Accurate Retargetable Decompilation Using Additional Debugging Information

Jakub Křoustek, Peter Matula, Jaromír Končický, Dušan Kolář
Faculty of Information Technology, IT4Innovations Centre of Excellence,
Brno University of Technology, Božetěchova 1/2, 612 66 Brno, Czech Republic
Email: {ikroustek, kolar}@fit.vutbr.cz, {xmatul01, xkonci01}@stud.fit.vutbr.cz

***Abstract*—In this paper, we present an extension of an existing automatically generated retargetable decompiler that is capable to parse, process, and utilize compiler-generated debugging information. This tool can be used for dealing with several security-related issues (e.g., forensics, malware analysis, vulnerability detection). Additional debugging information is used for an accurate reconstruction of platform-dependent binary applications into a well-readable high-level-language representation. The proposed solution is platform and debugging-format independent. In present, two major debugging formats—DWARF and Microsoft PDB—are supported; the extracted information is used for a recovery of several high-level constructions (e.g., variables, functions and their arguments). The proposed concept was validated by experimental results.**

***Keywords-decompilation*; *debugging information*; *PDB*; *DWARF*; *Lissom*.**

I. Introduction

Reverse engineering is an old method used for analysis and reconstruction of a given object. In information technology, a lot of reverse-engineering tools have been created for understanding and reconstruction of binary software, e.g., disassemblers, dumpers, and sniffers. However, the result of mentioned tools is usually a very low-level representation of the input object, e.g., assembler code, raw dump of memory, or data stream. Therefore, it is hard to understand its meaning. The reconstruction of the original object is not automatic and it requires manual re-implementation based on the gathered information.

A machine-code decompiler (i.e., reverse compiler) is a more advanced reverse-engineering tool. It can theoretically produce the most readable and re-compilable code; it transforms the input binary executable application into a particular high-level-language (HLL) representation. In software maintenance, this process can be used for source code recovery, translation of code written in an obsolete language into a newer language (see [1] for such an application), migration of binary code to a new platform, injection of additional features into an existing application, etc.

However, decompilation is typically used in the field of computer security and forensics. Within this field, decompilation is used for compiler verification (i.e., verification of code and debugging information generated by compiler in software-critical systems, since the compiler cannot be trusted in these systems [2]), vulnerabilities detection, and for malicious code analysis and its understanding.

In comparison, disassemblers have been used for malware analysis in the past decades, but decompilers are used more and more often last years. Their primary advantage over disassemblers is a more readable output (i.e., HLL code) that is independent on a particular processor architecture. Moreover, it is possible to precisely analyse code without a deep knowledge of underlying architecture (e.g., processor resources, instruction set, micro-architecture). This is very useful because of a massive expansion of malware to the new platforms (e.g., smartphones, tablets, or even cars [3]). Therefore, for a malware analyst, it is not necessary to learn all the details about these platforms.

However, the machine-code decompilation is very complicated task because the input applications are often obfuscated by packers or heavily optimized by compilers. Therefore, it is necessary to take advantage of every available information. The decompiled code must meet two essential criteria—it has to be functionally equivalent to the input binary code, and it has to be highly readable. It is a great advantage if the decompiled code is re-compilable too.

In this paper, we present a concept of a debugging-information exploitation within a decompilation process. The proposed solution allows extracting additional information from two major debugging formats—DWARF and Microsoft PDB. It is possible to find a lot of valuable information within these two formats, such as lists of original source files, line numbers, functions, or variable names and their types.

Such information is used for the recovery of several high-level constructions (e.g., functions together with their arguments and local variables) in an existing automatically generated retargetable decompiler developed within the Lissom project [4]. This decompiler is independent on a particular target architecture, operating system, or origin of the decompiled application (i.e., the used programming language and its compiler). This tool can be used in every previously mentioned area.

The paper is organized as follows. Section II discusses existing debugging information formats. Then, we briefly

 ISBN: 978-1-61208-209-7

describe the retargetable decompiler developed within the Lissom project in Section III. The exploitation of debugging information within the decompiler is then presented in Section IV. Section V describes the state of the art of debugging information exploitation in decompilation. Experimental results are given in Section VI. Section VII closes the paper by discussing future research.

II. Debugging Information Formats

Despite all programmers efforts, any meaningful program contains bugs or defects that need to be found and fixed. This process is called debugging and it can be done in several different ways (e.g., code insertion, stepping through instructions, displaying registers or memory). The goal of today's debuggers is to provide source-level approach, which matches lines of source code with machine-code instructions. For this purpose, debugged executables or libraries must contain additional information to identify the corresponding positions, variables, functions and other useful details in the form of debugging data format. This section introduces two of today's the most common formats.

A. DWARF (Debugging With Attributed Record Formats)

DWARF [5] is a debugging data format originally developed in the mid-1980s at Bell Labs for Unix System V Release 4. Even though it is mainly associated with ELF [6], it is independent on the used object file format, programing language, operating system, or target architecture (e.g., MIPS, ARM, Intel x86). The latest, fourth version was published in 2010, but its second version (DWARFv2) is still the most commonly used one among compilers and debuggers.

DWARF information is stored in special sections in the same object file as machine code. Each of these sections contains different kind of debugging data, such as basic file information, line numbers, look-up tables, call frames, and macros. The whole program is represented as a tree, whose nodes can have children or siblings. Each node is a *Debugging Information Entry* (DIE) structure that has a tag and a list of attributes. A tag specifies the type of the DIE (e.g., function, data type, variable) and attributes contain all the description details. There are two general types of DIEs:

- Those describing data and types such as base types, variables, arrays, structures, classes, etc. Locations of data are defined by location expressions, consisting of a sequence of operations and values, evaluated by a simple stack machine.
- Those describing executable code. Functions (they have a return value) and subprograms are treated as two flavours of the same construction. These DIEs typically own many other DIEs that describe them (e.g., parameters, local variables, lexical blocks).

There is also a special type of DIE called compilation unit which is the parent for all DIEs created by a separate compilation of a single source file. It holds information about compilation (e.g., name of the source, language, producer) and locations of other DWARF data not described by DIEs (e.g., line number table, macro information).

DWARF is generated by the most major compilers (e.g., `gcc`, LLVM `llc`, Intel `icc`), supported by nearly all debuggers, and there are several libraries and utilities that allow its examination or manipulation (e.g., `libdwarf`, `dwarfdump`, `objdump`).

B. PDB (Program Database)

PDB is a format developed by Microsoft Corporation. It is generated only by Microsoft Visual Studio compilers and used by Microsoft debuggers. PDB is mainly used on the ARM and Intel x86 architectures and it is based on the older CodeView format. PDB debugging information for a particular application is stored in a separate file with the `pdb` extension. Each executable binary file in a PE format [7] has its counterpart in a PDB file. Both files are paired by a unique GUID code.

PDB is a proprietary format, and its specification has never been publicly published. Therefore, the analysis of this format can be only done via reverse engineering. A PDB file structure is similar to a file system. A PDB file consists of many streams (i.e., sub-files), each of them contains a different kind of information. There is a stream with type information (e.g., common types, enumeration, structures), compilation units (i.e., modules), symbol tables and PE sections. Furthermore, each module has its own stream with information about functions, local and global variables, and line number information within the module.

The official DIA SDK is provided for processing debugging information [8]. However, this toolkit is platform dependent because it can be used only under the MS Windows operating system.

C. Other Formats

Except the two aforementioned formats, there are some other formats that can be used to store debugging information. The best known are *Symbol Table Strings* (STABS), where data are stored as special entries in symbol table, and *Common Object File Format* (COFF) [9], that could contain debugging information in special sections much like DWARF. Both were strongly tied to specific object file formats and became obsolete along with them.

III. Lissom Project's Retargetable Decompiler

In this section, a brief description of an existing automatically generated machine-code retargetable decompiler is presented. This tool is developed within the Lissom project at Brno University of Technology [4]. It is supposed to be independent on a particular target architecture, operating system, or executable file format. See [10] for its detailed

 ISBN: 978-1-61208-209-7

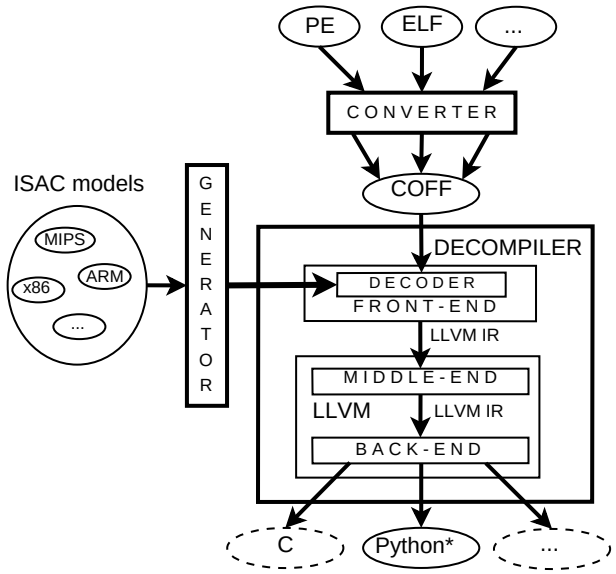

Figure 1. The concept of the Lissom project's retargetable decompiler.

description. The decompilation process consists of four phases, see Figure 1.

(1) At first, the input binary executable file is transformed using a *binary file converter* from a platform-specific executable format (e.g., Windows PE, Unix ELF) into an internal, uniform COFF-based file format, see [11] for details.

(2) The subsequent part of the decompiler is a *front-end*, which is its only platform-specific part because of the instruction decoder that is automatically generated based on the target architecture model (e.g., MIPS, ARM, Intel x86). The architecture description language ISAC [12], developed also within the Lissom project, is used for this purpose. The front-end analyzes and transforms the input application (i.e., its machine code, data and other information) into the LLVM assembly language code, LLVM IR [13], which is used as an internal code representation of decompiled applications in the remaining decompilation phases.

(3) Afterwards, this program representation is optimized in a *middle-end* using many optimization passes.

(4) Finally, the program intermediate representation is emitted as the target HLL in a *back-end*. Currently, a Python-like language is used for this purpose, while a support of other target languages is under development (e.g., the C language). Both middle-end and back-end are built on the top of the LLVM Compiler System [14]. This language is very similar to Python, except a few differences (e.g., we use several C-like constructs, address and dereference operators).

IV. Debugging Information Exploitation Within The Retargetable Decompiler

In this section, the concept of the DWARF and PDB information parsing, processing, and utilization within the Lissom project's retargetable decompiler is discussed.

The debugging information is parsed by two different parsers, the first for DWARF and the second for PDB. Afterwards, the parsed information is processed and stored in a debugging-format-independent way. This information is utilized for an enhancement of the LLVM IR code representation within the front-end part of the decompiler; see Figure 2 for details. Finally, this code representation is passed to the subsequent decompiler parts, see Section III for details.

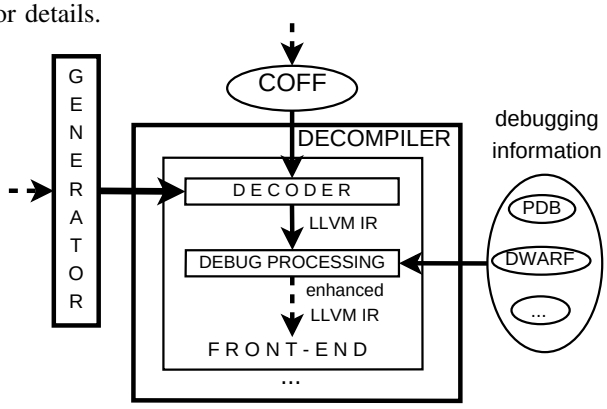

Figure 2. Debugging information processing within the front-end.

A. DWARF Parsing

Parsing of debugging information from an object file to structures defined in the DWARF specification is done by the `libdwarf` [15] library. Original `libdwarf` is using `libelf` to access sections of interest in ELF binaries. However, the input of our decompiler are uniform COFF-based objects. In order to parse information from such files, we exploited `libdwarf` object access capabilities and provided a new interface using our internal object manipulation library.

`libdwarf` creates a low-level representation of debugging data, whose usage in a decompiler or debugger would be very complicated and unintuitive. That is why we decided to create a new, mid-layer library called `dwarfAPI`, that builds high-level, object-oriented data structures and provides convenient access methods. `dwarfAPI` represents all debugging information about a binary file as one entity consisting of several vectors of objects such as compilation units, lines, types, functions and global variables. Each of these objects contains complete information about itself and about all the other objects they own (e.g., local variables in functions). This way, the information that was in a low-level representation scattered among multiple DIEs is grouped together, what allows very natural and easy way of processing complex DWARF structures. It should be mentioned that `dwarfAPI` is not limited to a particular DWARF version or target architecture.

B. PDB Parsing

The only available official toolkit for PDB parsing—DIA SDK—is unusable for our aims because of the limitations

discussed in Section II. Therefore, our own PDB parser has been created. At first, it was necessary to reverse-engineer the PDB format and analyze its internal structures. For this purpose, we reused an existing unofficial PDB analyzer [16].

The PDB file parsing consists of two steps. (1) At first, the streams have to be extracted and separated. They are divided into constant-size data blocks. At the end of the PDB file, there is a root directory, which stores each stream's size and the indexes of used data blocks. (2) The main stream processing is done in a consequent step. Most of the streams are organized into *symbols*, which are data structures with a type, size, and data.

While processing all the symbols, the parser fills the internal data structures. For example, we can extract and process an address, length, return type, arguments, local variables and a line number information (i.e., mapping between machine-code and HLL code location) for each function described in the PDB file.

Meaning of many data entries depends on a context (i.e., the previous data entries). The information stored in a particular stream is often interconnected with another stream. For example, each data type has an index to its definition—base types (e.g., `int`, `float`) have a predefined index, but the user types are defined in a type information stream.

In present, we are able to extract and utilize most of the PDB streams, but a few remaining streams are still unknown for us. As well as the DWARF parser, the PDB parser is not limited to a particular target architecture (e.g., ARM, Intel x86).

C. Decompiler

After the debugging information is extracted by a parser, it is ready to be used in the decompiler's front-end, which updates its intermediate code representation (LLVM IR) based on this extracted information. In the following paragraphs, we depict the useful information and its usage within the front-end.

Module information: Based on the type of the original HLL, the decompiled binary application is created from one or more *modules* (e.g., source files). The debugging information is usually divided into smaller pieces, where each piece corresponds to one particular module. This is useful for the decompiler because it is possible to divide the resulting code into similar modules, e.g., the decompiler can divide the decompiled code into several files with the original names.

Function information: Probably the most important debugging information is related to functions. A proper function detection and recovery is a crucial task of each decompiler. Otherwise, the unstructured and hardly readable ("spaghetti") code will be generated (e.g., code containing `goto` statements). The Lissom project's retargetable decompiler utilizes its own function detection heuristics based on the principles described in [17], [18], [19]. However, the function's debugging information is used instead whenever it is available because it is essentially more precise. For example, it is possible to obtain the function location within the machine code, its name, the number of its arguments, their names and types, and many others.

Variables: Our retargetable decompiler is able to detect both global and local variables, and guess their types. In the resulting code, it generates variable names from a fixed list of well-readable names, see [10] for details. However, the decompiler emits the original name whenever it is stored in the debugging data. This is just a detail, but it gives the feeling of listing the original code. Furthermore, both DWARF and PDB support storage of variable types, too. This is handy because the recovery of composite data types is a non-trivial task, see [20].

Line Information: Mapping of source-code locations to the machine-code is done via so-called *line information* and it is an essential part of each debugger. However, its usage within decompilation is not so important. It can be only used for re-formatting the decompiled code to better fit its original format (e.g., splitting the complex expressions into more statements).

In present, the decompiler uses the function and variable debugging information, while exploitation of other information is under development.

V. Related Work

We can find two existing decompilers that exploit debugging information. Both of them support more target architectures; however, none of them is truly retargetable because the support of such architectures is hand-coded by their authors.

The Hex-Rays Decompiler [21] supports decompilation of the ARM and Intel x86 target architectures. It allows using PDB debugging information for enhancing the generated C code. However, this proprietary software probably uses the Microsoft DIA SDK [8] or Microsoft Debugging Tools for Windows (see their limitations in Section II) for PDB parsing and processing. Therefore, this feature is only available in the MS Windows version of this software. The Unix version of Hex-Rays Decompiler needs a remote MS Windows server with this library for gathering PDB information. This decompiler can exploit the PDB debugging information for reconstruction of functions, their arguments, global variables, etc. However, recovery of local variables is not always correct (details about this problem are described in Section VI).

The REC decompiler [22] has a more extensive list of supported target architectures (e.g., MIPS, PowerPC). According to the official homepage [22], the DWARFv2 support is in an early stage of development. However, it is supported only for a subset of the supported architectures. Therefore, it is plausible that their DWARF parser is also not

 ISBN: 978-1-61208-209-7

retargetable and it needs to be manually re-configured for each given architecture. Finally, the PDB support is planned as well.

As can be seen, the debugging information is a valuable resource of additional information and it is already exploited by existing decompilers. However, their current implementation is limited for several reasons.

VI. Experimental Results

To demonstrate several selected abilities and advantages of our solution, this section presents a decompilation of a simple program calculating factorial function for the ARMv4 architecture. The C source code for this program is given in Figure 3. It was compiled using Microsoft Visual Studio 2005 compiler with enabled PDB debugging-information (`/Zi`) and disabled optimizations (`/Od`).

```
#include <stdio.h>

volatile int value = 6;

int fact(int n)
{
   if (n==0)
      return(1);
   else
      return n*fact(n-1);
}

int main(int argc, char *argv[])
{
    int a = fact(value);
    printf("%d\n", a);
    return a;
}
```

Figure 3. Source code in C.

The size of the generated PDB file is 1.07MB. Its processing by our parser is almost instant (approximately one second on Intel Core i5 with 3.3 GHz, 8GB RAM running Windows 7 64-bit operating system). The resulting HLL code generated by our decompiler can be seen in Figure 4. Observe the following key aspects of our solution.

(1) With the use of debugging information, we are able to completely recover functions, i.e., their names, arguments, and their mapping to a machine code (e.g., `fact`, `main`). Our other algorithms of functions, arguments, and return values detection that do not use debugging information (briefly mentioned in Section IV) achieve 80-90% accuracy of detection. However, the detection method based on debugging information is absolutely precise as long as a compiler generates the correct debugging information.

(2) Furthermore, the recovery of global and local variable names is also possible in most cases (e.g., `value`, `a`, `n`). The only limitation is for local variables because LLVM optimizations, used in the decompiler's middle-end, sometimes remove them in the resulting code. On the other hand, the optimizations can produce variables that were not in the original code; therefore, they are missing in the debugging information. In such situations, the decompiler assigns more appropriate names to such variables than just their addresses.

```
# -------- Global Variables --------
value = 6

# -------- User Functions ----------
def fact(n):
    if n == 0:
        return 1
    return n * fact(n - 1)

# -------- Main Function -----------
def main(argc, argv):
    a = fact(value)
    printf("%d\n", a)
    return a
```

Figure 4. Decompilation result in the Python-like language.

In the real world, the debugging information is not always available for the decompiled binaries. Its presence strongly depends on a type of decompiled binary application and purpose of its decompilation. In a case of source code recovery, binary code migration, or vulnerability detection, there is a very good chance that the debugging information is present because applications are usually not stripped, obfuscated or protected. Presence of debugging information within the decompiled application is guaranteed in the case of compiler verification.

On the other hand, malware rarely contains such additional information. The main reason is that this harmful code is stripped after creation (i.e., the unnecessary information is removed). Furthermore, the application is often obfuscated, ciphered, and packed by some of the existing packing or protecting software. Based on our internal malware database, only a few percents of malware contain debugging or symbolic information. However, as has been said in the introduction, the decompilation is so much complicated process that every possible additional information must be exploited.

Finally, it should be noted that decompilation of compiler-optimized application can produce a more readable code than decompilation of a non-optimized application in some cases. For example, it is easier to track values in registers (i.e., optimizations enabled) than in memory (i.e., optimizations disabled) for the decompiler. However, this is possible only for several compilers because debugging-information generation sometimes disables the most aggressive optimizations.

VII. Conclusion and Future Work

This paper has proposed an extension of the existing retargetable decompiler which produces a more accurate decompiled code. The presented solution is based on the exploitation of compiler-generated debugging information whenever it is available. The available debugging information is utilized for a precise recovery of several HLL constructions and enhancement of code readability. In present,

 ISBN: 978-1-61208-209-7

two common debugging formats are supported—DWARF and Microsoft PDB.

The fundamentals of debugging information parsing, processing, and utilization were briefly discussed and we have presented results of the current state of the implementation. As can be seen, we are already able to use debugging information for the recovery of several HLL constructions (e.g., functions, their arguments, variables). The resulting code is a highly readable code in a Python-like language that can be used in many areas related to software maintenance and security.

However, there is still a lot of space for improvements. In the first place, it is necessary to finish the reverse-engineering analysis of the remaining PDB streams. Moreover, the decompiler does not use several extracted information (e.g., source-code line information, information about modules) at the moment. After that, we will be able to produce a more accurate LLVM IR code, which will improve the resulting HLL code.

ACKNOWLEDGMENT

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

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

[3] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, and T. Kohno, "Comprehensive experimental analyses of automotive attack surfaces," in *20th USENIX conference on Security (SEC'11)*. USENIX Association, 2011.

[4] Lissom, `http://www.fit.vutbr.cz/research/groups/lissom/`, [retrieved: June, 2012].

[5] *DWARF Debugging Information Format*, 4th ed., DWARF Debugging Information Committee, 2010.

[6] TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification," 1995.

[7] Microsoft Corporation, "Microsoft portable executable and common object file format specification," `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`, [retrieved: June, 2012], version 8.2.

[8] ——, "Debug interface access SDK," `http://msdn.microsoft.com/en-us/library/ee8x173s.aspx`, [retrieved: June, 2012].

[9] G. R. Gircys, *Understanding and Using COFF*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1988.

[10] L. Ďurfina, J. Křoustek, P. Zemek, D. Kolář, K. Masařík, T. Hruška, and A. Meduna, "Design of a retargetable decompiler for a static platform-independent malware analysis," *International Journal of Security and Its Applications*, vol. 5, no. 4, pp. 91–106, 2011.

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

[12] K. Masařík, *System for Hardware-Software Co-Design*, 1st ed. Brno, CZ: Faculty of Information Technology BUT, 2008.

[13] LLVM Language Reference Manual, `http://llvm.org/docs/LangRef.html`, [retrieved: June, 2012].

[14] The LLVM Compiler System, `http://llvm.org/`, [retrieved: June, 2012].

[15] D. Anderson, "Libdwarf and dwarfdump," `http://reality.sgiweb.org/davea/dwarf.html`, [retrieved: June, 2012].

[16] "Parsing library for PDB file format," `http://code.google.com/p/pdbparser`, [retrieved: June, 2012].

[17] D. Kästner and S. Wilhelm, "Generic control flow reconstruction from assembly code," *ACM SIGPLAN Notices*, vol. 37, no. 7, 2002.

[18] J. Kinder, F. Zuleger, and H. Veith, "An abstract interpretation-based framework for control flow reconstruction from binaries," in *10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'09)*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 214–228.

[19] N. Bermudo, A. Krall, and N. Horspool, "Control flow graph reconstruction for assembly language programs with delayed instructions," in *5th IEEE International Workshop on Source Code Analysis and Manipulation SCAM'05*, 2005, pp. 107–116.

[20] K. Troshina, Y. Derevenets, and A. Chernov, "Reconstruction of composite types for decompilation," in *10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'10)*. IEEE Computer Society, 2010, pp. 179–188.

[21] Hex-Rays Decompiler, `www.hex-rays.com/products/decompiler/`, [retrieved: June, 2012].

[22] Reverse Engineering Compiler (REC), `http://www.backerstreet.com/rec/rec.htm`, [retrieved: June, 2012].

 ISBN: 978-1-61208-209-7