Preprocessing of Binary Executable Files Towards Retargetable Decompilation

Jakub Křoustek, Dušan Kolář
Faculty of Information Technology, IT4Innovations Centre of Excellence
Brno University of Technology
Brno, Czech Republic
{ikroustek, kolar}@fit.vutbr.cz

***Abstract*—The goal of retargetable machine-code decompilation is to analyze and reversely translate platform-dependent executable files into a high level language (HLL) representation. This process can be used for many different purposes, such as legacy code reengineering, malware analysis, etc. Retargetable decompilation is a complex task that must deal with a lot of different platform-specific features and missing information. Moreover, input files are often compressed or protected from any kind of analysis (up to 80% of malware samples). Therefore, accurate preprocessing of input files is one of the necessary prerequisites in order to achieve the best results. This paper presents a concept of a generic preprocessing system that consists of a precise signature-based compiler and packer detector, plugin-based unpacker, and converter into an internal platform-independent file format. This approach has been adopted and tested in an existing retargetable decompiler. According to our experimental results, the proposed retargetable solution is fully competitive with existing platform-dependent tools.**

Keywords*—*reverse engineering, decompilation, packer detection, unpacking, executable file, Lissom

I. INTRODUCTION

Reverse engineering is used often as an initial phase of a reengineering process. As an example we can mention reengineering of legacy software to operate on new computing platforms. One of the typical reverse-engineering tools is a machine-code decompiler, which reversely translates binary executable files back into an HLL representation, see [1], [2] for more details. This tool can be used for binary code migration, malware analysis, source code reconstruction, etc.

More attention is paid to retargetable decompilation in recent years. The goal is to create a tool capable to decompile applications independent of their origin into a uniform code representation. Therefore, it must handle different target architectures, operating systems, programming languages, and their compilers. Moreover, applications can be also packed or protected by so-called *packers* or *protectors*. This is a typical case of malware. Therefore, such input must be *unpacked* before it is further analyzed; otherwise, its decompilation will be inaccurate or impossible at all. Note: in the following text, we use the term *packing* for all the techniques of executable file creation, such as compilation, compression, protection, etc.

In order to achieve retargetable decompilation, its preprocessing phase is crucial because it eliminates most of the platform-specific differences. For example, this phase is responsible for a precise analysis of an input application (e.g., detection of a target platform). Whenever a presence of a packed code is detected, such application has to be unpacked. Furthermore, the platform-dependent object file format (OFF) is converted into an internal uniform code representation. The final task of preprocessing is an information gathering, such as detection of originally used programming language, compiler, or its version. This information is valuable during the following phases of decompilation because different languages and compilers use different features and generate unique code constructions; therefore, such knowledge implies more accurate decompilation.

In this paper, we present several platform-independent preprocessing methods, such as language and compiler detection, executable file unpacking, and conversion. These methods were successfully interconnected, implemented, and tested in a preprocessing phase of an existing retargetable decompiler developed within the Lissom project [3].

The paper is organized as follows. Section II discusses the related work of executable file preprocessing. Then, we briefly describe the retargetable decompiler developed within the Lissom project in Section III. In Section IV, we give a motivation for a compiler and packer detection within decompilation. Afterwards, our own methods used in the preprocessing phase are presented in Section V. Experimental results are given in Section VI. Section VII closes the paper by discussing future research.

II. RELATED WORK

There are several studies and tools focused on binary executable file analysis and transformation. Most of them are not focused directly on decompilation but some of these ideas can be applied in this field. Their major limitation for such usage is their bounding to one particular target platform.

In this section, we briefly mention several existing tools used for packer detection, unpacking, and OFF conversion.

A. Compiler and Packer Detection

The knowledge of the originally used tool (e.g., compiler, linker, packer) for executable creation is useful in several security-oriented areas, such as anti-virus or forensics software [4]. Overwhelming majority of existing tools are limited to the Windows Portable Executable (WinPE) format on the Intel x86 architecture and they use signature-based detection. Almost all of these tools are freeware but not open source.

Formats of signatures used by these tools for pattern matching usually contain a hexadecimal representation of the first few machine-code instructions on the application's entry point (EP). EP is an address of the first executed instruction within the application. A sequence of these first few instructions creates a so-called *start-up* or *runtime* routine, which is quite unique for each compiler or packer and it can be used as its footprint. Accuracy of detection depends on the signature format, their quality, and used scanning algorithm. Identification of sophisticated packers may need more than one signature.

Databases with signatures are either internal (i.e., precompiled in code of a detector), or stored in external files as a plain text. The second ones are more readable and users can easily add new signatures. However, detection based on external signatures is slower because they must be parsed at first. Some detection tools are distributed together with large, third-party external databases.

B. Unpacking

Binary executable file packing is done for one of these reasons—code compression, code protection, or their combination. The idea of code compression is to minimize the size of distributed files. Roughly speaking, it is done by compressing the file's content (i.e., code, data, symbol tables) and its decompression into memory or into a temporal file during execution.

Code protection can be done by a wide range of techniques (e.g., anti-debugging, anti-dumping, insertion of self-modifying code, interpretation of code in internal virtual machine). It is primarily used on MS Windows but support of other platforms is on arise in the last years (e.g., gzexe and Elfcrypt for Linux, VMProtect for Mac OS X, multi-platform UPX and HASP).

Packers are proclaimed to be used for securing commercial code from cracking; however, they are massively abused by malware authors to avoid anti-virus detection. Decompilation of compressed or protected code is practically impossible, mainly because it is "just" a static code analysis and unpacking is done during the runtime. Therefore, it is crucial to solve this issue in order to support decompilation of this kind of code.

UPX is a rare case of packers because it also supports decompression. Unpacking is a very popular discipline of reverse engineering and we can find tools for unpacking many versions of all popular packers (e.g., ASPackDie, tEunlock, UnArmadillo). We can also find unpacking scripts for popular debuggers, like OllyDbg, which do the same job.

Currently, about 80% to 90% of malware is packed [5] and about 10 to 15 new packers are created from existing ones every month [6], more and more often using polymorhic code generators [7]. In past, there were several attempts to create generic unpackers (e.g., ProcDump, GUW32), but their results were less accurate than packer-specific tools. However, creation of single-purpose unpackers from scratch is a time consuming task. Once again, these unpacking techniques are developed primarily for MS Windows and other platforms are not covered.

C. Object-File-Format Conversion

This part is responsible for converting platform-dependent file formats into an internal representation. We can find several existing projects focused on this task. They are used mostly for OFF migration between two particular platforms and they were hand-coded by their authors just for this purpose. Therefore, they cannot be used for retargetable computing.

A typical example is the MAE project [8], which supports execution of Apple Macintosh applications on UNIX. Sun Microsystems Wabi [9] allows conversion of executables from Windows 3.x to Solaris. AT&T's FreePort Express is another binary translator of SunOS executables into the Digital UNIX format. More examples can be found in [10].

III. Lissom Project's Retargetable Decompiler

The Lissom project's [3] retargetable decompiler aims to be independent on any particular target architecture, operating system, or OFF. It consists of two main parts—the preprocessing part and the decompilation core, see Figure 1. Its detailed description can be found in [11], [12].

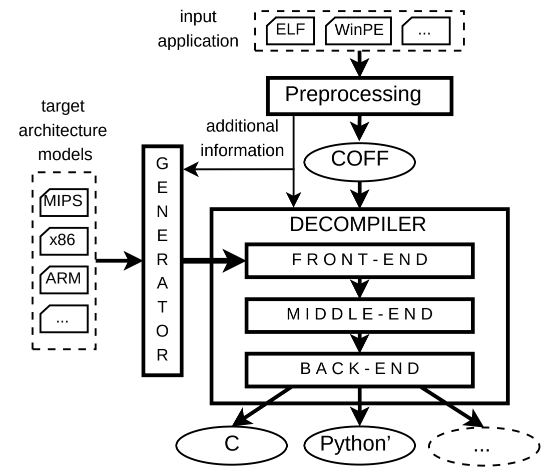

Fig. 1: The concept of the Lissom project's retargetable decompiler.

The preprocessing part is described in the following section. Basically, it unpacks and unifies examined platform-dependent applications into an internal Common-Object-File-Format (COFF)-based representation.

Afterwards, such COFF-files are processed in the decompilation core, which is partially automatically generated based on the description of target architecture. This decompilation phase is responsible for decoding of machine-code instructions, their static analysis, recovery of HLL constructions (e.g., loops, functions), and generation of the target HLL code. Currently, the C language and a Python-like language are used for this purpose and the decompiler supports decompilation of MIPS, ARM, and x86 executables.

IV. Motivation

The information about the originally used compiler is valuable during the decompilation process because each compiler generates a quite unique code in some cases; therefore, such knowledge may increase a quality of the decompilation results. One of such cases are so-called instruction idioms. Instruction idiom represents an easy-to-read statement of the HLL code that is transformed by a compiler into one or more machine-code instructions, which behavior is not obvious at the first sight. See [13] for an exhausting list of the existing idioms.

We illustrate this situation on an example depicted as a C language code in Figure 2. This program uses an arithmetical expression "`-(a >= 0)`", which is evaluated as `0` whenever the variable `a` is smaller than zero; otherwise, the result is evaluated as `-1`. Note: the following examples are independent on the used optimization level within the presented compilers. All compilers generate 32-bit Linux ELF executable files for Intel x86 architecture [14] and the assembly code listings were retrieved via `objdump` utility.

```
#include <stdio.h>

int main(int argc, char **argv)
{
   int a;

   scanf("%d", &a);
   // Prints - "0" if the input is smaller than 0
   //        - "-1" otherwise
   printf("%d\n", -(a >= 0));

   return 0;
}
```

Fig. 2: Source code in C.

Several compilers substitute code described in Figure 2 by instruction idioms. Moreover, different compilers generate different idioms. Therefore, it is necessary to distinguish between them. For example, code generated by the GNU compiler GCC version 4.0.4 [15] is depicted in Figure 3. As we can see, the used idiom is non-trivial and its readability is far from the original expression.

```
Address   Hex dump    Intel x86 instruction
--------------------------------------------
; scanf
; Variable 'a' is stored in %eax
80483e2:  f7 d0       not  %eax
80483e4:  c1 e8 1f    shr  $31,%eax
80483e7:  f7 d8       neg  %eax
; Print result stored in %eax
; printf
```

Fig. 3: Assembly code generated by gcc 4.0.4.

The Clang compiler is developed within the LLVM project [16], [17]. Output of this compiler is illustrated in Figure 4. As we can see, Clang uses idiom, which is twice as long as the previous one and it is assembled by the different set of instructions. Therefore, it is not possible to implement one generic decompilation analysis. Such solution will be inaccurate and slow (i.e., detection of all existing idioms no matter on the originally used compiler).

Decompilation of instruction idioms (or other similar constructions) produces a correct code; however, without any compiler-specific analysis, this code is hard to read by a human because it is more similar to a machine-code representation than to the original HLL code. Compiler-specific analyses are focused on these issues (e.g., they detect and transform idioms back to a well-readable representation), but the knowledge of the originally used compiler and its version is mandatory.

Figure 5 depicts decompilation results for the gcc compiled code listed in Figure 3 (i.e., code generated by gcc 4.0.4). The Lissom retargetable decompiler was used for this task. As we

```
Address   Hex dump        Intel x86 instruction
------------------------------------------------
; scanf
; Variable 'a' is stored on stack at -16(%ebp)
8013bf:   83 7d f0 00     cmpl   $0,-16(%ebp)
8013c3:   0f 9d c2        setge  %dl
8013c6:   80 e2 01        and    $1,%dl
8013c9:   0f b6 f2        movzbl %dl,%esi
8013cc:   bf 00 00 00 00  mov    $0,%edi
8013d1:   29 f7           sub    %esi,%edi
8013d3:   ;...
8013d6:   89 7c 24 04     mov    %edi,4(%esp)
; Print result stored on stack at 4(%esp)
; printf
```

Fig. 4: Assembly code generated by clang 3.1.

can see, the expression contains bitwise shift and xor operators instead of the originally used comparison operator. This makes the decompiled code hard to read.

```
#include <stdint.h>
#include <stdio.h>

int main(int argc, char **argv)
{
   int apple;
   apple = 0;
   scanf("%d", &apple);
   printf("%d\n", -(apple >> 31 ^ 1));
   return 0;
}
```

Fig. 5: Decompiled source code (without compiler-specific analyses).

Furthermore, it is important to detect compiler version too. In Figure 6, we illustrate that the different versions of the same compiler generate different code for the same expression. We use gcc version 3.4.6 and the C code from the Figure 2.

```
Address  Hex dump              Intel x86 instruction
----------------------------------------------------
; scanf
; Variable 'a' is stored on stack at -4(%ebp)
80483f3: 83 7d fc 00           cmpl $0,-4(%ebp)
80483f7: 78 09                 js   8048402
80483f9: c7 45 f8 ff ff ff ff  movl $-1,-8(%ebp)
8048400: eb 07                 jmp  8048409
8048402: c7 45 f8 00 00 00 00  movl $0,-8(%ebp)
8048409:
; Print result stored on stack at -8(%ebp)
; printf
```

Fig. 6: Assembly code generated by gcc 3.4.6.

In this assembly code snippet, we can see that no instruction idiom was used. The code simply compares the value of a variable with zero and sets the result in a human-readable form. It is clear that the difference between the code generated by the older (Figure 6) and the newer version (Figure 3) of this compiler is significant. Therefore, we can close this section stating that information about the used compiler and its version is important for decompilation.

V. Preprocessing Phase of the Retargetable Decompiler

In this section, we present a design of the preprocessing phase within the Lissom project retargetable decompiler. The complete overview is depicted in Figure 7. The concept consists of the following parts.

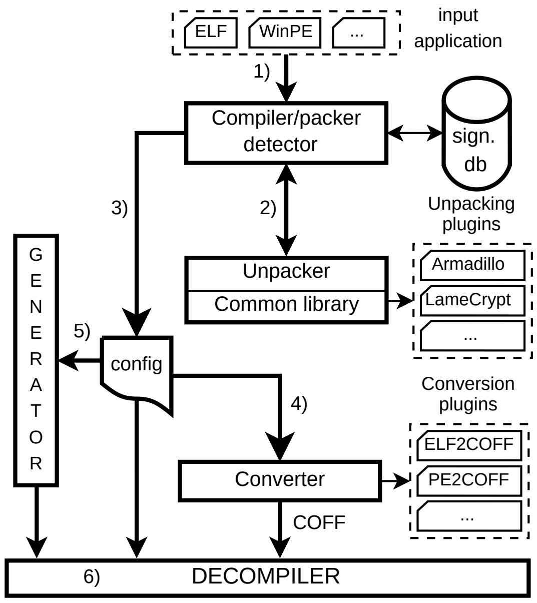

Fig. 7: The concept of the preprocessing phase.

At first, the input executable file is analyzed and the used OFF is detected. All common formats are supported (e.g., WinPE, UNIX ELF, Mach-O). Information about the target processor architecture is extracted from the OFF header (e.g., `e_machine` entry in ELF OFF) and it is used together with other essential information in further steps.

The next part of this step is a detection of a tool used for executable creation. This is done using a signature-based detection of start-up code as described in Section II. Example of such a start-up code can be seen in Figure 8. Signature for this code snippet is "`5589E583EC18C 7042401000000FF15--------E8`", where each character represents a nibble of instruction's encoding. All variable parts must be skipped during matching by a wild-card character "`-`", e.g., a target address in the `call` instruction. This signature format is quite similar to formats used by other detectors listed in Section II.

```
Address    Hex dump          Intel x86 instruction
--------------------------------------------------
0040126c:  55                push %ebp
0040126d:  89e5              mov  %esp, %ebp
0040126f:  83ec18            sub  $0x18, %esp
00401272:  c7042401000000    movl $0x1, (%esp)
00401279:  ff1500000000      call *0x0
0040127f:  e8                ...
```

Fig. 8: Start-up code for MinGW gcc v4.6 on x86 (`crt2.o`) generated by `objdump -d`.

Our signature format also supports two new features—description of nibble sequences with zero or more occurrences and description of unconditional short jumps. Example of the former one is "`(90)`", denoting an optional sequence of `nop` instructions for x86 architecture. Example for the second one is "`#EB`", denoting an unconditional short jump for the same architecture, which size is specified in the next byte; everything between the jump and its destination is skipped. In Figure 9, we can find a code snippet covered by signature "`#EB(90)40`".

```
Address     Hex dump          Intel x86 instruction
---------------------------------------------------
00401000:   eb 02             jmp short <00401004>
00401002:   xx xx             ; don't care
00401004:   90                nop
00401005:   90                nop
00401006:   40                inc %eax
```

Fig. 9: Example of advanced signature format.

These features come handy especially for polymorphic packers [7] producing a large number of different start-up codes (e.g., Obsidium packer). Describing one version of such packer usually needs dozens of classical signatures. However, this number can be significantly reduced using the above-mentioned features.

Signatures within our internal database were created with focus on the detection of the packer's version. This information is valuable for decompilation because two different versions of the same packer may produce diverse code constructions. The database also contains signatures for non-WinPE platforms; therefore, it is not limited like other tools. Finally, new signatures can be automatically created whenever the user can provide at least two files generated by the same version of packer. Presence of multiple files is mandatory in order to find all variable nibbles in the start-up code.

Whenever a usage of packer was detected in the first phase, the unpacking part is invoked. Unpacking is done by our own generic unpacker, which consists of a common unpacking library and several plugins implementing unpacking of particular packers. The common library contains the necessary functions for rapid unpacker creation, such as detection of the original entry point (OEP), dump of memory, fixing import tables, etc. Therefore, a plugin itself is very tiny and contains only code specific to a particular packer.

A plugin can be created in two different ways: either it can reverse all the techniques used by the packer and produce the original file, or the plugin can execute the packed file, wait for its decompression, and dump its unprotected version from memory to file. The first one is hard to create because it takes a lot of time to analyze all the used protection techniques. Its advantage is that unpacking can be done on any platform because the file is not being executed. That is the main disadvantage of the second approach. Such a plugin can be created quickly; however, it must be executed on the same target platform. In present, we support unpacking of several popular packers like Armadillo, UPX (Linux and Windows), NoodleCrypt and others in the second way. See Section VII for its future research.

After unpacking, the re-generated executable file is once more analyzed. In rare cases, second packer was used and we need to unpack this file once more. Otherwise, the analysis will try to detect the used compiler and its version, and generate a configuration file, which is used by other decompilation tools. This configuration file also contains information about the target architecture, endianness, bitwidth, address of OEP, etc.

Afterwards, the platform-specific unpacked executable file is converted into an internal COFF-based representation. The converter is also realized in a plugin-based way and each plugin converts one particular OFF. Currently, we support ELF, WinPE, Mach-O, and several others OFF. See [10] for more details about this tool.

Using the information about the target architecture in the configuration file, the instruction decoder is automatically created by the generator tool [12]. Instruction decoder is the first part of the decompiler's front-end, which translates machine code instructions into a semantics description of their behavior.

Finally, the COFF executable file is processed in the generated decompiler according to the configuration file. Using the provided information about used compiler, it can selectively enable compiler-specific analyses (e.g., detection of instruction idioms, recovery of functions).

VI. Experimental Results

This section contains an evaluation of the previously described method of packer detection. The accuracy of our tool (labeled as "Lissom") is compared with the latest versions of existing detectors. Their short overview is depicted in Table I.

TABLE I: Overview of existing compiler/packer detection tools.

tool		signatures		
name	version	internal	external	total
Lissom	1.00	2181	0	2181
RDG Packer Detector [18]	0.6.9	?	10	?
ProtectionID (PID) [19]	0.6.4.0	499	0	499
Exeinfo PE [20]	0.0.3.2	667	7075	7742
Detection is Easy (DiE) [21]	0.6.4	?	1870	?
NtCore PE Detective [22]	1.2.1.1	0	2806	2806
FastScanner [23]	3.0	1605	1832	3437
PEiD [24]	0.95	672	1774	2446

All of these detection tools use the same approach as our solution—detection using signature matching. As we can see in Table I, most of them use a combination of pre-compiled internal signatures and a large external database created by the user community. The competitive solutions are limited to WinPE OFF and a number of their signatures varies between hundreds and thousands. The number of internal signatures is not always absolutely precise because some authors do not specify this number, like RDG or FastScanner. Therefore, we had to analyze such applications and try to find their databases manually (e.g., using reverse engineering). We were unable to find it in the RDG and DiE detectors. Our solution consists of 2181 internal signatures for all supported OFFs and we also support the concept of external signatures.

Using reverse engineering, we also figured out that several tools (e.g., PEiD) use additional heuristic technique for packer detection. These techniques are not focused on the start-up code or machine code at all. They are analysing several properties of the executable file (e.g., attributes of sections, information stored within file header) and they perform a detection of packer-specific behavior. Using this heuristic analysis, it is possible to detect even the polymorphic packers like Morphine encryptor.

Twenty WinPE packers (e.g., ASPack, FSG, Obsidium, UPX) and several their versions (105 different tools in total) were used for comparison of previously mentioned detectors. We used these packers for packing several compiler-generated executables—with different size (50kB to 5MB), used compiler, compilation options, and packer options. The purpose is that some packers create different start-up code based on the file size and characteristics (data-section size, PE Header flags, etc.). The test set consists of 5267 executable files in total. We prepared three test cases for the evaluation of the proposed solution.

At first, we evaluated the detection of packer's name. This type of detection is the most common and also the easiest to implement because generic signatures can be applied (i.e., signatures with only few fixed nibbles describing complete packer family). On the other hand, this information is critical for the complete decompilation process because if we are unable to detect usage of executable-file protector, the decompilation results will be highly inaccurate. The results of detection are compared in Figure 10.

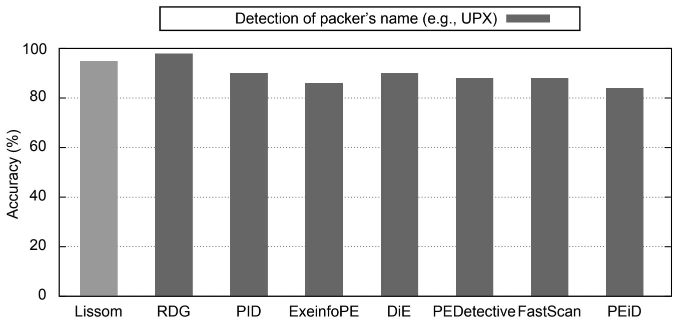

Fig. 10: Summary of packer detection (packer names).

According to the results, the RDG [18] detector has the best ratio of packer's name detection (98%), while our solution was second with ratio over 95%. All other solutions achieved comparable results—between 80% and 90%. We can also notice that larger signature databases do not imply better results in this cathegory (e.g., Exeinfo PE). Such large databases are hard to maintain and they can produce several false-positive results because of too much generic signatures.

Afterwards, we tested the accuracy of tool's major version detection. In other words, this test case was focused on tool's ability to distinguish between two generations of the same tool (e.g., UPX v2 and UPX v3). This feature comes handy in the front-end phase during compiler specific analyses. For example, the compiler may use in its newer versions more aggressive optimizations that have a very specific meaning and they need a special attention by the decompiler (e.g., instruction idioms, loops transformation, jump tables), see Section IV for details. The results are depicted in Figure 11.

Within this test case, RDG and our solution once again achieved the best results (both scored 93%). Only ExeinfoPE and ProtectionID exceeded 80% success ration from the others.

Finally, we tested the ratio of precise packer's version detection. This task is the most challenging because it is necessary to create one signature for each particular version of each particular packer. This information is crucial for the unpacker because the unpacking algorithms are usually created

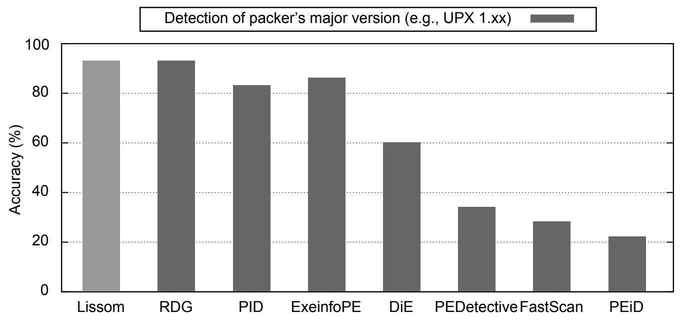

Fig. 11: Summary of packer detection (packer versions).

for one particular packer version and their incorrect usage may lead to a decompilation failure.

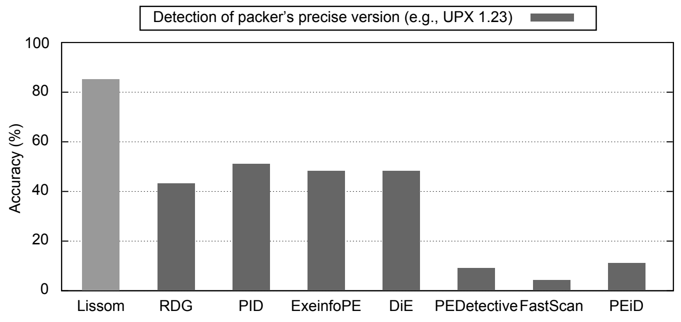

Fig. 12: Summary of packer detection (detailed detection).

Based on the results depicted in Figure 12, our detector achieved the best results in this category with 86% accuracy. The results of other solutions were much lower (51% at most). This is mainly because we focus primarily on detecting the precise version and we also support search in the entire PE file and its overlay and not just on its entry point.

VII. CONCLUSION

This paper was aimed on architecture-independent preprocessing methods used within the existing retargetable decompiler. We introduced methods of packer detection, unpacking, and OFF conversion. Up to now, this concept has been successfully tested on the MIPS, ARM, and x86 architectures within the Lissom project's [3] retargetable decompiler.

We made several tests focused on accuracy of our solution and according to the experimental results, it can be seen that our concept is fully competitive with other existing tools.

We close the paper by proposing two areas for future research. (1) The unpacking phase can be enhanced by using retargetable simulators [25]. Such tools can emulate the target host system and, therefore, it will not be necessary to unpack executables on the same system as its origin. (2) We can further increase decompilation results by creation of new signatures and compiler-specific analyses (e.g., better loop statement recovery, detecting different types of function calls).

ACKNOWLEDGMENTS

This work was supported by the project TA ČR TA01010667 System for Support of Platform Independent Malware Analysis in Executable Files, BUT FIT grant FIT-S-11-2 and FEKT-FIT-J-13-2000 Validation of Executable Code for Industrial Automation Devices using Decompilation, by the project CEZ MSM0021630528 Security-Oriented Research in Information Technology, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

REFERENCES

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

[2] M. J. V. Emmerik, "Static single assignment for decompilation," Ph.D. dissertation, University of Queensland, Brisbane, AU-QLD, 2007.

[3] Lissom, 2013, available on URL: http://www.fit.vutbr.cz/research/groups/lissom/ [retrieved: 2013-04-22].

[4] G. Taha, "Counterattacking the packers," in AVAR, 2007.

[5] M. M. T. Brosch, "Runtime packers: The hidden problem?" in Black Hat, 2006.

[6] K. Babar and F. Khalid, "Generic unpacking techniques," in 2nd International Conference on Computer, Control and Communication, 2009, pp. 1–6.

[7] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," in 14th ACM Conference on Computer and Communications Security (CCS'07). ACM, 2007, pp. 541–551.

[8] Apple Inc., "Macintosh application environment," 1994, available on URL: http://www.mae.apple.com/ [retrieved: 2013-04-22].

[9] D. R. P. Hohensee, M. Myszewski, "Wabi cpu emulation," Hot Chips VIII, 1996.

[10] 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'2011). Lviv Polytechnic National University, 2011, pp. 127–130.

[11] L. Ďurfina et al., "Design of a retargetable decompiler for a static platform-independent malware analysis," International Journal of Security and Its Applications, vol. 5, no. 4, 2011, pp. 91–106.

[12] 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, Ontario, CA: IEEE Computer Society, 2012, pp. 51–60.

[13] H. Warren, Hacker's Delight. Boston: Addison-Wesley, 2003.

[14] Intel Corporation, "Intel 64 and ia-32 architectures software developer's manual volume 1: Basic architecture," 2011.

[15] GNU Compiler Collection, 2013, available on URL: http://gcc.gnu.org/ [retrieved: 2013-04-22].

[16] Clang, 2013, available on URL: http://clang.llvm.org/ [retrieved: 2013-04-22].

[17] The LLVM Compiler System, 2013, available on URL: http://llvm.org/ [retrieved: 2013-04-22].

[18] RDG Packer Detector, 2013, available on URL: http://rdgsoft.8k.com/ [retrieved: 2013-04-22].

[19] ProtectionID, http://pid.gamecopyworld.com/, 2013, available on URL: [retrieved: 2013-04-22].

[20] ExeinfoPE, 2013, available on URL: http://www.exeinfo.xwp.pl/ [retrieved: 2013-04-22].

[21] Die, 2013, available on URL: http://hellspawn.nm.ru/ [retrieved: 2013-04-22].

[22] NtCore PE Detective, 2013, available on URL: http://www.ntcore.com/ [retrieved: 2013-04-22].

[23] FastScanner, 2013, available on URL: http://www.at4re.com/ [retrieved: 2013-04-22].

[24] PEiD, 2013, available on URL: http://www.peid.info/ [retrieved: 2013-04-22].

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