Advanced Preprocessing of Binary Executable Files and its Usage in Retargetable Decompilation

Jakub Křoustek, Peter Matula, Dušan Kolář, and Milan Zavoral
Faculty of Information Technology, IT4Innovations Centre of Excellence
Brno University of Technology
Brno, Czech Republic
{ikroustek, imatula, kolar}@fit.vutbr.cz, xzavor02@stud.fit.vutbr.cz

Abstract—**Retargetable machine-code decompilation is used for a platform-independent transformation of executable files into a high level language (HLL) representation (e.g., C language). It is a complex task that must deal with a lot of different platform-specific features and missing information. Accurate preprocessing of input executable files is one of the necessary prerequisites in order to achieve the best results. Furthermore, we can use gathered information to achieve higher quality of decompilation. This paper presents an extended version of our previous system for an accurate code preprocessing. It is implemented as a generic preprocessing system that consists of a precise compiler and packer detector, plugin-based unpacker, converter into an internal platform-independent file format, and debugging information gathering library. We also describe an utilization of the collected information in a problem of automatic data-type reconstruction. This system 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

This article is closely related to the paper [1]. We extend this previous paper by presenting several new methods of packer and compiler detection (e.g., heuristics-based detection, signatures for ELF file format) and by describing our novel approach of preprocessing in type-recovery phase of decompilation (e.g. exploitation of debugging information, known library function calls). We also present re-evaluation of all experimental tests.

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 [2], [3] 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, its version, or detection and processing of debugging information. 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, conversion, and format-independent debugging information processing. We also demonstrate a utilization of this information on example of the data-type recovery analysis. These methods were successfully interconnected, implemented, and tested in a preprocessing phase of an existing retargetable decompiler developed within the Lissom project [4].

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. Section VI shows how the data-type recovery analysis uses previously gathered information. Experimental results are given in Section VII. Section VIII 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, OFF conversion and debugging information gathering.

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 [5]. 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 unpacking itself. 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 [6] and about 10 to 15 new packers are created from existing ones every month [7], more and more often by using polymorhic code generators [8]. 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 [9], which supports execution of Apple Macintosh applications on UNIX. Sun Microsystems Wabi [10] 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 [11].

D. Debugging Information Gathering

Debugging information is generated by compilers and traditionally used by debuggers to find and fix software bugs [12]. Since it represents relationship between the machine code and the original source code, it may as well be exploited in decompilation, or any other executable file analysis. The typical use in reverse engineering is to evaluate accuracy of the analysis by comparing inferred results with those from the debugging information.

This approach was used in [13] where the `readelf` utility was used to extract the debugging data, or in [14] and [15] by using the `libdwarf` library. It is however not clear whether any of these tools is capable to incorporate such information to its algorithm and produce more accurate output because of it. Since most of the executable-analysis applications are linked to a particular architecture and platform it is also unlikely that they are able to process different debugging formats.

Despite the fact malware rarely contains such additional data, it would be foolish not to capitalize on them if they are actually present. This also opens new areas of decompiler applications. For example in binary verification of critical programs, which may be intentionally compiled with the debugging information to make analysis easier.

For this reasons, we have already created the debugging information preprocessing libraries for the two most widely used formats (DWARF, Microsoft PDB), and used them for recovery of variables, functions, and arguments [16]. In the original paper [1], we used an untyped Python-like language and we did not exploit the full potential. In this paper we show how to incorporate precise data types obtained from the debugging information to our type recovery algorithm and propagate them throughout the whole program.

III. Lissom Project's Retargetable Decompiler

The Lissom project's [4] 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 [17], [18].

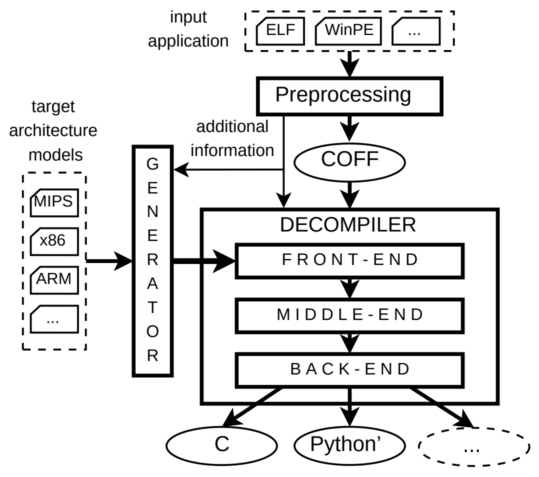

Figure 1. The concept of the Lissom project's retargetable decompiler.

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 different code in some cases; therefore, such knowledge may increase a quality of the decompilation results. One of such cases is a usage of 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 [19] 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 [20] 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;
}
```

Figure 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 [21] 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
```

Figure 3. Assembly code generated by gcc 4.0.4.

The Clang compiler is developed within the LLVM project [22], [23]. 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).

```
; 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
```

Figure 4. Assembly code generated by clang 3.1.

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 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;
}
```

Figure 5. Decompiled source code from program listed in Figure 3. In this case, the decompiler lacks any compiler-specific analysis and the result is hard to read.

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...    movl $-1,-8(%ebp)
  8048400: eb 07                jmp  8048409
  8048402: c7 45 f8 00 00...    movl $0,-8(%ebp)
  8048409:
; Print result stored on stack at -8(%ebp)
; printf
```

Figure 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.

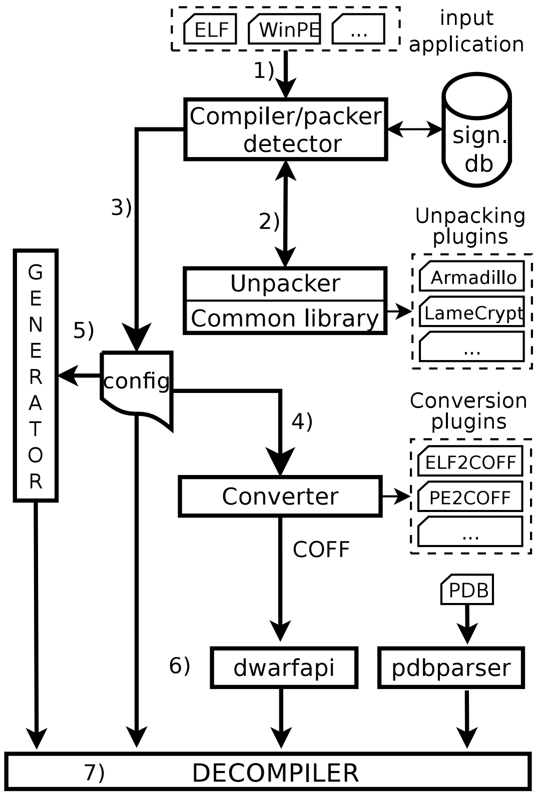

Figure 7. The concept of the preprocessing phase.

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 by 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 VII.

```
; 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               ...
```

Figure 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
```

Figure 9. Example of advanced signature format.

These features come handy especially for polymorphic packers [8] 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 by 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 most of 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.

Unfortunately, some polymorphic packers (e.g., Morphine encryptor) cannot be described via extended format of signatures. These packers generate entirely different start-up routine for each packed file. For instance, there is an example of three different start-up routines generated by the Morphine encryptor:

```
1C1C26083EB00F6DFF6DF535BFC7C03C1EC005
7408525566C1C4105D5A51510AC95959F9FC60
510FB6C9770525FFFFFFFF8E2F35983FA2D8B
```

If we use these samples to create signature, we get the following result:

```
--------------------------------------
```

As we can see, resulting signature contains only wild-card characters, which is useless for detection. Therefore, in order to support a precise detection, we support concept of additional heuristics that are focused on polymorphic packers. These heuristics are analysing several properties of the executable file (e.g., attributes of sections, information stored within file header, offset of EP in executable file) and they perform the detection based on a packer-specific behavior. Example of such heuristic is illustrated in Figure 10—it takes into account file format, target architecture, offset of EP in file, and information about file sections. Heuristic is written in C++-like pseudocode.

```
if(file_format == WIN_PE &&
   target_architecture == INTEL_X86 &&
   EP_file_offset >= 0x400 &&
   EP_file_offset <= 0x1400 &&
   sections[0].name == ".text" &&
   sections[1].name == ".data" &&
   sections[2].name == ".idata" &&
   sections[2].size == 0x200)
{
   return "Morphine 1.2";
}
```

Figure 10. Heuristic for Morphine encryptor v1.2.

Except of heuristics for precise detection of polymorphic packers, we also support simpler heuristics, which are focused only on a name, number, and order of sections. Such heuristics cannot detect exact version of used packer, but they are useful if signature database does not contain entry for related tool. Their overview is depicted in Table I.

Whenever a usage of packer is 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 VIII 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 implemented in a plugin-based way and each plugin converts one particular OFF. Currently, we support ELF, WinPE, Mach-O, and several others OFF. See [11] for more details about this tool.

TABLE I. OVERVIEW OF HEURISTICS FOCUSED ON THE NAME AND ORDER OF SECTIONS.

packer	heuristic
Upack	`sections[0].name == ".Upack"`
PE-PACK	`sections[last].name == ".PEPACK!!"`
WWPack32	`sections[last].name == ".WWP32"`
yoda's Crypter	`sections[last].name == "yC"`
LameCrypt	`sections[last].name == "lamecryp"`
ASPack	`sections[last - 1].name == ".aspack" && sections[last].name == ".adata"`
PEBundle	`sections[last - 1].name == "pebundle" && sections[last].name == "pebundle"`
UPX	`numberOfSections == 3 && sections[0].name == "UPX0" && sections[1].name == "UPX1" && (sections[2].name == "UPX2" || sections[2].name == ".rsrc")`
Petite	`numberOfSectionsWithName(".petite") == 1`
PKLite	`numberOfSectionsWithName(".pklstb") == 1`
Krypton	`numberOfSectionsWithName(".krypton") == 1 && numberOfSectionsWithName("YADO") >= 1`
NFO	`numberOfSectionsWithName("NFO") == numberOfSections`
PELock NT	`numberOfSectionsWithName("PELOCKnt") > 0 && (numberOfSectionsWithName("PELOCKnt") >= numberOfSections - 2 || numberOfSections == 1)`
PELock v1.x	`numberOfSectionsWithName(".pelock") > 0 && numberOfSectionsWithName(".pelock") >= numberOfSections - 1`
MEW v10	`numberOfSections == 2 && section[0].name == ".data" && section[1].name == ".decode"`
MEW v11 SE 1.x	`numberOfSections == 2 && section[0].name.containString("MEW")`
NsPack v2.x	`for(i = 0; i < numberOfSections; ++i) sections[i].name == string("nsp" + numToStr(i))`
NsPack v3.x	`for(i = 0; i < numberOfSections; ++i) sections[i].name == string(".nsp" + numToStr(i))`

Using the information about the target architecture in the configuration file, the instruction decoder is automatically created by the generator tool [18]. Instruction decoder is the first part of the decompiler's front-end, which translates machine code instructions into a semantics description of their behavior.

The last step before the actual decompilation is processing of debugging information (if present). Currently we support two major debugging formats, architecture independent DWARF and Microsoft PDB. Each of them is handled by a separate specialized library that loads their contents to the internal representation and provides convenient access methods. Thanks to the previous steps, it is possible to determine input's platform and check for the presence of the DWARF sections in the COFF file.

Since the PDB debugging information is distributed in a separate file, checking object file would be pointless and it is necessary to provide such file whenever an additional information has to be used. DWARF format is preprocessed by the mid-layer library called `dwarfapi`. It uses another library named `libdwarf` to parse low-level debugging information, upon which it builds high-level, object-oriented data structures.

Because there is no available PDB toolkit that would suit our needs, we created our own parser library called `pdbparser`. Basic principles behind both of these tools were described in [16]. Since then, we further extended them to support all data types present in the C programming language. Adding object-oriented features used in the C++ and other similar languages is planned in the near future.

Finally, the COFF executable file is processed in the generated decompiler according to the configuration file. Using the provided information about used compiler, the decompiler can selectively enable compiler-specific analyses (e.g., detection of instruction idioms, recovery of functions). One of the first steps is to check for the debugging information presence and load it to the internal canonical representation using already described libraries. This way, any further analysis can access this information in an unified manner no matter the format of an underlying source.

VI. Preprocessing in the Type Recovery Analysis

The goal of a type recovery analysis is to associate each piece of data with a high-level data type as close to the original source code type as possible. We presented the design of a data-flow based type recovery algorithm used by our retargetable decompiler in [24]. Simplified overview is depicted in Figure 11.

Reconstruction can be divided into the three main phases: (1) Object (registers, global/local variables, function arguments, etc.) initialization, where each occurrence gets an initial type, and these types are interconnected by the propagation equations. (2) Simple and composite analysis run over the set of objects and equations. Objects' types are inferred based on their initial types and the semantics of the operations they occurred in. Analysis ends once the system's fixpoint is reached. (3) Reconstructed types are used in the output intermediate representation (IR). The original paper focused solely on the core of the analysis – simple data-type inference based on the semantics of the individual instructions.

This approach can be applied to any input without additional conditions. The disadvantage is its lower accuracy compared to the other potential type information sources. This section presents utilization of two highly accurate data-type sources use of which is enabled by the extensive preprocessing.

A. Data-Type Debugging Information

Debugging information contains exact types of all objects existing in the original source code. By the time of the type recovery analysis run, they have already been preprocessed and are easily accessible. All that needs to be done is to apply

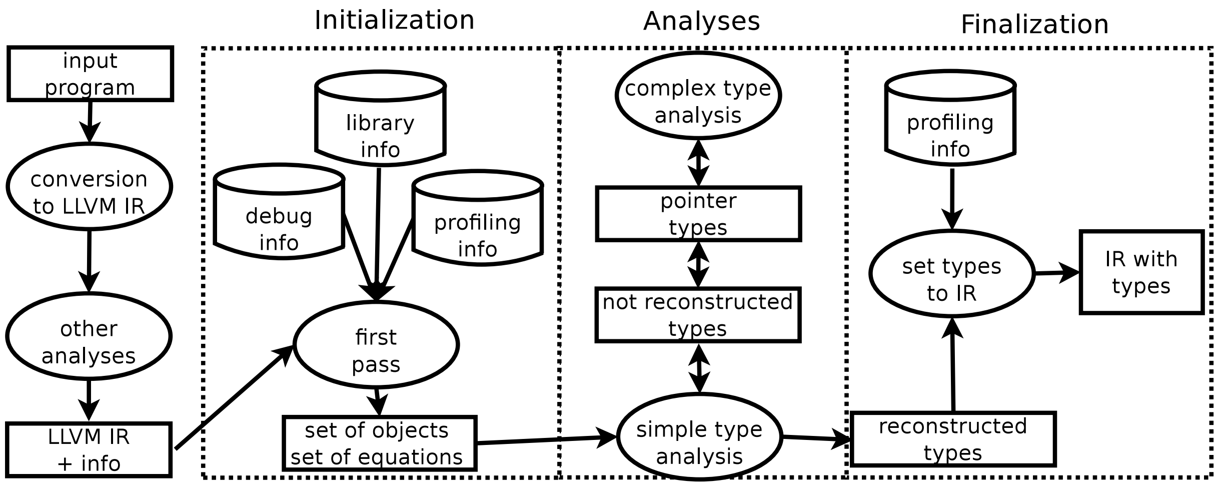

Figure 11. Data-type recovery analysis scheme. Taken from [24].

them in an initialization phase instead of inferring initial types from the nature of object's occurrence. This may look simple enough, but we have to make sure that the type will be assigned to the correct object. If the debugging information is present, functions and their arguments are reconstructed precisely and it is trivial to link them with their true types.

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

struct s { int i; char c; float f; };
struct s global;

int main()
{
   FILE *pFile;
   pFile = fopen("some.file", "w+");

   float local = rand();
   fprintf(pFile, "%f", local);

   fscanf(pFile, "%d %c %f",
          global.i, global.c, global.f);
   printf("%d %c %f",
          global.i, global.c, global.f);

   return 0;
}
```

Figure 12. Example of the original source code used for demonstration of the type recovery.

For global variables, things gets slightly complicated. Variable analysis detected all the global memory accesses, whose addresses can be statically determined. Then it created simple global variables for these addresses and used their names in the access instructions. For example, variable `global` in Figure 12 was recognized from three different accesses to its elements as three separate variables. The first one at address X, the second at $X + 4$ and the third at $X + 8$.

As is displayed in Figure 13 (showing relevant fragment of the DWARF data), debugging information contains entry only for the entire composite object. Fortunately, it is not complicated to link the computed address X with the address from the debug data (`0x0891daa4` in this case) and assign the true type to the first simple variable.

```
DW_TAG_subprogram
  DW_AT_name              "main"
  DW_AT_frame_base <loclist with 3 entries follows>
    [0]<lowpc=0x0000><highpc=0x0004> DW_OP_reg29
    [1]<lowpc=0x0004><highpc=0x0010> DW_OP_breg29+24
    [2]<lowpc=0x0010><highpc=0x0104> DW_OP_breg30+24

DW_TAG_variable
  DW_AT_name              "local"
  DW_AT_location          DW_OP_fbreg -24

DW_TAG_variable
  DW_AT_name              "global"
  DW_AT_location          DW_OP_addr 0x0891daa4
```

Figure 13. Relevant fragment of the DWARF debugging information generated for the code in Figure 12.

Based on the known type size, the algorithm can merge all subsequent simple variables into one composite object. To achieve the code quality similar to the original source, all instructions accessing simple globals must be changed to operations reading or writing the corresponding composite elements. Using this method, we are able to assign the true types even to the objects accessed by addresses, whose values cannot be statically computed (e.g., accessing global array in a cycle using an iterator). However, creating correct access instructions with iterators demands usage of the composite type recovery analysis, which is beyond the scope of this article.

The same principles used for the globals can be applied to the stack (local) objects as well. However, linking types to the related objects gets much more complicated in this case.

Looking at the same examples as before, we can see that stack object `local` is located at `DW_OP_fbreg -24`. This means that it is at offset of `-24` from the current frame base. The frame base is determined by the actual program counter and the corresponding expression in the function's location list. For example, if program counter is between `0x0010` and `0x0104`, then frame base is equal to `DW_OP_breg30 + 24`, where `DW_OP_breg30` represents current value of the register labeled by DWARF with the number `30`.

To successfully use this information, we need to make sure our stack analysis computes the same stack variable offsets as would be calculated by the debugger using the debugging data. Furthermore, we need to be able to repeat computation of the DW_AT_location. To do so, mapping between the DWARF register numbers and real architecture registers must be known. If stack offsets were computed correctly, it is possible to link them with the DW_AT_location results and find corresponding DWARF entries for the detected stack variables. Finally, it is trivial to apply true types and perform the same kind of aggregation and instruction replacement as for the global variables.

Note: Exploitation of the PDB debugging information is similar or sometimes even easier to the presented DWARF usage.

B. Known Library Function Calls

Calls of the known library functions are another highly accurate data-type source whose optimal usage is enabled by the extensive input preprocessing. It is providing types of the same quality as the debugging data without the need of any additional information in the executable. However, decompiler must be able to detect linked function calls and have the database of functions' prototypes containing information about the types they use. Generic signature based function code detection and library type information (LTI) file creation is described in [25]. In this paper, we deal with its application in the type recovery algorithm and the advantages gained from the preprocessing.

```
%struct.struct_drand48_data =
  type { [3 x i16], [3 x i16], i16, i16, i64 }

# FILE * fopen (const char *name, const char *mode)
  fopen %struct.struct__IO_FILE* 2 i8*, i8*

# int fscanf (FILE *stream, const char *form, ...)
  fscanf i32 3 %struct.struct__IO_FILE*, i8*, ...

# void * malloc (size_t size)
  malloc void* 1 i32

# double strtod (const char *str, char **ptr)
  strtod double 2 i8*, OUT REF i8**
```

Figure 14. Examples of the library type information entries from *stdlib.lti* and *stdio.lti* files.

Separate LTI file containing the function prototypes and the definitions of used composite types is generated for each known standard library. Figure 14 depicts few real examples from the *stdio.h* and *stdlib.h*. Lines starting with the symbol # are comments, other lines are actual entries written in the LLVM IR syntax. The first record is an example of a structure definition containing two arrays and three simple members. Other lines show prototypes of some well-known functions.

Function name is always the first, followed by the function's return type, number of arguments and finally arguments' types. If the function is variadic, its parameter list ends with the ... token. Arguments may be also flagged to express some additional information about their typical use. For example, OUT signals that something is returned through the parameter, REF means that argument is typically passed by the reference. This makes it possible to decide, which of the different call variants of strtod() depicted in Figure 15 is more likely to be used.

Thanks to the input preprocessing, it is possible to pick the optimal set of LTI files matching program's architecture, operating system, and compiler. These LTI files are used to assign the true data-types to argument and return objects each time known function call is detected. Subsequent type propagation will spread the information between other object occurrences and to all objects in its equivalence class (defined by the relation: *to have the same data type*).

```
char in[] = "365.24 29.53";

// Variant #1:
  char* pEnd;
  double d = strtod (in, &pEnd);

// Variant #2:
  char** pEnd;
  double d = strtod (in, pEnd);
```

Figure 15. Several possible variants how to call strtod().

C. Analysis Modification

Beside the original source code objects, decompiler's output usually contains other variables arising from the use of registers or auxiliary local/global variables. Types of such objects are not present in a potential debugging information; therefore, they cannot be set directly. Data types recovered from the function calls are initially set only to the immediate objects used by the call. For this reasons, no matter the type sources quality, decompiler always performs full data-flow type propagation. The only difference is, that some initial object types set in the analysis initialization are more precise than others.

The analysis core presented in [24] infers types by repeated application of the propagation rules and the join function. For each object, the greatest lower bound of all its occurrences is found. Algorithm described in the article takes all initial type estimates as equal and may refine them in order to unify all object's occurrences. Since types obtained from the sources shown in this paper are already precise, this behavior is undesired for their propagation.

The solution is to tag each type with the identification of its origin. More precise the origin, greater the priority during propagation between object's occurrences and other objects in the equivalence class. Types with high enough tags are also saved from any modifications, so that already correct types are not broken in the process.

Origin tags in an ascending order of precision are: (1) Default type, 32-bit signed integer. (2) Type inferred from the instruction semantics. (3) Non-default type that was set by some previous analysis. (4) Type from the dynamic analysis. (5) Type from the known function call. (6) Type from the debugging information. Only the last two are saved from any modifications. It is however possible, in some special cases, to further enhance the final outcome. Allocation related functions are the typical example. As we can see in Figure 14, return type of the `malloc()` function is pointer to `void`. However, this cannot be the type of the variable where the result is stored. In this case, analysis puts together types from two sources of different precision to get one final data type.

VII. Experimental Results

This section contains an evaluation of the previously described methods 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 II. Detectors are compared in three test sets, see below: (A) WinPE packers, (B) WinPE polymorphic packers, (C) ELF packers and compilers. This section also contains discussion about accuracy of type recovery analysis (D).

TABLE II. Overview of existing compiler/packer detection tools.

tool		signatures		
name	version	internal	external	total
Lissom	1.5	2282	0	2282
RDG Packer Detector [26]	0.7.2	?	10	?
ProtectionID (PID) [27]	0.6.5.5	543	0	543
Exeinfo PE [28]	0.0.3.4	718	7076	7794
Detect It Easy (DiE) [29]	0.81	?	2100	?
NtCore PE Detective [30]	1.2.1.1	0	2806	2806
FastScanner [31]	3.0	1605	1832	3437
PEiD	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 II, most of them use a combination of precompiled internal signatures and a large external database created by the user community. The competitive solutions (except of tool DiE) 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 2282 internal signatures for all supported OFFs and we also support the concept of external signatures.

By using reverse engineering, we also figured out that several tools (e.g., PEiD, RDG) use additional heuristic technique for packer detection. These techniques are similar to our solution descripted in Section V. Using this heuristic analysis, PEiD and RDG detectors are able to detect polymorphic packers like Morphine encryptor. However, our solution achieved more accurate results in tests focused on polymorphic packers.

A. WinPE Packers

In total, 40 WinPE packers (e.g., ASPack, FSG, UPX, PECompact, EXEStealth, MEW) and several their versions (107 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 5317 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 16.

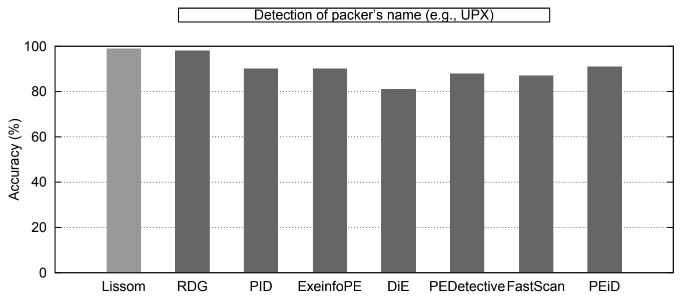

Figure 16. Summary of packer detection (packer names).

According to the results, our tool has the best ratio of packer’s name detection (over 99%), while the RDG [26] detector was second with ratio 98%. All other solutions achieved comparable results—between 80% and 91%. 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 analysis. 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 17.

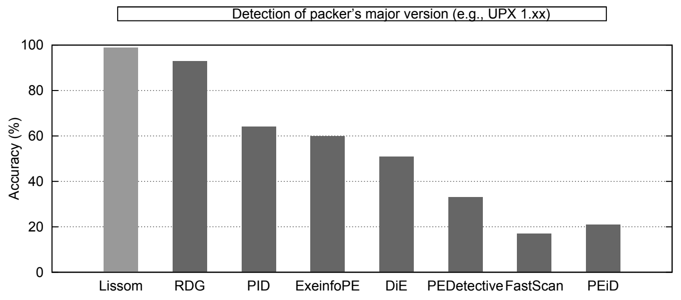

Figure 17. Summary of packer detection (packer versions).

Within this test case, our solution and RDG once again achieved the best results (Lissom scored 99%, RDG scored 93%). None of the programs has exceeded the limit of 80%. Only ExeinfoPE and ProtectionID exceeded 60% success ratio 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 for one particular packer version and their incorrect usage may lead to a decompilation failure.

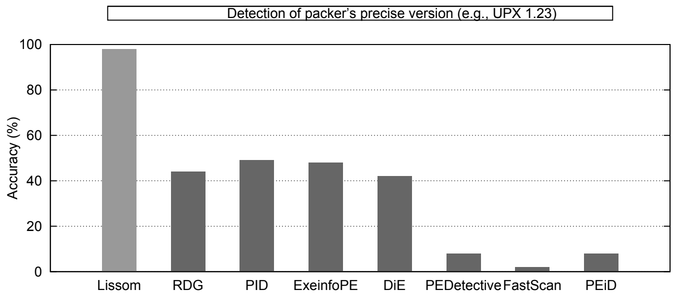

Figure 18. Summary of packer detection (detailed detection).

Based on the results depicted in Figure 18, our detector achieved the best results in this category with 98% accuracy. The results of other solutions were much lower (50% 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.

B. WinPE Polymorphic Packers

Second test suite is focused on WinPE polymorphic packer labelled as Morphine encryptor. We used two versions of this tool (v1.2 and v2.7). The test set consist of 339 executable files in total and we used the same testing methodology as in the previous case. Only three detectors (Lissom, RDG, and PEiD) are able to detect this packer. Therefore, other detectors were excluded from the results. The results are depicted in Table III.

TABLE III. RESULTS FROM TESTING OF DETECTION OF MORPHINE ENCRYPTOR.

tool	type of detection test		
name	name	major version	detailed version
Lissom	100%	100%	100%
RDG	94.69%	27.73%	27.73%
PEiD	59.88%	59.88%	35.10%

As we can see, our heuristics detection is the most successful (with ratio 100% in all cases). RDG detector exceeded 90% success ratio in detection of packer name, but in other cases its results are poor. Not even PEiD has achieved good results.

C. ELF Packers and Compilers

Last test suite for compiler/packer detectors is focused on detection of ELF compilers (e.g., GCC) and packers (ELFCrypt, UPX)—we used 18 tools and 197 files in total. Only two detectors (Lissom and DiE) supports processing of ELF OFF files. Thus, only these detectors were tested. The results are compared in Table IV.

Even in this test suite, our tool had the most accurate results in all cases. Poor performance of DiE detector in two from three test categories is probably caused by a small number of signatures for ELF OFF.

TABLE IV. RESULTS FROM TESTING OF DETECTION OF ELF COMPILERS AND PACKERS.

tool	type of detection test		
name	name	major version	detailed version
Lissom	98.98%	98.98%	87.31%
DiE	69.04%	4.57%	4.57%

D. Accuracy of type recovery analysis

Figure 19 shows the comparison of the data-type detection accuracy with and without usage of library function call prototypes. Graph does not contain column with the debugging information precision since it is used as reference for other two type sources and it would always be 100% accurate. The set of 26 real world programs written in the C programming language was compiled by the *gcc* compiler for four architectures (MIPS, x86, ARM, Thumb) and four optimization levels (O0 through O3). Debugging information generation was also enabled.

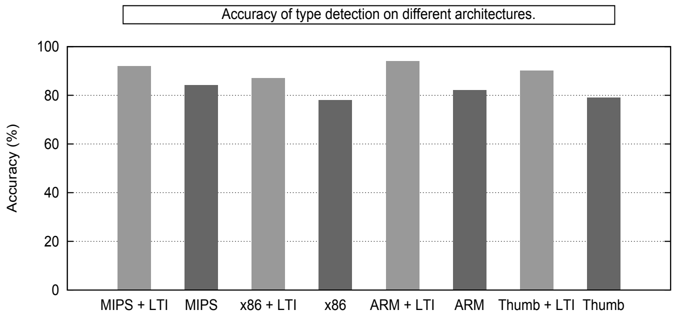

Figure 19. Summary of data-type detection.

All the resulting binaries were subsequently decompiled three times. Data-type recovery was allowed to use different type sources each time (dynamic analysis was not considered or used): (1) Any type source including debugging information. Results served as reference. (2) Enabled library function usage, but no debugging information. First column for each architecture. (3) Only type inference from the instruction semantics or as a result of some previous analysis. Second column for each architecture.

Conservative metric as described in [24] was used to determine data-type accuracy. Average precision rate for each architecture was computed from all combinations of input files and optimizations. We can see that the exploitation of library type information significantly improves type accuracy across all platforms. The improvement is all the more important given that it often provides exact reconstruction of complex well-known structures, which would never be possible just by the static type inference.

VIII. CONCLUSION

This paper was aimed on architecture-independent preprocessing methods used within the existing retargetable decompiler. We introduced methods of packer detection, unpacking, OFF conversion, and debugging information processing. Moreover, we have shown their benefits on precise data-type recovery. Up to now, this concept has been successfully tested on the MIPS, ARM, and x86 architectures within the Lissom project's [4] 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. Our solution achieved more than 98% accuracy in all test cases focused on packer and compiler detection, which was the best result of all examined tools.

We close the paper by proposing three areas for future research. (1) The unpacking phase can be enhanced by using retargetable simulators [32]. 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, heuristics, and compiler-specific analyses (e.g., better loop statement recovery, detecting different types of function calls). The process of heuristics creation can be also based on a machine-learning approach. (3) The decompilation results can be increased by extending support on C++ object-oriented debugging information and creation of new function type libraries.

Acknowledgments

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

References

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

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

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

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

[5] G. Taha, "Counterattacking the packers," in Anti-Virus Asia Researchers Conference (AVAR'07), 2007.

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

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

[8] 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.

[9] Apple Inc., "Macintosh application environment," http://www.mae.apple.com, 1994.

[10] P. Hohensee, M. Myszewski, and D. Reese, "Wabi cpu emulation," Hot Chips VIII, 1996.

[11] 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.

[12] J. Rosenberg, How Debuggers Work: Algorithms, Data Structures, and Architecture. New York, US-NY: John Wiley, 1996.

[13] E. N. Troshina and A. V. Chernov, "Using information obtained in the course of program execution for improving the quality of data type reconstruction in decompilation," Programming and Computer Software, 2010, pp. 343–362.

[14] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs." in NDSS. The Internet Society, 2011. [Online]. Available: http://dblp.uni-trier.de/db/conf/ndss/ndss2011.html#LeeAB11

[15] K. ElWazeer, K. Anand, A. Kotha, M. Smithson, and R. Barua, "Scalable variable and data type detection in a binary rewriter," SIGPLAN Not., vol. 48, no. 6, Jun. 2013, pp. 51–60. [Online]. Available: http://doi.acm.org/10.1145/2499370.2462165

[16] 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.

[17] 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.

[18] L. Ďurfina, J. Křoustek, P. Zemek, and B. Kábele, "Detection and recovery of functions and their arguments in a retargetable decompiler," in 19th Working Conference on Reverse Engineering (WCRE'12). Kingston, ON, CA: IEEE Computer Society, 2012, pp. 51–60.

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

[20] Intel Corporation, "Intel 64 and IA-32 architectures software developer's manual volume 1: Basic architecture," 2013, http://download.intel.com/products/processor/manual/253665.pdf.

[21] GCC: the GNU Compiler Collection, http://gcc.gnu.org/, 2014.

[22] Clang: A C Language Family Frontend for LLVM, http://clang.llvm.org/, 2013.

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

[24] P. Matula and D. Kolář, "Reconstruction of simple data types in decompilation," in 4th International Masaryk Conference for Ph.D. Students and Young Researchers (MMK 2013), 2013, pp. 1–10. [Online]. Available: http://www.fit.vutbr.cz/research/view_pub.php?id=10486

[25] L. Ďurfina and D. Kolář, "Generic detection of the statically linked code," in Proceedings of the Twelfth International Conference on Informatics INFORMATICS 2013. Faculty of Electrical Engineering and Informatics, University of Technology Košice, 2013, pp. 157–161. [Online]. Available: http://www.fit.vutbr.cz/research/view_pub.php?id=10461

[26] RDG Packer Detector, http://www.rdgsoft.net/, 2014.

[27] ProtectionID, http://pid.gamecopyworld.com/, 2014.

[28] ExeinfoPE, http://exeinfo.atwebpages.com/, 2014.

[29] DiE, http://www.ntinfo.biz/index.php/detect-it-easy/, 2014.

[30] NtCore PE Detective, http://www.ntcore.com/, 2014.

[31] FastScanner, http://www.at4re.com/, 2014.

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