Journal of Network and Innovative Computing.
ISSN 2160-2174 Volume 2 (2014) pp. 224–232

A Novel Approach to Online Retargetable Machine-Code Decompilation

Lukáš Ďurfina, Jakub Křoustek, Peter Matula, and Petr Zemek

Faculty of Information Technology,
IT4Innovations Centre of Excellence,
Brno University of Technology,
Božetěchova 1/2, 612 66 Brno, Czech Republic

Abstract: **Machine-code decompilation, belonging to the area of reverse engineering, has found its applications in many real-world areas. Analysis of malicious software, search for vulnerabilities, and source-code recovery are some of the most important uses. As there exists a diversity of different platforms on which software can be run, an existence of a generic decompiler would be highly appreciated.**

This paper presents an extended version of our retargetable decompiler that also allows decompilation of raw binary code, such as firmware or code snippets. More specifically, in the present paper, we provide a description of a retargetable decompiler that is being developed within the Lissom project. First, we give an introduction into the area of machine-code decompilation, including a brief discussion of existing tools. Then, we describe the concept and architecture of the decompiler. As it is available in the form of a web service, we also provide its description. Finally, we summarise our results, present a case study of using the tool for analysing malicious software, and conclude the paper by several remarks on future research.

Keywords: reverse engineering, decompilation, retargetable decompiler, raw machine code, code snippets, web service, Lissom

I. Introduction

A machine-code decompiler performs the reverse operation to that of a compiler—it takes binary program code for a given architecture, and produces a high-level representation of the program. Since the appearance of the first decompilers, they have found their applications in many areas, including (but not limited to) high-level analysis of malicious software (malware), discovery of vulnerabilities, retrieval of lost source code, and reversing binary code for the purposes of interoperability [1].

Over the years, the number of different platforms on which software can be run has dramatically increased [2]. This fact poses a challenge for the authors of decompilers because creating a decompiler for a new platform from scratch is no longer feasible. To cover the current state of the art, we next focus on existing decompilers and analyse their generality.

The Boomerang decompiler [3–5] tried to use a generic description of a processor, which could be used alongside with common processing code. However, the used description language was too weak to cover more complex architectures, such as x86. Moreover, this decompiler is not developed anymore. The currently widely used Hex-Rays Decompiler [6] is a proprietary solution, so we do not know the details. Nevertheless, from the fact that it has supported for a long time only x86 and it took quite a long time to add support for ARM, we can assume that this solution is not easily retargetable. Another decompiler, SmartDec [7], supports only the x86 and x86-64 architectures. Finally, we can mention the REC decompiler [8], which is available for free, but without source code. It supports architectures x86, x86-64, MIPS, PowerPC, and mc68k. The range of features varies for individual architectures, so it seems that adding support for a new architecture does not follow any uniform way.

This paper is focused on describing a retargetable decompiler that is being developed withing the Lissom project [9]. Its main advantage over the existing tools lies in its generality—to add support for a new architecture, one has to first describe this architecture, and then utilize the already developed tools to build a decompiler for that architecture. In this way, the development time for adding support for a new architecture can be dramatically decreased [10, 11]. The decompiler is freely available in the form of a web service, and is accessible via any commonly used web browser.

In this paper, we further extend the previous concept described in [12]. Most importantly, the novel approach is in the dealing with the decompilation of raw binary snippets, which represents a very useful feature often used during malware analysis.

The present paper is organized as follows. After this introductory section, Section II describes the concept and architecture of the decompiler. Then, Section III introduces a new feature of our decompiler—decompilation of raw machine code. Section IV provides a brief description of the decompilation service. Section V summarises our results and presents a case study of using the tool for malware analysis. Finally, Section VI concludes the paper by stating several remarks on future research.

II. Architecture and Concept

The Lissom project [9] retargetable decompiler is developed to process any executable file, independently of a particular target architecture or object file format [11, 13, 14]. Its structure is depict in Figure 1.

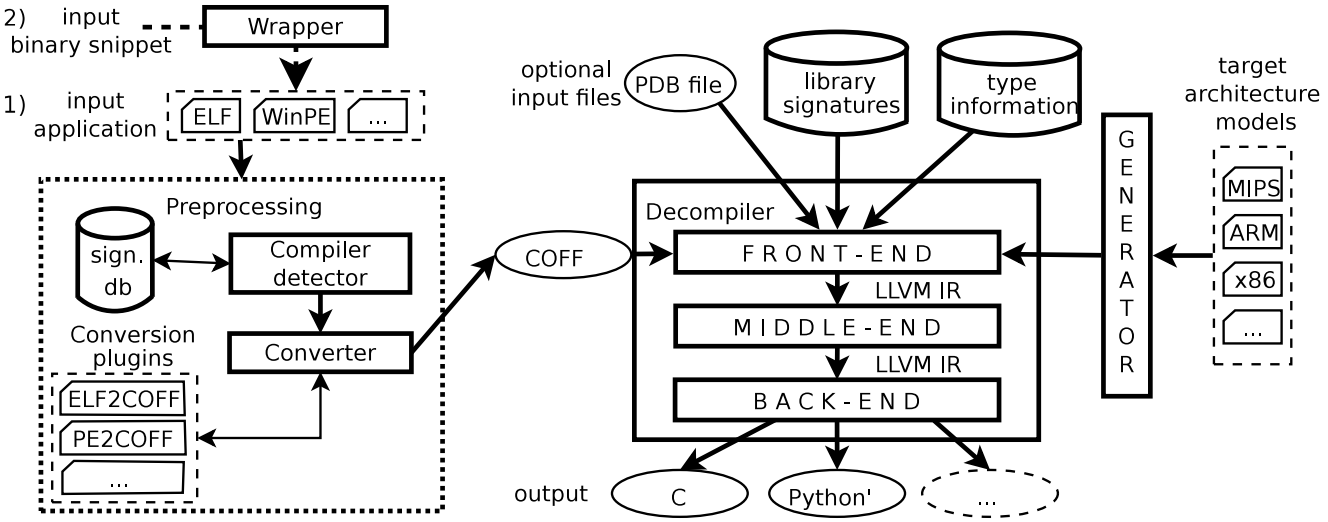

Figure. 1: The architecture of the decompiler.

As we can see, it consists of two essential parts—the *preprocessing part* and the *decompiler core* (containing the *front-end*, *middle-end*, and *back-end* parts),

A. Binary Snippet Wrapper

Besides the common inputs—executable applications, object files or libraries, as of version 1.7 of the decompiler, it is also possible to reverse-compile machine-code snippets. These are pieces of binary code stored in a raw data file. They are usually created by dumping a running program's image (like firmware) or cutting out parts of an object file (interesting functions, sections, etc.). To process such files, a new step has been added to the decompiler's tool chain. It wraps raw data into an object-file-format container and passes the result to the preprocessing part. The wrapping process and necessary changes made in the front-end to enhance decompilation quality for incomplete inputs are described in Section III.

B. Preprocessing

The preprocessing part analyses the input platform-dependent application to recognise its file format, compiler, and potentially the used packer.
The information about the originally used programming language and compiler is valuable during the decompilation process because each compiler generates a quite unique code in some cases; therefore, such knowledge may increase the quality of the decompilation results. The information about the originally used language may be also used for selecting the output language.
The detection is based on signatures that uniquely identify the used compiler and, therefore, the used language too. Each signature represents a sequence of machine-code instructions that are executed as the first ones during the start of the application.
Whenever we detect a usage of a compression or protection scheme within the decompiled application, we try to unpack-/unprotect it at first by using our plugin-based unpacker. The details can be found in [15].
After that, the preprocessing phase converts the decompiled file into an internal Common-Object-File-Format (COFF)-based representation. This conversion is performed by our plugin-based converter [16, 17]. Currently, it converts from Windows PE, UNIX ELF, Apple Mach-O, and other file formats. Other file formats can be supported by implementing a new plugin. Afterwards, such a COFF file is processed by the decompiler core.

C. Front-End

The decompiler core is built on top of the LLVM Compiler System [18]. 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 parts—a *front-end*, a *middle-end*, and a *back-end*, whose description follows.
Firstly, the front-end processes the input COFF file. This is the only platform-specific part of the decompiler. The processing is done by an instruction decoder, which is automatically generated based on the target architecture model. This model is described by the ISAC architecture description language [19], which is also developed within the Lissom project. The ISAC processor model consists of the following two main parts.

(1) In the resource part, processor resources, such as registers or memory, are declared.

(2) In the operation part, processor instruction set (i.e. assembler language syntax, binary encoding, and behaviour of each instruction) is specified.

From this model, we are able to obtain a sequence of LLVM IR instructions, which represent the semantic behaviour of each processor instruction. The model also contains the binary encoding of instructions. This is used for an automatic generation of the instruction decoder.
The decoder reads binary code, resolves instructions and translates them into LLVM IR instructions. This process transforms platform-specific executable files into uniform L-LVM IR code. The front-end applies various static analyses on the translated LLVM IR code. It checks for the presence of statically linked code to reduce the amount of data to be be analysed. It recovers local variables, used ABI, functions

and their arguments, etc. [11]. Unstripped executable files can be decompiled more easily and precisely by utilizing debugging information or symbols. The decompiler supports both mainstream formats PDB and DWARF [20].
The front-end part is also responsible for the code de-optimization. Its task is to detect the used compiler optimizations and to recover the original high-level language (HLL) code representation from the hard-to-read machine code. One example of this optimization type is the usage of *instruction idioms* [21]. An instruction idiom is a sequence of machine-code instructions representing a short HLL construction (e.g. arithmetic expression) that is highly-optimized for its execution speed and/or size. A notoriously known example is the usage of an exclusive or to clear the content of a register (i.e. `xor reg, reg`) instead of an instruction assigning zero to this register (i.e. `mov reg, 0`). Many other types of compiler-optimized LLVM IR sequences have to be re-arranged and emitted in their de-optimized form to the subsequent decompilation phases. A more detailed description of this process can be found in [22].

D. Middle-End

The LLVM IR code generated by the front-end part contains a complete behavioral description of each instruction used within the original application. In practice, such a complex description may be sizable and slow to analyse in the further decompilation phases. In many cases, however, the input LLVM IR representation can be reduced and optimized. For example, each side-effect of an instruction (e.g. setting a register flag based on instruction operands) is represented via the LLVM IR code, but results of these side-effects may not be used anywhere. Therefore, the front-end output is further processed within the middle-end phase, which is built on top of the LLVM's `opt` tool. This phase is responsible for reduction and optimization of this code by using many built-in optimizations available in LLVM as well as our own passes (e.g. optimizations of loops, constant propagation, removal of dead code, control-flow graph simplifications).

E. Back-End

The target HLL is produced from the optimized IR code by the back-end part. At this moment, we can generate output code in two languages: C and a Python-like language. The latter one is similar to Python, except that specific constructions without a support in Python are replaced by C-like constructions (e.g. pointers and `gotos`). The reason behind choosing a Python-like language as one of the possible target HLLs stems from the fact that Python is dynamically typed and uses just `int` as the basic integral type. This means that the code is not cluttered with explicit types and type casts, which is usual for C code. Moreover, Python uses whitespace indentation, rather than curly braces, to delimit code blocks. Thus, the resulting code may be more readable, which is useful in cases when the code is supposed to be just analysed, not recompiled and run.
The back-end transforms internal input LLVM IR into our own intermediate representation: *backend intermediate representation* (BIR). The reason for an additional internal representation is that LLVM IR is fairly low-level. Therefore, we need a high-level representation that will be used after the identification and reconstruction of high-level control-flow constructs, such as loops and conditional statements.
Finally, the obtained BIR is optimized for the last time, and it is emitted in the form of the target HLL. Other outputs from the back-end are the call graph of the decompiled application, control-flow graphs for all functions, and an assembly representation of the application.

III. Machine-Code Snippet Decompilation

This section describes in detail the concept of machine-code snippet decompilation, introduced in Section II. At first, the wrapping process is explained. Then, the quality-improving changes of the decompiler's front-end are examined. The section is concluded with a discussion of possible drawbacks, imperfections and future improvements of our approach.

A. Wrapping Process

The goal of the wrapping procedure is to create a valid object file from a raw binary input. The generated file is then passed to the preprocessing phase as an ordinary executable, making the whole pre-preprocessing transparent to the rest of the decompilation.
We make use of the GNU objcopy [23] utility from the GNU binutils [24] package. Objcopy is able to copy the contents of one object file to another, modifying it in the process. Supported modifications include various kinds of file header, section or symbol manipulations, which can be used to achieve our goal. The most important capability for our needs is to write a destination object file in a format different from that of the source object file. Given the fact that one of the supported input/output targets is `binary`, the objcopy utility is a suitable utility for the job. To successfully use objcopy, the snippet's original architecture (Intel x86, ARM, MIPS, PIC32, PowerPC), file format (ELF, PE), and endianness (little, big) have to be explicitly specified, since it is impossible to deduce them from the raw data.
The transformation starts by choosing an objcopy variant corresponding to the specified architecture and object file format (there is a stand-alone objcopy for each combination). This allows to select an appropriate output target and machine, knowing that it will be supported by the application. For example, if the user chooses x86/ELF, the output machine and target will be `i386/elf32-i386`. For ARM/PE/little endian, it will be `arm/pei-arm-wince-little`. It would be possible to use a single objcopy compiled with all of the necessary architectures, but using objcopies shipped in the standard-architecture developer's packages seemed more convenient to us. The next process is dependent on the desired target object file format.
For ELF, choosing `binary` as an input target will make objcopy create a new object file with a single section filled with the input's contents. At first, this section is named `.data`, but flags typical for an executable code are set and it is then renamed to `.text`. To increase the quality of the decompiled code, two more object file parameters are always set. The first one is the virtual memory address (VMA) of the created code section. It is a memory address where the section is supposed to be loaded before an actual execution. The correct value plays an important role in case the machine

code contains absolute references (e.g. function jumps) or other constructs relying on the exact memory mapping. In case the user cannot provide the original virtual memory address, the default architecture value used by our compiler is set. In most cases, this will not improve the quality of the output since even the smallest difference has the same negative effect as a huge deviation. However, it is safer than to set a non-zero value that might confuse some programs in the decompiler's tool chain. The load memory address (LMA) is always set identically to the VMA.
The second file parameter is the entry point address. It is the starting address for the execution. In some situations, it can helpful during function detection. The user should provide the original entry point address of the `main()` function, or the position of some other function he wishes to decompile. The value should make sense in the context of given VMA– it should lie within the range defined by VMA as a lower bound, and section's size as a maximum offset. If only one of the load memory address/entry point couple is specified by the user, we use this value for the second parameter as well. If none of them are provided, we set the entry point to the default VMA.
A simplified example of the described transformation of a raw binary `input.raw` into the ELF file `output.elf` is shown in Figure 2.

```
i686_pc_linux_gnu_objcopy input.raw output.elf
  -I binary -O elf32-i386 -B i386
  --set-section-flags
    .data=contents,alloc,load,code,data
  --change-section-vma .data=0x8048000
  --change-section-lma .data=0x8048000
  --set-start 0x8048000
  --rename-section .data=.text
```

Figure. 2: A simplified example of raw binary data to an x86/ELF executable file transformation by using the objcopy utility.

If the same procedure described for an ELF object file were used for PE transformation, PE's NT header entries would remain unset (all values zeroed). Unfortunately, objcopy is not able to manipulate them, making the output file unsuitable for the upcoming analyses. For this reason, a slightly different approach is employed. First, the corresponding objcopy and compiler for the specified architecture are chosen. Then a simple hello world program is compiled by using the selected compiler. Now, it is possible to remove the existing `.text` section from the created executable file and inject a new one. It will be created from the data of the input raw binary file. The original section is removed by using the `-j` parameter, which preserves only the indicated sections. Providing it with a non-existent name will remove all the original sections. The setting of the other file properties is identical as in the ELF case above.
A simplified example of a raw binary `input.raw` injection into the valid PE file `input.exe` producing `output.exe` is shown in Figure 3.

```
arm_mingw32ce_gcc -o input.exe hello.c

arm_mingw32ce_objcopy input.exe output.exe
  -j .non.existent.section
  --add-section .text=input.raw
  --set-section-flags
    .text=contents,alloc,load,code,data
  --change-section-vma .text=0x11000
  --change-section-lma .text=0x11000
  --set-start 0x11000
```

Figure. 3: A simplified example of raw binary data to an ARM/PE/little endian executable file transformation by using the objcopy utility.

B. Front-End Modifications

So far in the front-end development, we have assumed the decompiler is processing a whole object file. This is no longer the case for the machine-code snippet decompilation. If we want to get a good-quality output, we have to take into account the fact that the input may be incomplete. There are the following main issues.

1. Not all present functions have to be reachable from the selected entry point.
2. The function located at the specified entry point may not be called anywhere within the program.
3. Not all functions called in the program have to be defined in the program.

The above three problems are demonstrated on a raw binary code decompilation of a simple function named `function_1()` defined in Figure 4.
The whole application is compiled into a MIPS/ELF/little endian executable file and only the machine code for the selected function is cut out and decompiled. No other function or data sections are present in the reversed raw binary file. The created section is set with the correct VMA, which is equal to the input's entry point. The decompiled result is shown in Figure 5.
The first problem does not occur in this simple example with only one raw function. By default, the decompiler produces only code reachable from the program's start address. This is, however, not a desired behaviour for reversing code snippets since the original functions' connections might not be preserved. The solution is to always analyse the entire input if it comes from a raw binary. Because the original `function_1()` is located at the entry point address, it is named `entry_point()` in the generated HLL code.
The second problem is that it is not called anywhere in the cut raw code. This makes our original function-arguments-detection algorithm inapplicable since it recognises arguments based on the intersection between objects written before the function call, and objects read in the function's body. Because there is no function call, there are no detected arguments. We solve this problem by applying weaker conditions to the detected uncalled functions' arguments. All objects read before they were written (undefined objects) are considered to be arguments. This may, however, cause a significant increase of falsely recognised arguments and it represents a challenge for the future research.

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

int function_2()
{
   return rand();
}

int function_1(int a, int b)
{
   int c = function_2();

   if (c) {
      if (a > 10) {
         printf("case 1");
      }
      else if (b > 10) {
         printf("case 2");
      }
   }

   return 0;
}

int main()
{
   function_1( rand(), rand() );
   return 0;
}
```

Figure. 4: A simple example used for a raw binary decompilation demonstration. Only `function_2()` is cut, wrapped, and reversed.

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

/* ------------ External Functions ----------- */
int  unknown_8900368();           // function_2()
void unknown_89004d4();           // printf()

/* ----------- Function Prototypes ----------- */
int32_t entry_point(int32_t a1, int32_t a2);

/* ---------------- Functions ---------------- */
int32_t entry_point(int32_t a1, int32_t a2)
{
   if (unknown_8900368() == 0) {  // function_2()
      return 0;
   }

   if (a1 >= 11) {
      unknown_89004d4();          // printf()
      return 0;
   }
   if (a2 >= 11) {
      unknown_89004d4();          // printf()
   }

   return 0;
}
```

Figure. 5: The result of the machine-code snippet (compiled for MIPS/ELF/little endian) decompilation for the example in Figure 4.

The third problem is the absence of definitions of called functions `function_2()` and `printf()`. The problem is more serious for the `function_2()` call because the whole `function_1()` body depends on the value set by it. To solve the issue, we modify the front-end so that it treats unknown function calls as external function calls. In this way, procedures do not have to be defined, but their calls are analysed and proper return objects are set. Functions are named by using the pattern `unknown_<call_address>()`. Because there is no data section, strings used at `printf()` calls are not found. As in the previous case, a better detection of arguments for such functions will be addressed in the future research.

C. Possible Drawbacks

The described machine-code snippet decompilation offers only the basic options of object file format creation. It is only possible to transform one input raw data file into a single executable section mapped in memory at a continuous address space. It is, however, possible that users will demand more advanced wrapping capabilities. For example, a creation of multiple sections with different properties (code, data, etc.) and memory layouts (VMA/LMA). If this is the case, our future work will respond to the most popular user requests.

IV. Online Decompilation Service

Our decompiler is not yet available as a stand-alone, downloadable package; however, it is available in the form of an online web service. The home page is located at `http://decompiler.fit.vutbr.cz`. Apart from the decompilation service itself, whose description is given shortly, there are news, a list of our publications, partners, and a contact form.

On the "Try Decompilation" page, there is a form where you can set up a decompilation job. You can either choose to decompile a binary executable file, a machine-code snippet or to provide a C source code, which we first compile and then run our decompiler on the compiled file. Each of these options have different requirements on additional information you have to provide:

- *Binary file.* You can either choose an automatic detection of the file format and architecture or select it manually.
- *Machine-code snippet.* You have to select the target architecture (Intel x86, ARM, MIPS, PIC32, PowerPC), file format (ELF, PE), and endianness (little, big). In addition, you can specify a virtual memory address and an entry point address used by the object file created from the raw binary data. It is strongly recommended to do so since it might significantly improve the quality of the generated HLL code.
- *C source code.* You have to select the target architecture (Intel x86, ARM, MIPS, PIC32, PowerPC), file format (ELF, PE), and compiler options.

As the target language, you can either choose C or a modified version of Python. To start the decompilation process, press the "Decompile" button.

After that, on the results page, you can see a real-time progress bar, including a log of what is happening. After the decompilation is complete, you can view the decompiled code alongside with the output from our disassembler. The

outputs are linked together by addresses and function names, so when you click on a function in the C code, it shows you the corresponding assembly code. Moreover, you can view or download control-flow graphs and a call graph, and download the compiled binary file.

The web service is implemented as a thin client, i.e. you only need a web browser to use it. Internally, it is a PHP application producing XHTML, CSS, and JavaScript. To ensure proper compatibility among various web browsers, we use the jQuery library [25]. To enhance the user's experience during the decompilation, we use AJAX to periodically update the progress bar, log, and links to the results.

We have chosen to implement and provide the decompilation service as an online web service for the following reasons. First, users do not need to install anything on their computer because only an ordinary web browser is needed. Moreover, there are little to no requirements on the user's computer in terms of computational demands because the decompilation runs on a remote server. Second, in the future, our web service may be extended by a public API for starting decompilation jobs and obtaining the results. This will allow users to write custom applications for their mobile devices, such as smartphones and tablets. Finally, in terms of manageability, all application changes can be made directly in the remote server and immediately take effect on the clients. This simplifies periodical updates of the decompilation service and ensures that the clients use the latest stable version that is available.

This online service can be applied in various use cases. Some of them are briefly described next.

- *Static code analysis.* It can help in the analysis of malicious software [10]. As it is shown in Section V, the user can analyse various executable files from untrustworthy websites or vendors. For example, let us suppose that there is a suspicion that an application sends some data over network and it should not do that. It can be processed by the online decompilation service and the decompiled code can be reviewed for a usage of network-related functions like `InternetOpen` on Windows or `connect` on Linux.

- *Source code migration.* The possibility of various output languages gives a good background for code migration [26]. The user can have an application with available assembly code, and after using the service, the user can have the code of the application in C or Python without any difficult rewriting of the assembly code. The decompiler tries to produce code which can be directly recompiled in the case when the output is in the C language. Therefore, one can take an executable file from Linux on the ARM architecture, decompile such an executable file, and use the resulting code on a different architecture. However, we do not take into account some special cases as compilation of source from a decompiled Windows executable file with a Linux compiler. This would be almost surely terminated by some WinAPI function call.

- *Compiler testing and validation.* The user can check the correctness of an executable file compiled by his or her own compiler or by some alternative compiler. In the early stage of compiler development, it can be helpful to verify produced binaries by more alternative ways and the decompilation can be one of them.

- *Embedded software inspection.* With its machine-code snippet decompilation capabilities, it is possible to analyse even programs from which the user do not have the original executable files. This is typical for a firmware loaded in embedded devices. The user can dump the running program's image from memory into a file and upload it to our web service. He has to select the right combination of architecture, format, and endianness because it is not possible to automatically detect them. He should also be able to provide the section's original loading address in virtual memory and the original entry point.

V. Results

The proposed retargetable decompiler is freely available as the previously described decompilation web service. In a typical usage scenario, the user wants to analyse an existing binary executable file. The purpose for this act may differ (e.g. compiler testing, vulnerability detection, malware analysis). All the results (decompiled HLL, disassembled code, graphs, etc.) may be displayed within the web browser or locally after their download. Furthermore, the decompilation process may be tweaked by using many decompilation options to suit the user's needs (elimination of unreachable code, aggressive optimizations, etc.).

The only limitation of this publicly available service is a time limit of the decompilation process (10 minutes for registered users at present). The reason is to be able to serve all user requests. The web service is regularly updated to the latest stable decompiler version. The decompiler currently supports the combinations of the Intel x86, ARM, MIPS, PIC32 and PowerPC architectures and the Windows Portable Executable and UNIX ELF file formats.

Based on the previously published results [11, 15, 27], the decompilation results are comparable with the existing commercial non-retargetable decompilers, such as Hex-Rays decompiler [6]. For example, we achieve over 90% accuracy of successfully recovered functions and 91% of recovered function arguments. Furthermore, the detection ratio of originally used compiler or packer is also high—up to 95%.

In the rest of this section, we present a case study of using the web service to analyse malicious software. Due to space constraints, we demonstrate the decompiler usage on a tiny binary executable file uploaded by one of the users of the decompilation service. Its size is only 2560 bytes, and its MD5 hash is `7ef688b3765b49dc8cd0dd2c4bf79fd0`. When one tries to analyse this executable file manually by using standard tools, he or she discovers that it is a program in the PE format for the x86 architecture (32b).

The disassembled code of its `.text` section is shown in Figure 6. It was produced by the `objdump` utility, which is part of the GNU Binutils package. On the first sight, there is nothing that would indicate malicious behaviour. What we can see is that there are several function calls, and their arguments are passed on the stack. However, as the called functions are not present in the executable file nor there are

symbols for them, we cannot see which functions are actually called. Moreover, several addresses are passed on the stack, and there is no clear indication of what is stored at these addresses. To obtain more information, additional analyses are required to be done.

```
Address  Hex dump           Intel x86 instruction
-----------------------------------------------
401000:  6a 00              push  $0x0
401002:  6a 00              push  $0x0
401004:  68 28 30 40 00     push  $0x403028
401009:  68 1f 30 40 00     push  $0x40301f
40100e:  6a 00              push  $0x0
401010:  6a 00              push  $0x0
401012:  e8 39 00 00 00     call  0x401050
401017:  6a 01              push  $0x1
401019:  68 4b 30 40 00     push  $0x40304b
40101e:  68 00 30 40 00     push  $0x403000
401023:  e8 1c 00 00 00     call  0x401044
401028:  6a 00              push  $0x0
40102a:  6a 00              push  $0x0
40102c:  6a 00              push  $0x0
40102e:  68 4b 30 40 00     push  $0x40304b
401033:  6a 00              push  $0x0
401035:  6a 00              push  $0x0
401037:  e8 14 00 00 00     call  0x401050
40103c:  6a 00              push  $0x0
40103e:  e8 07 00 00 00     call  0x40104a
401043:  cc                 int3
401044:  ff 25 04 20 40 00  jmp   *0x402004
40104a:  ff 25 00 20 40 00  jmp   *0x402000
401050:  ff 25 0c 20 40 00  jmp   *0x40200c
```

Figure. 6: Disassembled code of the `.text` section obtained via the `objdump` utility.

When the executable file is uploaded to our decompilation service, its decompilation takes about two seconds, and produces the output shown in Figure 7. Due to space constraints, we have stripped all the comments from the code.

```
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <windows.h>

int main(int argc, char **argv)
{
   ShellExecuteA(0, NULL,
      "iexplore",
      "http://91.188.59.10/report/log.php",
      NULL, 0
   );
   CopyFileA("\\\\91.188.59.10\\smb\\sweater.exe",
      "1.exe", true
   );
   return ShellExecuteA(0, NULL, "1.exe",
      NULL, NULL, 0
   );
}
```

Figure. 7: The result produced by the decompilation service.

From this output, it is clear what the program actually does. First, it starts MS Internet Explorer and visits its site `http://91.188.59.10/report/log.php`. Even though the site is unavailable at the time of writing this paper, it apparently reports to its author the fact that is has been executed. Next, it obtains a file `sweater.exe` from a remote server, names the copy `1.exe`, and executes it. After that, it exits. The server is also not available at the time of writing.

From this analysis, we can clearly see that the program can be considered as having malicious behaviour. And indeed, when we check this by uploading the file to VirusTotal [28], we see that 27 from 47 anti-virus tools have classified it as malware.

Apart from the decompiled code, the web service produces a call graph of the executable file, shown in Figure 8, control-flow graph of the `main` function, and disassembled output. The control-flow graph and disassembled output are not included in this paper due to space constraints.

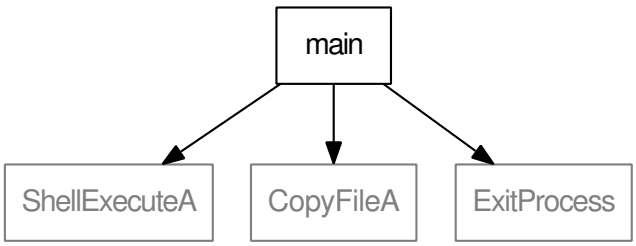

Figure. 8: The call graph of the program.

Moreover, the results from the web service tell us that the program is for the x86 architecture (32b), and that is was produced by `fasm` [29]. A screenshot of the web page is depicted in Figure 9.

VI. Concluding Remarks

As can be seen from the present paper, the main advantages of our solution when compared to the existing decompilers are retargetability (new architectures can be supported more rapidly [10, 11]) and the fact that the decompilation and analysis can be done by simply using a web browser.

We close the paper by suggesting future research possibilities. First, many todays applications are written in C++. To decompile such applications, we will need yet more advanced analyses concerning the reconstruction of types, class hierarchies, and use of the standard library, including the standard template library (STL). Second, the decompilation of more sophisticated malware poses a challenge that we will face in the near future [30].

Acknowledgments

This work was supported by the BUT FIT grant 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] E. Eilam, Reversing: Secrets of Reverse Engineering. Wiley, 2005.

[2] International Data Corporation (IDC), “Worldwide quarterly mobile phone tracker,” 2011.

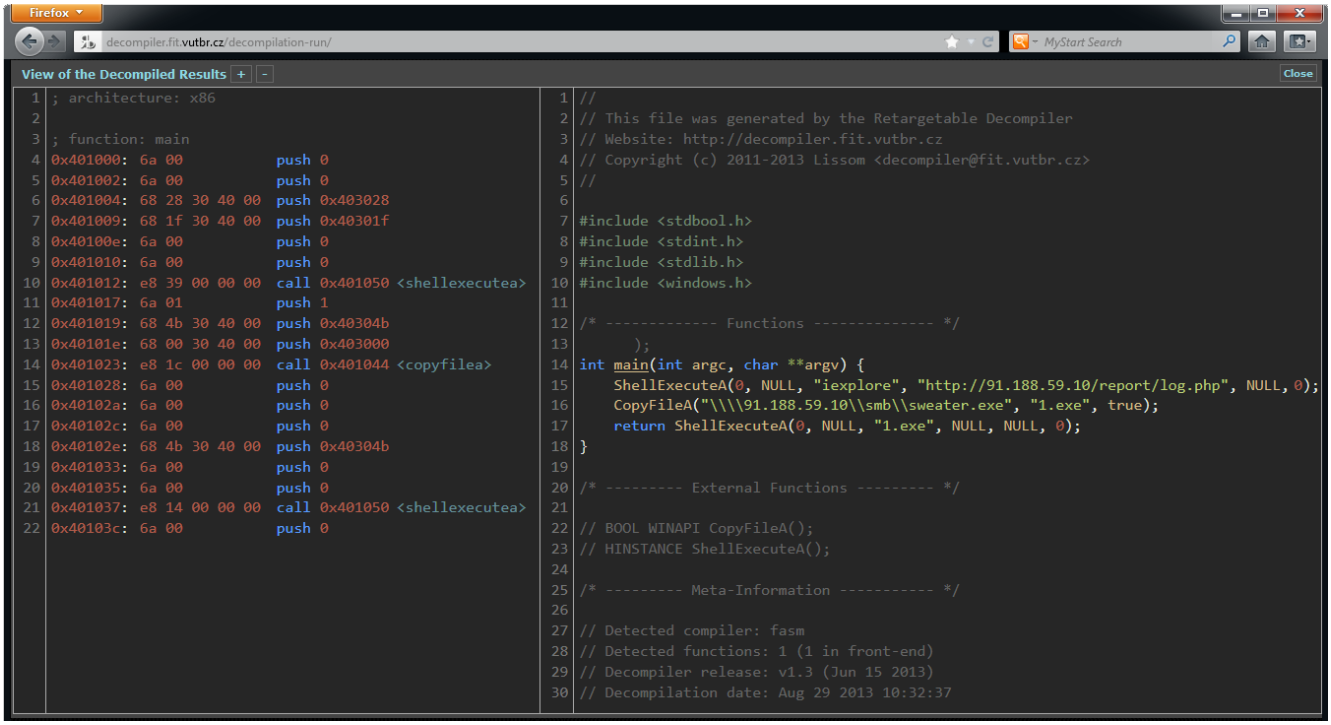

Figure. 9: A simplified screenshot of the page displaying the code generated by the web service.

[3] Boomerang, http://boomerang.sourceforge.net/, 2014.

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

[5] M. van Emmerik and T. Waddington, "Using a decompiler for real-world source recovery," in Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04). Washington, DC, USA: IEEE Computer Society, 2004, pp. 27–36.

[6] Hex-Rays Decompiler, www.hex-rays.com/products/decompiler/, 2014.

[7] SmartDec, http://decompilation.info/, 2014.

[8] Reverse Engineering Compiler (REC), http://www.backerstreet.com/rec/rec.htm, 2014.

[9] Lissom, http://www.fit.vutbr.cz/research/groups/lissom/, 2014.

[10] 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), vol. 5, no. 4, 2011, pp. 91–106.

[11] 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). Kingston, ON, CA: IEEE Computer Society, 2012, pp. 51–60.

[12] L. Ďurfina, J. Křoustek, and P. Zemek, "Retargetable machine-code decompilation in your web browser," in 3rd IEEE World Congress on Information and Communication Technologies (WICT'13). Hanoi, VN: IEEE Computer Society, 2013, pp. 57–62.

[13] 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). North Atlantic University Union, 2011, pp. 199–204.

[14] J. Křoustek and D. Kolář, "Context parsing (not only) of the object-file-format description language," Computer Science and Information Systems (ComSIS), vol. 10, no. 4, 2013, pp. 1673–1702.

[15] ——, "Preprocessing of binary executable files towards retargetable decompilation," in 8th International Multi-Conference on Computing in the Global Information Technology (ICCGI'13). Nice, FR: International Academy, Research, and Industry Association (IARIA), 2013, pp. 259–264.

[16] 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). Ministry of Education, Science, Youth and Sports of Ukraine, Lviv Polytechnic National University, Institute of Computer Science and Information Technologies, 2011, pp. 127–130.

[17] J. Křoustek and D. Kolář, "Object-file-format description language and its usage in retargetable decompilation," in AIP Conference Proceedings (SCLIT'12), vol. 1479. American Institute of Physics (AIP), 2012, pp. 466–469.

[18] The LLVM Compiler Infrastructure, http://llvm.org/, 2014.

[19] K. Masařík, System for Hardware-Software Co-Design, 1st ed., ser. VUTIUM. Brno, CZ: Brno University of Technology, Faculty of Information Technology, 2008.

[20] 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). International Academy, Research, and Industry Association (IARIA), 2012, pp. 79–84.

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

[22] J. Křoustek and F. Pokorný, "Reconstruction of instruction idioms in a retargetable decompiler," in 4th Workshop on Advances in Programming Languages (WAPL'13). Krakow, PL: IEEE Computer Society, 2013, pp. 1507–1514.

[23] GNU objcopy, https://sourceware.org/binutils/docs/binutils/objcopy.html, 2014.

[24] GNU Binutils, http://www.gnu.org/software/binutils/, 2014.

[25] jQuery – Fast, Small, and Feature-Rich JavaScript Library, http://jquery.com/, 2014.

[26] L. Ďurfina, J. Křoustek, and P. Zemek, "Generic source code migration using decompilation," in 10th Annual Industrial Simulation Conference (ISC'2012). EUROSIS, 2012, pp. 38–42.

[27] L. Ďurfina, J. Křoustek, P. Zemek, and B. Kábele, "Accurate recovery of functions in a retargetable decompiler," in 15th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'12), ser. LNCS 7462. Berlin, Heidelberg, DE: Springer-Verlag, 2012, pp. 390–392.

[28] VirusTotal – Free Online Virus, Malware and URL Scanner, https://www.virustotal.com/, 2014.

[29] Flat Assembler, http://flatassembler.net/, 2012.

[30] P. Ször, The Art of Computer Virus Research and Defense. Upper Saddle River, US-NJ: Addison-Wesley, 2005.

Author Biographies

Lukáš Ďurfina (born in Nitra, Slovakia, 1986) is a Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his MSc degree from the same university in 2010. He worked as a front-end developer of the retargetable decompiler. His current research interests include reverse engineering and malware detection.

Jakub Křoustek (born in Jihlava, Czech Republic, 1984) is a Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his MSc degree from the same university in 2009. He is currently working on the Lissom research project as the leader of the retargetable decompiler. His current research interests include reverse engineering, malware detection, and compiler design, with special focus on code analysis and reverse translation.

Peter Matula (born in Martin, Slovakia) is a Ph.D. student at the Faculty of Information Technology, Brno University of Technology, Czech Republic. He received his MSc degree from the same university in 2013. He is currently working as a front-end developer of the retargetable decompiler. His current research interests include reverse engineering and malware detection, with focus on high-level data type reconstruction.

Petr Zemek (born in Opava, Czech Republic) is a developer of the retargetable decompiler. He received his Ph.D. in computer science from the Faculty of Information Technology, Brno University of Technology, Czech Republic in 2014. He has published many studies on formal models and decompilation in international conferences and distinguished computer science journals, including two books on regulated rewriting in formal language theory.