Psyb0t Malware: A Step-by-Step Decompilation Case Study

Lukáš Ďurfina, Jakub Křoustek, Petr Zemek
Faculty of Information Technology, IT4Innovations Centre of Excellence,
Brno University of Technology, Božetěchova 1/2, 612 66 Brno, Czech Republic
{idurfina, ikroustek, izemek}@fit.vutbr.cz

***Abstract*—Decompilation (i.e. reverse compilation) represents one of the most toughest and challenging tasks in reverse engineering. Even more difficult task is the decompilation of malware because it typically does not follow standard application binary interface conventions, has stripped symbols, is obfuscated, and can contain polymorphic code. Moreover, in the recent years, there is a rapid expansion of various smart devices, running different types of operating systems on many types of processors, and malware targeting these platforms. These facts, combined with the boundedness of standard decompilation tools to a particular platform, imply that a considerable amount of effort is needed when decompiling malware for such a diversity of platforms.**

This is an experience paper reporting the decompilation of a real-world malware. We give a step-by-step case study of decompiling a MIPS worm called *psyb0t* by using a retargetable decompiler that is being developed within the Lissom project. First, we describe the decompiler in detail. Then, we present the case study. After that, we analyse the results obtained during the decompilation and present our personal experience. The paper is concluded by discussing future research possibilities.

***Index Terms*—Reverse engineering, decompilation, retargetable decompiler, Lissom, malware, psyb0t, experience**

I. Introduction

Decompilation (i.e. reverse compilation) represents one of the most toughest and challenging tasks in reverse engineering. Its difficulty stems from a loss of information during compilation, existence of many different file formats, architectures, programming languages, compilers, and last, but certainly not least, from the fact that many problems that arise during decompilation have been proven as unsolvable in general.

Decompilation of malware represents an even more difficult task [1, 2], which is frequently done by security companies to inspect the behaviour of such malicious software. The reason of the increased difficulty is that malware typically does not follow standard application binary interface (ABI) conventions, has stripped symbols, is obfuscated, and can contain polymorphic or metamorphic code [3, 4].

For the past 20 years, malware was primarily targeted at personal computers (i.e. architectures Intel x86 and x86-64). The techniques for malware analysis were well optimized for this platform during this time and security companies were able to keep pace with malware authors [3, 4]. However, the expansion of smart devices (e.g. smartphones, tablets, routers) is very rapid in the last years [5]. Such devices are powered by various processors and running several types of operating systems. Users use these platforms for manipulating sensitive user data (e.g. passwords, credit card numbers), which comes to the attention of malware authors. Furthermore, the variety of these platforms is problematic for security companies because their solutions are mostly oriented on the classical ones, and they do not fully protect new platforms at the moment. Those are the main reasons why the amount of malware for these platforms increases steadily for the last years.

To help with a platform-independent malware analysis, we have proposed the concept of a retargetable decompiler [6]. This tool is being developed within the Lissom project [7] at Brno University of Technology, Czech Republic, and aims to be independent of any particular file format, target architecture, and operating system. Currently, the decompiler supports the decompilation of MIPS, ARM, and Intel x86 executables in the UNIX ELF and Windows PE file formats. The output language is either C or a Python-like language.

In [6] and in our successive papers [8, 9], we have only considered the design of the decompiler and its components, without a detailed study of its applicability in practice. This brings us to the topic of the present paper, which fills this gap. Indeed, the present paper gives a step-by-step case study of decompiling a computer worm called *psyb0t* [10]. This worm targets modems and routers with MIPSel [11] processors running on the Linux-based systems and creates a botnet operated by IRC (Internet Relay Chat) command-and-control (C&C) servers. It received quite an attention during its discovery in January 2009 [10, 12, 13].

This is an experience paper reporting the decompilation of a real-world malware. Such a study is rarely attempted, and seldom reported in the literature [14]. The reason for choosing this particular malware is that its targets (modems and routers with MIPS processors) are outside the mainstream. This makes the situation difficult (if not impossible) for the "standard" decompilers, like Hex-Rays [15], because they do not support such targets. At the same time, as shown later in the present paper, the developed retargetable decompiler succeeds in the decompilation. We would also like to point out that analysis using decompilation is much easier than the previous attempts described in [12, 13]. Hence, the present paper shows the abilities of the decompiler in a practical scenario.

The remainder of this paper is organized as follows. After this introductory section, Section II describes the Lissom project's retargetable decompiler in detail. Then, in Section III,

978-1-4799-2931-3/13 © 2013 IEEE

we present the case study of decompiling the psyb0t malware. Section IV analyses the results obtained during the decompilation and presents our personal experience. Finally, Section V concludes the paper by discussing future research possibilities.

II. Lissom Project Retargetable Decompiler

The Lissom project [7] retargetable decompiler aims to be independent of any particular target architecture, operating system, or file format [6]. It consists of two main parts—the *preprocessing part* and the *decompiler core*, see Figure 1.

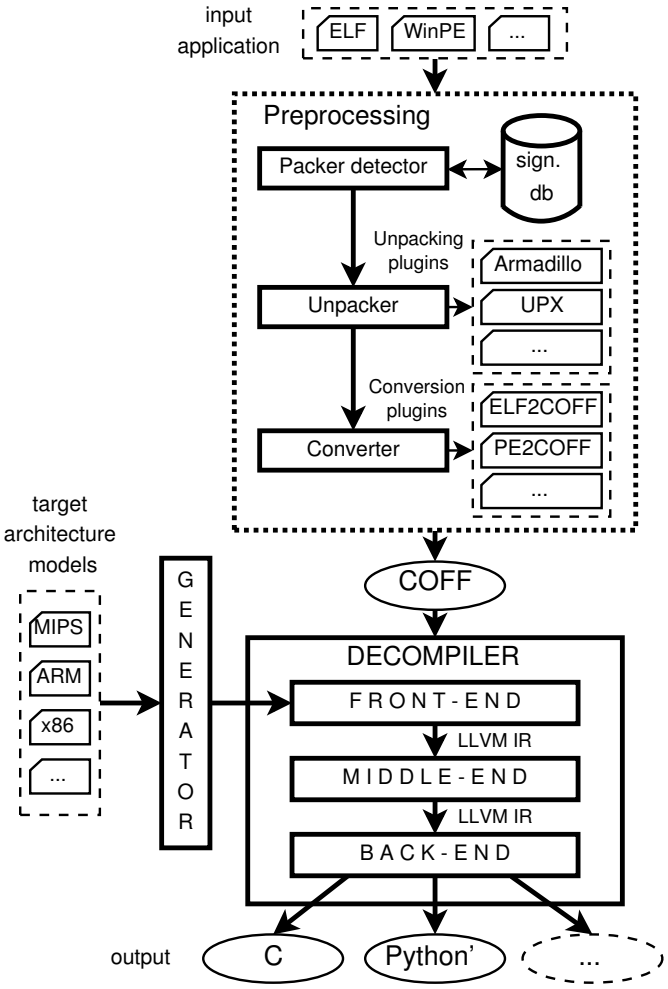

Fig. 1. The concept of the retargetable decompiler.

The preprocessing part analyses the input application to detect the used file format, compiler, and, if the file was packed, the used packer. A detailed description of this process is given in [8]. After that, it unpacks and converts the examined platform-dependent application into an internal Common-Object-File-Format-based representation (COFF). 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 [8]. Afterwards, such COFF files are processed by the decompiler core.

The decompiler core is built on top of the LLVM Compiler System [16]. 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 COFF files are first 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). In our decompiler, we use the ISAC ADL [7], which is also developed within the Lissom project. The ISAC processor model consists of two essential 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. The ISAC model is transformed by a *semantics extractor* [17], 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 behaviour. 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 behaviour 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 etc. [9]. 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 two target HLLs: C and a Python-like language. The latter 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.

III. PSYB0T DECOMPILATION

In this section, 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* [10], which attacks network infrastructure devices (e.g. modems and routers) running MIPS processors with Linux-based operating systems. The following text describes all the major decompilation phases with illustrations.

A. Initial Recognition Using the Third-Party Tools

The size of the examined binary file is 29,264 bytes and its MD5 hash is `58f00c14942cae1e9f24b03d55cd295d`. This is the latest known version of this malware[1]. As will be discussed later, it marks itself as "`PSYB0T v2.9L`". The previous mentioned articles about psyb0t analysis were focused mainly on the older version `2.5L` [10].

The very first step of an initial analysis is detection of the file format and the target platform. The file starts with an identifier "'`0x7f'ELF`"'. In other words, it is the ELF file format [19] that is used on UNIX-based systems. Therefore, we can obtain additional information by using standard Linux tools like `readelf` or `objdump`. Relevant parts of the former one's output are shown in Figure 2.

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

Fig. 2. Information about the (packed) executable file gathered by using the `readelf` utility.

As can be seen from the output, it is an executable file for the 32-bit MIPS architecture and it uses the little-endian encoding (this architecture is explicitly called MIPSel). Its entry point address (i.e. address of the first instruction executed during the application run-time) is atypical because it is usually placed somewhere nearby `0x08000000`; the section and symbol tables are empty, which is also unusual but correct. The information about the originally used compiler is usually stored in the optional `.comment` section, but this file lacks such a section. Moreover, the file content is also atypical because there are no visible strings, such as symbol names, section names, or strings for user interaction during run-time.

Based on these clues, we can guess that the file is packed and maybe obfuscated by some packer or protector. In comparison with Windows, the number of Linux packers is very limited (e.g. gzexe, Elfcrypt, UPX, and HASP). A detection of the used packer is difficult because the existing packer detectors (e.g. PEiD, ProtectionID, Exeinfo PE) do not support the ELF format and its packers, see [8] for details. In the classical approach, presented in [10, 12, 13], it is necessary to distinguish the used packer manually, unpack it, and analyse it by using a MIPS disassembler. Luckily, our retargetable decompiler can handle such situation automatically. A detailed description of the decompilation process follows.

[1]It should be noted that psyb0t has successors, like the Chuck Norris botnet [18].

B. Preprocessing Phase

The analyses done in the previous subsection are usually used when inspecting malware manually. We do not need any of the above-mentioned third-party software to perform such analyses. Indeed, our decompiler performs them automatically by itself so no manual intervention is needed. In a greater detail, the first part of the decompilation process begins at our file-information-gathering application called `fileinfo`. It obtains the same information as the `readelf` utility does in Figure 2 but independently on the used target format (i.e. it supports ELF, Windows PE and other common formats). Another its advantage is a built-in packer/compiler detector.

The detection algorithm is based on pattern matching of the entry-point instructions with an internal signature database. The sequence of the entry-point instructions for the psyb0t malware starts at file offset `0x6828` and contains sequence "`e00011040000f7272028a4000000e6ac00800d3c`"; its translation to the MIPS machine code is illustrated in Figure 3.

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

Fig. 3. Entry-point instructions of the UPX packed code (little-endian encoding).

This sequence is matched with the internal signature for the MIPSel/ELF UPX packer [20] of the shortened little-endian form "`---11040000f7272028a4000000e6ac`". Symbol '-' denotes a variable part—in this case an immediate value of the conditional branch instruction `bal`.

Therefore, we figured out that the UPX packer for the MIPS architecture was used for application packing. The used version of UPX was 3.03 and this was the up-to-date version when the malware started spreading. In normal circumstances, we are able to unpack such a file by using our internal plugin-based unpacker, see [8] for details. The UPX unpacking plugin is trivial—it simply invokes the UPX packer with argument `-d`. This argument switches UPX's behaviour to unpacking mode. However, this input file was manually modified to disable this form of unpacking.

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

are mandatory for detection by UPX. The third part laying at file offset (0x6848) is not necessary for the detection and it originally contained the following string:

```
$Info: This file is packed with the UPX executable
packer http://upx.sf.net $ $Id: UPX 3.03 Copyright
(C) 1996-2008 the UPX Team. All Rights Reserved. $
```

The unpacking of such a file can be done in two ways. (1) Execute the application in a MIPS emulator and break execution after the UPX decompression routine is done and the original entry point is hit. Afterwards, dump the memory content to disk and reconstruct the ELF file by using this memory dump. Every step of this process can be done automatically without a user interaction. Retargetability can be preserved via the concept of a retargetable simulation, presented in [21]. (2) Manually patch the three missing "UPX!" strings and use UPX for unpacking.

The first method is marked as our future research but unavailable yet. Therefore, we have to manually modify the file by using the second method. This is the only manual interaction needed during the complete decompilation process of this file. The modified file can be easily unpacked via the `upx -d` command. The unpacked file size is 127,892 bytes that gives us a 22.88% compression ratio. The unpacked file contains 20 sections, 133 symbols, and several hundred strings.

The last part of the preprocessing phase is a conversion of the unpacked ELF file into an internal COFF based format. This is done by using our another plugin-based application as illustrated in Figure 1.

C. Front-End Phase

Next, the unpacked psyb0t application in the COFF format is processed in the front-end phase. At first, as was mentioned in Section II, 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 COFF into LLVM IR platform-independent representation that is further processed by the following analyses.

In the front-end phase, various analyses are applied, but in what follows, we focus only on those related to our subject. This means that we exclude, for example, a description of analysis that reads DWARF debugging information from the executable because psyb0t does not contain any DWARF data.

Firstly, we process the whole executable to reveal data as strings. This is important for later usage of these strings in function calls. It is implemented by analysing data sections. The analysis tries to find a sequence of printable characters terminated by the zero byte. Such a sequence is marked as a string and its address is stored. If we detect an access to this address, we know that it uses a specific string and we have the value of that string.

The executable contains also symbols for functions. As we will see later, it does not have the symbols for all functions, but we can use the available symbols to improve the decompilation results. This analysis is simple and just stores the pairs with the address and name of each symbol, see Table I for illustration. Since there is a symbol for the `main` function, we can skip the entry point analysis. If the executable was without that symbol (i.e. stripped), this analysis would try to find the address of `main` by using its internal compiler-specific database or by using a heuristic detection.

TABLE I
SHORTENED LIST OF FUNCTIONS EXTRACTED FROM SYMBOLS.

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

The studied executable is for the MIPS architecture, which utilizes so-called delay slots. A *delay slot* is an instruction slot that is executed together with the previous instruction. On MIPS, a delay slot is located immediately after every branch instruction. The architecture description in the ISAC language supports setting the latency of instructions. If this latency is equal to 2, the instruction is followed by a single delay slot.

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

The next analysis is aimed on creating a control-flow graph (CFG). It examines all branch instructions, tries to get the target addresses and resolve the type of branches. The goal is to recognize 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 [9] for details. We illustrate how the interpreter works on the piece of psyb0t code listed in Figure 4.

On address `0x4105c8`, there is a call of a function whose address is stored in register `t9`. Therefore, we call the interpreter to track this register and find its value. The interpreter goes backwards in the CFG and identifies the write of a value into `t9` on address `0x4105c0`. The written value is read from memory on offset `-32268` from value of the `gp` register. Next, the interpreter has to get the value of `gp`. This register is written on the beginning of the function. Therefore, it is not a problem to find it by traversing the CFG. The value is given by

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

Fig. 4. Example of static code interpretation.

the following expression: `0x0fc0 « 16 - 30884 + t9`. The current function is called by register `t9`, so the interpreter uses the address of the current function in this expression. It has the value of `gp`. It subtracts `32268` from this value and the result is address of memory, where the final value is stored.

After control-flow analysis, we can detect functions. As we mentioned before, the executable under decompilation has symbols, but this analysis is run nevertheless because the set of symbols can be incomplete. This is also that case. The number of available function symbols is 34, but the overall number of detected functions is 91. This can be caused by linked code from libraries without symbols or by special compiler routines. The detection of functions is realized by our algorithms that were presented in [9].

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

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

D. Middle-End Phase

In this stage, we have a very low-level LLVM IR of the input binary. Each basic block represents a single assembly instruction, and there may be many redundant instructions (recall that each assembly instruction is decompiled in isolation). The key role of the middle-end part of our decompiler is to optimize the input LLVM IR code and prepare it for the back-end.

For example, consider the block in Figure 5, which was generated by the front-end for the instruction `jalr t9` on address `0x41a088`.

As described in Section III-C, `jalr` is an indirect branch to an address stored in a register, and a store of the return address in another register. By using our interpreter and the import table from the executable file, we were able to detect that the branch is actually a call to the function `usleep` from `<unistd.h>`. However, due to generality, a lot of boilerplate code has to be emitted along with the call, which is optimized in the middle-end. The block from Figure 5 after optimizations can be seen in Figure 6.

```
%u0_41a088 = add i32 4300940, 0
%_c_41a088 = add i32 4, 0
%u1_41a088 = add i32 %u0_41a088,  %_c_41a088
%_e_41a088 = add i32 31, 0
%u2_41a088 = load i32* @gpregs25
%u0_ds_41a088 = add i16 119, 0
%u1_ds_41a088 = sext i16 %u0_ds_41a088 to i32
store i32 %u1_ds_41a088, i32* @gpregs24
%arg1049_41a088 = load i32*  @gpregs4
%r_41a088 = call i32 @usleep(i32 %arg1049_41a088)
store i32 %r_41a088, i32*  @gpregs2
```

Fig. 5. A block generated by the front-end for the instruction `jalr t9` on address `0x41a088`.

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

Fig. 6. The block from Figure 5 after optimizations.

The `arg1` variable is actually the name of the parameter of the function, in which this call to `usleep` appears. It should be noted that the optimizer in the middle-end takes into account also the surrounding blocks so it performs the optimization globally, not just locally over a single block.

E. Back-End Phase

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

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

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

their symbolic names, and we are able to utilize it to improve the readability of the generated code.

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

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

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

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

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

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

IV. 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 [13].

There are two known versions of psyb0t. We have decompiled the newer one, which identifies itself as `[PRIVATE] PSYB0T v2.9L`. This version is better secured against unpacking by UPX and it affects more network devices, mainly models by Linksys, Netgear, and other routers running DD-WRT or OpenWrt firmware. The application is written in the C language. This can be spotted by the names of called functions and also by the usage of position independent code, which can be simply turned on by flag `fPIC` of the GNU `gcc` compiler.

In this section, we introduce a brief description of psyb0t's behaviour by using snippets of code from the decompiler in order to show how the decompiler is useful for faster analysis of malware.

In the previous section, we have presented the whole decompilation process in a step-by-step way, and we have shown the code from our own decompiler. Now, we can analyse the obtained HLL source code. We describe the behaviour of psyb0t immediately after its execution, i.e. the code starting at the entry-point—the `main` function.

Firstly, we can take a look on the call graph. It is good for a fast detection of relations between functions. A part of that graph is shown in Figure 8. A complete call graph is omitted due to space constraints. The most important parts of the `main` function are listed in Figure 7. The comments were added manually. Selected parts are listed separately with describing notes.

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

Fig. 7. Simplified code of the `main` function by using the Lissom project retargetable decompiler.

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

```
uint32_t *file = fopen("/var/tmp/udhcpd.mtx", "w");
var3 = (uint32_t)file;
if (file == NULL) {
    return 1;
}
```

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

```
uint32_t fd = fileno(file);
if (fd == -1) {
    var3 = 1;
    return 1;
}
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`,

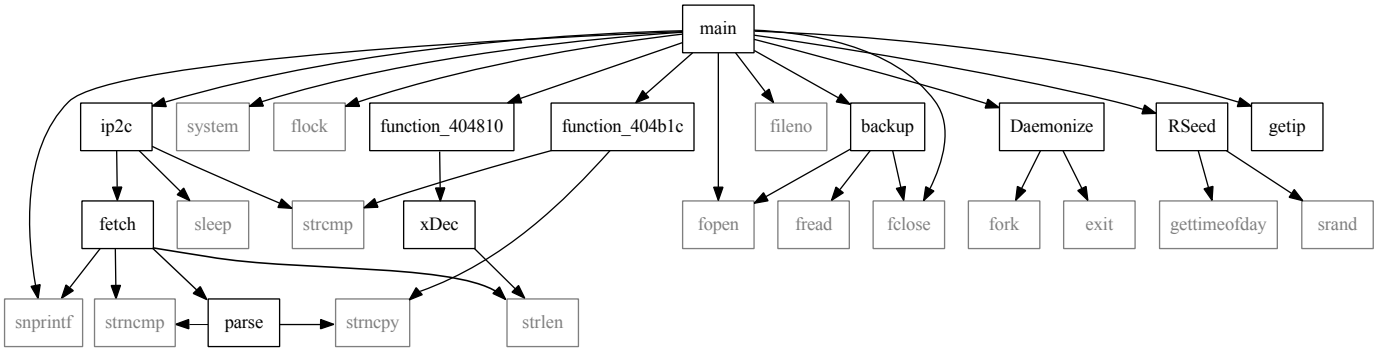

Fig. 8. A part of the call graph for `main`. Nodes in black are user-defined functions while grey nodes denote external functions.

it uses `fd` as a file descriptor, and finally, for `flock`, it uses `err_code`. We can take a closer look on the call of `flock`. The original second argument is `6`, but the back-end is able to find out the names of the symbolic constants that form this value.

If the lock is acquired, the application calls internal function `RSeed`, which initializes the pseudo-random generator of numbers by calling `srand`. An important call is that of function `Daemonize`, where the application is forked and the parent process is terminated. The child process continues in its execution on background with starting and setting a firewall, and removing itself from the file system. The second call of `system` updates firewall rules to drop all the packets on `tcp` port `23` (i.e. disable inbound telnet communication). The third command removes the file that psyb0t uses for spreading, probably to cover its tracks. After removal, psyb0t is located only in memory and a reset of the infected device will disinfect it. The executed shell commands are of the following form:

```
/etc/firewall_start
iptables -A INPUT -p tcp --dport 23 -j DROP
rm -f /var/tmp/udhcpc.env
```

Afterwards, the memory-located psyb0t backups the file `/var/tmp/hosts` inside the `backup` function and reports itself to the C&C IRC channel naming itself as a regular expression `\[NIP\]-[A-Z0-9]{9}` (inside `function_404b1c`). The last nine symbols are generated randomly as an index to string `"0123456789 ABCDEFGHIJKLMNOPQRSTUVWXYZ"` using the previously initialised pseudo-random generator.

In Section III-C, we have described the data section analysis. It provides us an array with strings that are the names of commands which are accepted by psyb0t. These commands are received from the topic of the connected IRC channel.

```
const char *STRINGS[] = {
  "mode", "login", "logout", "_exit_", "sh",
  "tlist", "kill", "killall", "silent", "getip",
  "visit", "scan", "rscan", "sleep", "sel", "esel",
  "rejoin", "upgrade", "wupgrade", "ver", "wget",
  "lscan", "rlscan", "getinfo", "rsgen", "vsel",
  "split", "gsel", "sflood", "uflood", "iflood",
  "pscan", "fscan", "r00t", "sql", "pma", "socks",
  "rsloop", "report", "uptime", "usel", "spoof",
  "viri", "smb", "cgen"
};
```

Some of these strings are the same as names of the reconstructed functions and we can presume that such functions implement these commands.

One of the commands is `fetch` and we have a function with the same name. If we take a look at it, we can see the following code:

```
snprintf((uint8_t *)&var_9, 5120,
  "GET /servlet/view/banner/javascript/zone?zid=81&
  pid=0&random=%d&millis=%lu HTTP/1.1\r\nHost: %s\r
  \n%s%sReferer: %s\r\n\r\n",
  var_18, var_20, (uint8_t *)&var_12, (uint8_t *)
  &var_13, (uint8_t *)&var_16, (uint8_t *)&var_17);
len = strlen((uint8_t *)&var_9);
dpage(-23184, (uint16_t)var_9, 0, 1);
```

There is a preparation of an `HTTP` command that is used in internal function `dpage` that uses standard functions `socket`, `connect`, `send`, and `recv` for network communication. Psyb0t uses a timeout by registering a function for handling `SIGALRM`. Before `connect`, there is a call `alarm(3)` to wait at most three seconds for connection, and before `recv`, there is `alarm(12)`.

During its run-time, psyb0t loops in `function_4056cc` awaiting for other commands obtained either from IRC channel topic or through a private massage. Commands `scan`, `rscan`, `lscan`, `rlscan`, `pscan`, and `fscan` tell psyb0t to scan for other vulnerable devices and try to spread itself to them (as described in the beginning of this section).

Finally, in Table II, we provide some statistics about the output from the decompiler. The result in the Python-like language is shorter because it does not use types. The size of both files is quite large, which is caused by the generation of many assignments, where some of them are not needed. In the future, we plan to improve our optimization algorithms in the back-end part to remove more such code and produce even more readable output (see the notes in Section V).

TABLE II
STATISTICS ABOUT DECOMPILER OUTPUT FOR PSYB0T.

Feature	Value
Internal functions count	91
External functions count	57
Function calls	1278
C output size	553 kB
Python-like output size	453 kB

V. Conclusion

In this paper, 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. From Section IV, we see that by using our decompiler, we are able to speedup the analysis of malware because we deal with high-level code (cf. [12, 13], where only the output from a disassembler is used, which requires many additional analyses to be done).

As outlined in Section II, the front-end part of the decompiler is automatically generated based on a processor model in the ISAC language. Moreover, our plugin-based converter supports an easy addition of new file formats. Therefore, the decompiler is not bound to a particular platform. To add support for a new platform, we do not have to create an entirely new decompiler. Rather, we just describe the new platform in ISAC, add custom passes and analyses into the front-end, and reuse the existing parts of the decompiler. This greatly simplifies the addition of new platforms, which becomes really handy in terms of the rapid expansion of various smart devices.

In terms of the present paper's topic, we suggest the following three areas for future research. First, we are designing a retargetable unpacker that emulates the input packed executable file and dumps the memory content to disk after the original entry point was hit. The emulator will be based an existing retargetable simulator [21]. The basic idea was already presented in [8]. Second, by improving our copy propagation optimization, we would get rid of more assignments, which are present in the current code (see the code snippets in Section IV). This would cut down the size of the generated output and improve the readability even more. Third, it would be interesting to compare the difficulties of decompiling malware for other architectures, like ARM and Intel x86. Such a topic is out of scope of the present paper.

Acknowledgments

This work was supported by the project TA ČR TA01010667 System for Support of Platform Independent Malware Analysis in Executable Files, BUT grant FEKT/FIT-J-13-2000 Validation of Executable Code for Industrial Automation Devices using Decompilation, BUT FIT grant FIT-S-11-2, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

References

[1] G. Wagener, R. State, and A. Dulaunoy, "Malware behaviour analysis," *Journal in Computer Virology*, vol. 4, no. 4, pp. 279–287, 2008.

[2] C. Willems, T. Holz, and F. Freiling, "Toward automated dynamic malware analysis using CWSandbox," *IEEE Security and Privacy*, vol. 5, no. 2, pp. 32–39, 2007.

[3] P. Ször, *The Art of Computer Virus Research and Defense*. US-NJ: Addison-Wesley, 2005.

[4] J. Aquilina, *Malware Forensics Investigating and Analyzing Malicious Code*. Burlington, US-MA: Syngress Publishing, 2008.

[5] International Data Corporation (IDC), "Worldwide quarterly mobile phone tracker," 2011.

[6] 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, pp. 91–106, 2011.

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

[8] J. Křoustek and D. Kolář, "Preprocessing of binary executable files towards retargetable decompilation," in *ICCGI'13*. Nice, FR: International Academy, Research, and Industry Association (IARIA), 2013, pp. 259–264.

[9] 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.

[10] T. Baume, "Netcomm NB5 botnet – psyb0t 2.5L," http://www.baume.id.au/psyb0t/PSYB0T.pdf?info=EXLINK, 2009.

[11] MIPS Technologies Inc., *MIPS32 Architecture for Programmers Volume II-A: The MIPS32 Instruction Set*, MIPS MD00086 ed., 2010, https://www.mips.com/products/architectures/mips32/.

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

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

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

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

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

[17] 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 *MEMICS'10*. Brno, CZ: MU, 2010, pp. 84–91.

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

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

[20] UPX – Ultimate Packer for eXecutables, http://upx.sourceforge.net/, 2013.

[21] Z. Přikryl, "Advanced methods of microprocessor simulation," Ph.D. dissertation, Brno University of Technology, Faculty of Information Technology, 2011.