of processor and memory requirements. There are used signature files for recognition. The function is represented by sequences of 32 bytes. If there are same sequences of bytes for two or more functions, the additional elements are used as CRC code, the first different byte, or different referenced function. Moreover, these files are compressed, so they take smaller amount of space, and also it is hard to verify their real content.

3. CREATION OF SIGNATURES

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 various formats into single common format. This is used in our solution. We use tool bintran [11], which converts object files from ELF, PE, and Mach-O formats into specific COFF file format developed within project Lissom. Hence, we can have a single tool for processing only this unified format. The static library is an archive of object module files. Bintran extracts these module files from library, converts each module to COFF format, and finally stores all of them into own archive format.

3.1. Pattern files

The second step is ensured by the tool *coff2pat*. 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 lower number for minimal size of non-variable bits causes too many false positives in recognition process. 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. This address is set by linker in the linking process.

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 there have all instructions same length 32 bits. This length forms minimal compare unit for signature creation and also for later searching in executables.

The example is shown in Figure 1, it is lightly edited for paper purpose. Sequences of bits are replaced by `[]` blocks. This pattern file is extracted from library for MIPS architecture, where all instructions have same size of 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 a variable bit. These 256 chars 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 behind 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 CRC16 algorithm. The size of module follows. Numbers are in hexadecimal format.

Behind module size, 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 last pattern in example). The references are used for distinguishing of modules, which have same other parts. 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 otherwise same modules.

```
R14kdP0a7q
8
32
6
[256 bits] 08 0397 0074 2 0000 printf 0040 _printf_r 1 0028 _vfprintf_r [tail bits]
[256 bits] 70 B422 0160 2 0000 putc 00B8 _putc_r 2 0090 __swbuf_r 0098 __sinit [tail bits]
[256 bits] 0C 05D8 00EC 1 0000 asprintf 3 0078 realloc 00AC malloc 00C0 strcpy [tail bits]
[256 bits] 1C 4487 009C 1 0000 bind 2 003C _errno 0060 sceNetInetBind [tail bits]
[256 bits] 0C 634C 0058 1 0000 closedir 3 0010 free 002C _set_errno 0034 _errno [tail bits]
[256 bits] 0C 5AC0 008C 1 0000 chdir 0 [tail bits]
```

Fig. 1 : A part of pattern file

```
Sig14sd77x
8
32
7
[256b] | 1 N FF E60F 0B0C 2 0000 permute 016C getopt
[32b] [32b] [192b] | 1 N FF 904A 0150 1 0000 memcpy
            [192b] | 1 N 64 D4BA 0084 1 0000 memmove
            [192b] | 1 N B4 6347 00E0 1 0000 memset
            [192b] | 1 N B4 6347 00D4 1 0000 stpncpy
            [192b] | 2 T 64 706F 008C 27 1 1 0000 strcpy T 0090 35 0 1 0000 stpcpy
      [32b] [192b] | 1 N 5C AD03 007C 1 0000 strcmp
            [192b] | 2 R 00 0000 0074 0020 _svfiscanf 1 0000 iscanf 0020 _svfscanf 1 0000 scanf
```

Fig. 2 : A part of signature file

3.2. Signature files

The signature file is created from one or more pattern files. The main reasons for this transformation are detection of conflicts and finding of common first bits. The conflict is a state, when two or more patterns have equal first bits, CRC codes, references, and tail bits. Such patterns are excluded from signature file into separate file, called exception file. This file is not further processed. It only stores information about excluded patterns and the reason for the exclusion.

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. In out example 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 a unique first 32 bits. This process continues in the 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, references, or tail bits. If they cannot be distinguished, they are moved out into exception file. According to the way how the module can be distinguished, there are three types of signature:

- N - normal type: the module is recognized by first bits or CRC code, there is described exactly one module.
- T - tail type: the modules are recognized by tail bit, there can be two or more described modules.
- R - reference type: the modules are recognized by different references, there can be two or more described modules.

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 module, because larger modules reduce more code, which afterwards is not tested 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 recognizing the type of file (`Sig14sd77x`). It is followed by the number of bits in byte, the size of compare unit and the number of lines with signatures. Header is followed by lines with signatures, where on the single line there is at least one signature. There could be more signature on one line only if tail or reference type is used.

Division of patterns into groups is used to create tree-like format, which saves a memory consumption of loaded signatures, and also, it helps to get more efficient search. In Figure 2, 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 192 bits, so they are listed separately.

The format of line is related to pattern format. Starting bits are first, but they can be divided into groups as it was described. Then, there is a separator `|`, its effect is only visual. The next number is count of described modules. The following letter designates the type, the valid letters are `N` – normal type, `T` – tail type, or `R` – reference type. For normal type the count of modules has to be always 1. The next two numbers are related to CRC and it is a number of used bytes and 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.

For the case of tail the count of described modules can be 2 or more. The additional information about important bit in tail bits is stored behind module size. There is the offset of that bit and its value. The part with public symbols is the same. The description of next module 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.

```
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.FILE*, %struct.IO_marker*, i32, i32, i32, i16, i8, [1 x i8],
                8*, i64, i8*, i8*, i8*, i8*, i32, i32, [40 x i8] }
9
fprintf i32 3 %struct.FILE* file, i8* buffer, ... # int fprintf (FILE *, const char *, ...)
printf i32 2 i8*, ... # int printf (const char *, ...)
remove i32 1 i8* path # int remove (const char *)
rename i32 2 i8* old_name, i8* new_name # int rename (const char *, const char *)
tmpfile %struct.FILE* tmp_file 0  # FILE * tmpfile ()
tmpnam i8* tmp_file_name 1 i8* # char * tmpnam (char *)
fclose i32 1 %struct.FILE* file # int fclose (FILE *)
fflush i32 1 %struct.FILE* file # int fflush (FILE *)
fopen %struct.FILE* file 2 i8* filename, i8* mode # FILE * fopen (const char *, const char *)
```

Fig. 3 : A preview of file with library type annotations

The reference type is similar to tail type, but instead of tail bit position and value, we have there the position of reference and the referenced function. There is requirement that used referenced function has to be also described by signature file. We are able to evaluate the correctness of detection only if referenced function is also detected and recognized.

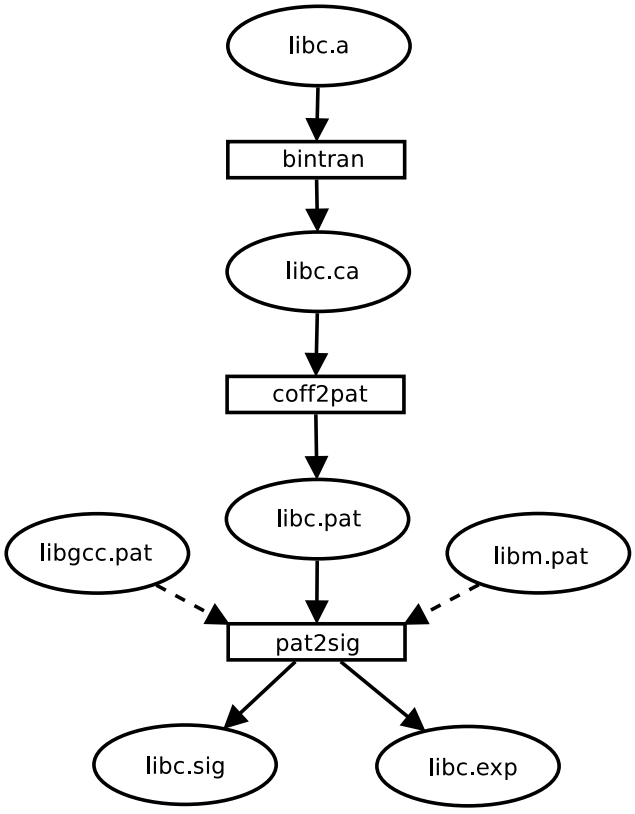

Fig. 4 : The process of signature file creation.

Process of the signature file creation is illustrated in Figure 4. With dash lines there is indicated an option that more pattern files can be transformed in a single signature. By this way we remove more collisions between patterns. Also, it is practical to use on more libraries from single compiler to cover the produced code of this compiler by the single signature file.

4. TYPE ANNOTATIONS

Type annotations bring an additional data for recognized functions in the statically linked code. The advantage of separation is in possibility of its usage for functions, which are known from imports of executable files. These functions are linked dynamically from shared libraries (dll/so). The file with type annotations is shown in Figure 3.

The base of file is generated from one or more C header files and optional data can be added manually (names of arguments and return values). The tool is implemented as a plugin into the Clang compiler. The Clang is C front-end for LLVM and it supports usage of plugins, which are able to connect to various states of C file or header file processing. The plugin is connected to a part, which creates an abstract syntax tree. If the declaration of a new function is finished, the plugin adds information for this function – the number and types of arguments and a type of the return value. If some of type is a structure, it processes its declaration and creates a separated type annotation for this structure.

The file consists of two main parts – declarations of structures and declarations of functions. The number of declared structures is on the first 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 specifying the count of function declarations. And then, there is one declaration for each line. If there is structure used in another 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*`. Information like modifier `const` is dropped, because it is not applicable in decompilation.

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 an optional string value,

which represents suitable name for variable where the return value is stored. Then, there follow a number of arguments, their types, and also optional suitable argument names. Each type, alternatively pair of type and name, is divided by a comma. Behind terminating symbol # there is the original C declaration, which is used by decompiler only for informative purposes.

Optional parts of annotations are an extension to version in paper [1]. Their disadvantage is a requirement to fill them by a hand. We are not able to get them fully automatically from the header files, because for the return values there are not at all, and also arguments can be listed only by the types. On the other hand, if there is a name for the argument, we take it to the type annotations file. If the author adds this extended information, the quality of code generated from the decompiler is higher. Earlier, for the call of function `fopen`, we generated the code similar to this: `var_8a9d = fopen("a.txt", arg_8a9b)`. With the extended type annotations we produce: `file = fopen("a.txt", mode)`.

5. DETECTION BY MATCHING SIGNATURES

There is a lot of different libraries, which results into a lot of signatures. 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 also by particular 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 to make a decision to use it or not.

As the part of decompiler there is a special tool for recognizing 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 this prevents finding of some library code. This tool helps the decompiler to set conditions for signature selection. For example if an executable is for `x86` architecture and it is `PE` format, compiled by `Delphi`, we take only signatures for these specific parameters. It saves a time and resources due to disposal a lot of signatures for other compilers as `gcc` or `Microsoft Visual C`.

Decompiler loads all selected signatures and sorts them by the same strategy as it is applied on signature file before writing into the file. After load of an executable, the search is started. The search is performed only on the bytes from the code sections, because it is wasteful to look on data or other sections. The decompiler determines the size of minimal compare unit from the loaded signatures. 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 bits.

If there is a hit on the first bits and CRC checksum, there is a control of tail bit or reference if needed. Tail bit is checked by a value on position given by the number from signature. The check of reference is more complicated. The instruction on the address of reference has to be decoded. It is checked that it contains a jump. The target address of this jump has to be calculated for the resolution if it is the address of required referenced function. If everything is correctly compared, the code is marked as statically linked code. Such a code is no more searched with other signatures and the 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 annotations. This is done simply by the matching names.

6. EXPERIMENTAL RESULTS

For tests, we take `libc.a` library for various architectures. There are small differences between compilers for each architecture. We use these compilers: `mips/elf - psp-gcc 4.3.5`, `pic32/elf - xc32-gcc 4.5.2`, `ppc/elf - powerpc-apm-linux-gnu-gcc 4.5.1`, `arm/elf - arm-elf-gcc 4.4.0`, `x86/elf - i686-pc-linux-gnu-gcc 4.7.2`, and `x86/pe - i686-pc-mingw32-gcc 4.8.1`. We count the number of modules for each library. Then, we create patterns, and note a number of created patterns. The difference between module and patterns count is created by too small modules, which are ignored by the `coff2pat` tool. After that we convert patterns into signatures by `pat2sig`. There is the difference again. This one is caused by collisions between patterns, which are moved into exception file. The results are shown in Table 1. The ratio column shows the conversion rate from patterns to signatures. The ratio 100% would mean no collision in creation of signatures. In real world, that is possible only for libraries with small number of modules. If we compare the current results with previous results, which were published in the paper [1], we have higher ratio. This positive effect is given by addition of the reference type of signatures. In another words, extended solution produces smaller amount of collisions.

	Modules	Patterns	Signatures	Ratio
mips/elf	588	541	529	97,8%
pic32/elf	1539	1215	1092	89,9%
ppc/elf	1483	1457	1402	96,2%
arm/elf	481	440	436	99,1%
x86/elf	1508	1423	1378	96,8%
x86/pe	1179	1021	917	89,8%

Table 1: A comparison between `libc.a` from different architectures.

Next test is aimed on an impact of signature usage during decompiler runtime. We use simple Hello world program. The test shows how the signatures save a time by elimination of decompiled code. The results are presented in Table 2. All times are in seconds. The column *Without sig.* shows the running time of decompiler, when no signatures are used. The column *With sig.* contains the times, when we use created signatures for `libc.a`. The time of searching for statically linked code is in the last column *Search*. This time is a subset of time from *With signatures*.

	Without sig.	With sig.	Search
mips/elf	3,38	0,67	0,21
pic32/elf	0,47	0,25	0,04
ppc/elf	202,82	61,82	0,25
arm/elf	9,91	7,26	0,31
x86/elf	14,53	3,12	0,27
x86/pe	11,73	2,28	0,18

Table 2: A running time of the decompiler.

The time for architecture ppc is significantly longer than other times. This is caused by some special features in architecture description and it is not related to signatures. The given tests prove the importance of signature usage in decompilation. The time with signatures is decreased in comparison to time without an application of signatures.

7. CONCLUSION

The improved framework for the generic recognition of statically linked code is presented. The solution is generic and it can be applied for various architectures and object file formats. The new important feature is the support of the references. This feature adds more complex connection with decompiler, because it requires decoding of instruction. The tests are performed on library from several different architectures and the reached results are excellent. The searching is fast enough, therefore it does not slow down the run of decompiler. And, moreover, the recognized code is not decompiled as other user code, what means less time for analysis of decompiler output.

For the future research, we want to enhance generation of type annotations. Now, there is a part for names of arguments and return values that have to be added manually. We see a possibility to develop a tool, which will be able to parse and extend base type annotations from Clang plugin. Such a tool should parse C files to be able to find out suitable names for them and finally update type annotations.

ACKNOWLEDGEMENT

This work was supported by the project FIT-S-14-2299 Research and application of advanced methods in ICT, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

REFERENCES

[1] ĎURFINA, L. – KOLÁŘ, D.: "Generic detection of the statically linked code," in *Proceedings of the Twelfth International Conference on Informatics (INFORMATICS 2013)*, Spišská Nová Ves, 2013, pp. 157–161, ISBN 978-80-8143-127-2

[2] Lissom, http://www.fit.vutbr.cz/research/groups/lissom/, 2013.

[3] ĎURFINA, L. – KŘOUSTEK, J. – ZEMEK, P. – KOLÁŘ, D. – HRUŠKA, T. – MASAŘÍK, K. – MEDUNA, A.: "Design of a retargetable decompiler for a static platform-independent malware analysis," *International Journal of Security and Its Applications (IJSIA)*, vol. 5, no. 4, pp. 91–106, 2011.

[4] The LLVM Compiler Infrastructure, http://llvm.org/, 2013.

[5] The dcc Decompiler, http://itee.uq.edu.au/~cristina/dcc.html, 2012.

[6] CIFUENTES, C.: "Reverse compilation techniques," Ph.D. dissertation, School of Computing Science, Queensland University of Technology, Brisbane, QLD, AU, 1994.

[7] Boomerang, http://boomerang.sourceforge.net/, 2012.

[8] Reverse Engineering Compiler (REC), http://www.backerstreet.com/rec/rec.htm, 2011.

[9] Hex-Rays Decompiler, www.hex-rays.com/products/decompiler/, 2013.

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

[11] KŘOUSTEK, J. – MATULA, P. – ĎURFINA, L.: "Generic plugin-based convertor of executable file formats and its usage in retargetable decompilation," in *6th International Scientific and Technical Conference (CSIT'11)*. Ministry of Education, Science, Youth and Sports of Ukraine, Lviv Polytechnic National University, Institute of Computer Science and Information Technologies, 2011, pp. 127–130.

Received December 2, 2013, accepted December 28, 2013

BIOGRAPHY

Lukáš Ďurfina was born on 19. 11. 1986. In 2010 he graduated (Ing.) at the Faculty of Information Technology, Brno University of Technology (FIT BUT). He is PhD student and his scientific research is focusing on reverse engineering.

Dušan Kolář was born on 3. 1. 1971. In 1994 he graduated (Ing.) with distinction at the Faculty of Electrical Engineering and Computer Science at Brno University of Technology (FEECS BUT). He defended his PhD in the field of cybernetics and computer science in 1998; his thesis title was "Functional Technology for Object-Oriented Modeling and Databases". Since 1998 he worked as assistant professor on FEECS BUT, and since 2005 he is working as associate professor at Department of Information Technology at FIT BUT. His scientific research is focusing on formal languages, automata, compilers, programming languages, and advanved database systems.