Design of an Automatically Generated Retargetable Decompiler

Lukáš Ďurfina, Jakub Křoustek, Petr Zemek, Dušan Kolář, Tomáš Hruška, Karel Masařík, Alexander Meduna
Faculty of Information Technology, Brno University of Technology
Božetěchova 1/2, 612 66 Brno, Czech Republic
{idurfina, ikroustek, izemek, kolar, hruska, masarik, meduna}@fit.vutbr.cz

Abstract: This paper presents a concept of a retargetable reverse compiler (i.e. a decompiler). This tool translates platform-specific binary applications into a high-level language (HLL) representation. A Python-like language was chosen as the target language. Our unique solution is automatically generated from the target platform description in the architecture description language (ADL) ISAC. The decompiler core is build on top of the LLVM Compiler System. As can be seen from the proof of concept, we are able to automatically generate a decompiler producing a highly readable HLL code for a Sony® PlayStation® Portable (PSP™) platform while preserving the functional equivalency with the original application.

Key–Words: decompilation, reverse engineering, malware, LLVM, Lissom, ISAC

1 Introduction

We can find several methods of decompilation (e.g. decompilation of machine code or bytecode) as well as their usage in reverse-engineering (e.g. source code reconstruction, binary code migration from a particular processor architecture to another, malware analysis). The most traditional is machine code decompilation into a HLL code, which represents the topic of the present paper.

The era of machine code decompilers was established more than 50 years ago. From this period, various kinds of decompilers have been created. They used to differ in supported architectures or in the format of input executables. The best known decompilers of this time are the *dcc* decompiler [1] from C. Cifuentes, the open source *Boomerang* decompiler [2], the *REC Decompiler* [3], and the *Hex-Rays Decompiler* [4]. The first two decompilers are not developed any more, but the second two are constantly improving. The comparison of these applications can be found in [5].

The existing decompilers are more or less bounded to a particular platform (i.e. instruction set and operating system). Every additional platform has to be manually implemented, because these decompilers do not have any automated generic solution.

In this paper, we present an automatically generated retargetable decompiler, which is not bounded to any particular target platform and it can be utilized for source code recreation, static malware analysis, etc.

Our retargetable decompiler is based on exploiting ADL ISAC [6], which is intended to be used for designing new application-specific instruction set processors (ASIPs). We use this formalism for describing existing platforms. The front-end of the decompiler is generated from this description. The decompiler core is based on the LLVM Compiler System [7], which we use for a reverse translation from a binary form into a Python-like language.

The LLVM Compiler System was designed as an innovative compiler framework. The key features of LLVM include a language-independent instruction set, many built-in optimization algorithms and passes, intermediate representation (LLVM IR), and support of several programming languages. For more information, see [7] and the references given therein.

This paper extends the basic concept presented in [5]. In comparison with the aforementioned paper, the decompiler is now automatically generated, the statically linked code detection has been further improved by function-type information, and our tool is now able to decompile binary code of real-world platforms.

The proof of our concept is presented on a Sony PSP handheld [8]. We chose this widespread device to demonstrate the flexibility of our solution, since it is not supported by any existing decompiler (except some basic support in [3]). From the decompilation point of view, the PSP is based on a non-classical processor architecture (i.e. not Intel x86), but it still uses some hard-to-decompile features like indirect jump instructions, stack, and compiler-specific application binary interface (ABI). Currently, we are able to automatically generate a decompiler for this architecture. This decompiler is capable to decompile applications for this platform and it produces a highly-readable HLL code, while still preserving the functional equivalence of the code.

Organization of this paper is as follows. Sec-

```
RESOURCES {                    // HW resources
  PC REGISTER bit[32] pc; // program counter
  REGISTER bit[32] gprs[32];// register file
  RAM bit[32] memory {ENDIAN(LITTLE);};
}
OPERATION reg REPRESENTS gprs
  { /* description of registers */ }
OPERATION op_add {// instruction description
  INSTANCE gprs ALIAS {rd, rs, rt};
  ASSEMBLER { "ADD" rd "," rs "," rt };
  CODING { 0b000000 rs rt rd 0b000001...};
                    // instruction behavior
  BEHAVIOR {gprs[rd] = gprs[rs]+gprs[rt];};
}
```

Figure 1: Example of an ISAC language source code.

tion 2 describes the ISAC language as well as other ADLs. Then, Section 3 briefly discusses the Sony PSP platform and its unofficial Software Development Kit (SDK). The design of our decompiler is then presented in Section 4. Experimental results are given in Section 5. Section 6 closes the paper by discussing future research.

2 ISAC Language

We can find a whole scale of architecture description languages. These languages are usually used within projects focused on development of a complete integrated development environment for the processor design, e.g. ArchC, SystemC, LISA, or nML. All of these languages are primarily used for development of a new processor architecture. However, we can also find languages used for describing existing architectures. The SLED language, developed within the New Jersey Machine-Code Toolkit [9] project, is an example of this language category. This project exploits the SLED language [10] for intuitive description of instruction syntax and coding. However, this language does not support description of instruction semantics. Therefore, this language itself cannot be used for generation of tools like compilers or decompilers.

The ISAC language [6] is used for the description of Multiprocessor Systems-on-Chip. It was developed within the Lissom project at Brno University of Technology [11]. The basic scope of this project is a transformation of a processor models into advanced software tools (e.g. a C compiler [12], a simulator), or into a hardware realization of each processor. The ISAC processor model consists of several parts. In the resource part, processor resources, such as registers or memory hierarchy, are declared. In the operation part, processor instruction set with behavior of instructions and processor micro-architecture is specified.

The *assembler* and *coding* sections capture the format of instructions in the assembly and machine language. For the behavioral model, the *behavior* section is used. In this section, a subset of the ANSI C language can be used. The behavior section defines the semantics of each operation.

We chose the ISAC language for description of architectures targeted by our decompiler. The main reason is that it supports complete description of resources and the instruction set, and it is well prepared for instruction-semantics automated extraction [12]. The extracted semantics was initially used only for an automatic generation of the C compiler, but we exploit such information also for an automatic generation of the retargetable decompiler (see details below).

3 Sony PSP

The PlayStation Portable (PSP) is a video-game handheld console manufactured by Sony Corporation since 2004. It has been remodeled three times, resulting in the final version called PSP Go.

The PSP is powered by a dual-core processor based on a RISC processor MIPS-4000 [13], which is enhanced by several specialized registers and instructions. The first core is used for standard system functions. The second core is utilized as *Media Engine Chip* for multimedia processing [8].

Software for the PSP is created either as official applications, which are distributed on Universal Media Disc (UMD) or over the PlayStation Network (PSN), or as the so-called home-brew applications created by unofficial tools (e.g. PSP-SDK [14]). The PSP is running proprietary plugin-based operating system from its firmware. Application code is stored in the PRX format, which is based on the standard Unix ELF format, but differs in program headers and it uses specific relocation types. The code is stored along with other application resources in the PBP file format.

The ISAC model of the PSP architecture has been created for the purpose of decompilation. Since the code of the second core is used only for visualization of multimedia data and not for computation of main program code, only the first core is described. The model has approximately 2000 lines of code in the ISAC language and additional 1100 lines in auxiliary C++ source files describing mainly instruction semantics. A simple instruction of the PSP architecture with its behavior is shown in Figure 1 (instruction *add*).

4 Retargetable Decompiler

In this section, the concept of the retargetable decompiler is discussed. This tool converts platform-dependent binary applications into a uniform HLL code. This tool is automatically generated based on the architecture model in the ISAC ADL.

The structure of the retargetable decompiler is similar to a classical compiler. It consists of a front-end, a middle-end, and a back-end (see Figure 2).

ISBN: 978-1-61804-056-5

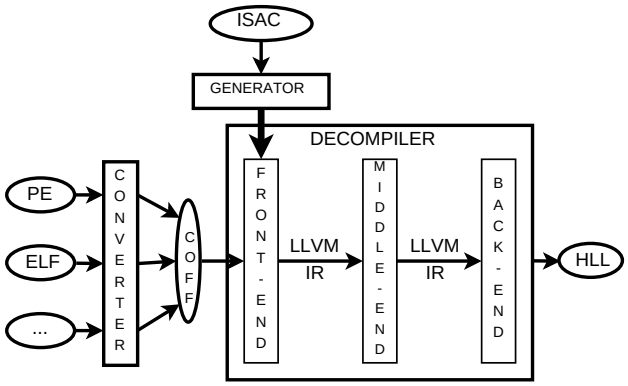

Figure 2: The concept of the retargetable decompiler.

```
instr instr_add__gpregs__gpregs__gpregs,
  //semantics
  %tmp1 = i32 read_register(gpregs, 1);
  %tmp2 = i32 read_register(gpregs, 2);
  %i    = add(%tmp2, %tmp1);
  write_register(gpregs, 0) = %i;,
  //syntax
  "ADD" 0~"," 1~"," 2,
  //coding
  0b000000 1=b[5] 2=b[5] 0=b[5] 0b000001...
```

Figure 3: Example of an extracted semantics.

The only platform-specific part is the front-end. For this purpose, the binary coding and semantics of each processor instruction is extracted from the architecture model in ISAC. This is a major difference against other retargetable decompilers, because it is not necessary to manually reconfigure the decompiler for a new architecture. It should be noted that in present, there is no other competitive method of automatically-generated retargetable decompilation.

4.1 Front-end

The very first step of retargetable decompilation is modeling the target architecture in the ISAC ADL. The user has to describe processor resources and its instruction set in this language. Afterwards, it is possible to extract the semantics and binary encoding of each instruction via the *semantics extractor* [12]. The semantics extractor transforms ANSI C code from the behavior section of the instruction into a sequence of LLVM IR-like instructions which properly describe its behavior. Therefore, we are able to map instruction semantics to its machine code. An example is depicted in Figure 3. This semantics is extracted from the behavior section illustrated in Figure 1. The *instruction decoder* further converts this representation into a proper LLVM IR form (see below).

The input file of a decompiled application is stored in a platform-specific binary file format (e.g. Windows PE, Sony PRX). Therefore, it is necessary to convert it into a unified form of representation. An internal COFF-based file format has been designed for this purpose, together with several conversion algorithms. The Binary File Descriptor library (BFD) [15] is used for describing the platform-specific binary file formats. Most of current binary file formats are derivatives of Unix ELF or COFF formats; therefore, those formats are supported by the BFD out of the box. The description of non-standard file formats is done via a plugin-based system, where the user implements a few necessary functions. The conversion uses this file format description and it transforms the application in a uniform way. Currently, the Windows PE, Unix ELF, Symbian E32, and Android DEX file formats are supported.

In the next step, the *instruction decoder* for the particular architecture is automatically generated based on the extracted semantics and binary coding. The instruction decoder is responsible for translating architecture-specific binary machine code into an internal code representation as a sequence of low-level LLVM IR instructions (i.e. a basic block with several LLVM IR instructions for each input machine instruction). As we can see, its functionality is similar to a disassembler, except that its output is not an assembly language, but rather the semantics description of each instruction. This part has to deal with platform-specific features. For example, it has to support architectures with different endianess or delay-slots. All aforementioned features are supported in our solution.

The front-end of the decompiler has an ability to recognize statically linked library code. Our solution is inspired by FLIRT [16]. Due to this feature, we can restrict the amount of code needed to be decompiled and reach higher accuracy of the result. A precondition for such a detection is an exact description of that library code. We have developed tools that are able to create signatures of static libraries. Based on these signatures, we are able to recognize which parts of an executable belong to statically linked libraries.

Each static library consists of object module files of some file format, such as PE or ELF. We use a converter to transform these module files into our unified object file format. In the next step, a pattern is extracted from each module. The pattern contains first 32 bytes of a module, CRC code, total length, and the names and addresses of exported public symbols and relocations. The relocations dictate the positions which will be updated by a linker. Therefore, we have to mark such bytes with a special sign as variable. These bytes will be skipped in comparison. Moreover, these bytes cannot form a part of a module used for CRC code calculation. Therefore, CRC code is calculated for various long parts, this part starts after the first 32 bytes and ends on the first relocation, note that the part can have zero length, when module is smaller then 32 bytes or also if there is a relocation

immediately after the first 32 bytes.

It may seem that these patterns can be sufficient for searching for the library code, but this presumption is false. The main problem of patterns is that they can be ambiguous. For example, they can have the same bytes at the beginning, the same length and the same CRC code, and they are just different in the names of relocations. Unfortunately, our system is not able to distinguish code which is described by such patterns and, therefore, these patterns are not included in signatures. An exclusion of such patterns is made when patterns are processed into a signature. In that process we build a tree where each collision is easily detected and patterns are excluded.

Another reason for creating a more precise level of description is efficiency. In patterns, some parts of the bytes at the beginning of modules use to be identical. If we unite these same prefixes, we can save memory and make searching faster. To remove these prefixes, we utilize a tree, where the same prefixes are joined into a single node. A tree can be branched into more levels and nodes can have more than two children. Also, the lengths of prefixes can be variable. The length of compared prefixes depends on the length of the shortest instruction size on the used architecture, e.g. on i386, it is a single byte, but on PSP (i.e. MIPS), it is four bytes.

The successfulness of a signature can be influenced by the order of patterns in it. Imagine a situation where there are two patterns in a signature. The first one for a bigger module with CRC code for a dozens of bytes, and the second for a small module, which is described only by bytes at its beginning and also contains variable bytes due to relocation. It is more possible to get false positives with the second pattern because it is less descriptive. This potential problem is eliminated by assigning a score to patterns according to the quantity of information provided by the pattern. Quantity of information is influenced mainly by the length of code, which is used for calculating CRC code and for cases with zero length of this part the number of relocations in the first 32 bytes is taken into account. The score is used for pattern sorting in the signatures and, therefore, the most descriptive patterns are used first.

After the recognition of a library code, we know the addresses and the names of functions, but we do not know what is the return value type, the number of arguments, and the types of these arguments. This information is stored in the header files of libraries. We have created a tool which is able to extract these data and prepare them in a suitable format for our decompiler. Header files contain usual C types, such as `int`, `double`, or `float`, but also type aliases introduced by `typedef`. We have to transform all these types into LLVM IR types. These requirements resulted into a use of the Clang library [17] as a base for this tool. This library provides a complete front-end for the C and C++ languages, and it is straightforward to integrate the code from the Clang compiler for the transformation of C types into LLVM IR types.

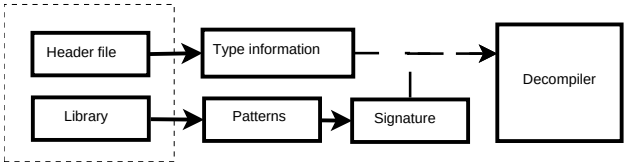

Figure 4: The toolchain for recognizing statically linked code

As an example, consider the following extracted type information: `sum i32 2 i32, i32 # int sum (int, int).` First, there is a function name followed by its return type. After that, there is the number of arguments with comma separated types. This is enough for the decompilation process, but to enhance the readability of the output code, we also store the original declaration of the function. This piece of information is then present in the decompiled code. The statically linked code recognition is shown in Figure 4. Library and a header file with an interface for this library are in the dashed box.

In the final part of the front-end, a static analysis of the emitted LLVM IR code is performed. This analysis is focused on transforming the LLVM IR code to produce more suitable and annotated code for the following parts of the decompiler. As an example of this part, we can mention the elimination of `J $31` instructions, which mean indirect jump to the address stored in register `$31`. This instruction is typically used for returning from a function. Another example is an application of detected ABI to function calls. With properly detected ABI, it is possible to remove boilerplate code related to function arguments passing and returning of function result. It should be mentioned that automatic ABI detection is not implemented yet, and more intensive research in this direction is necessary.

4.2 Middle-end

In this stage, we have a very low-level LLVM IR of the input binary. Each basic block represents a single assembly instruction, there may be many redundant instructions (recall that each assembly instruction is decompiled in isolation), and there is no evidence of high-level constructs, such as loops. The key role of the middle-end part of our decompiler is to improve the properties of the generated LLVM IR code and prepare it for the final emission of the output HLL. The following three types of passes are performed over the LLVM IR code. (1) Search for idioms. These are sequences of code whose com-

bined semantics is not immediately apparent from the instructions' individual semantics. (2) Retrieval of high-level constructs, such as `if/else` statements and loops. (3) Optimization of the code using many built-in optimizations available in LLVM and our own passes (e.g. reassociation of expressions, optimizations of loops, renaming of variables).

4.3 Back-end

In this final decompilation stage, we transform the optimized LLVM IR into a HLL. We currently use a Python-like language as the target language, briefly described next. However, a support for different back-ends is planned.

Our HLL is non-typed, block structured, and uses whitespace indentation, rather than curly braces or keywords, to delimit blocks. Since we focus on code analysis by humans, the used language emphasises code readability. Whenever there is no support in Python for a specific construction, we use C-like constructs. Instead of arrays, we use lists, and instead of structures, we utilize dictionaries. We also use the address and dereference operators from C. As there are cases when the code cannot be structured by high-level constructs only (e.g., an irreducible subgraph of the control-flow graph is detected, see [1], which means that `goto` is needed), an explicit `goto` represents a necessary addition to our language. Even though our decompiler has partial access to type information, it is not yet utilized to its fullness for simplicity and readability reasons. However, type support is planned for back-ends producing typed HLLs, such as C.

After the generation is completed, an additional post-processing phase is done to further improve the readability of the code. These modifications are done on the textual level, and include the elimination of redundant brackets and expressions introduced by the back-end and simplification of conditions (e.g., "`if not (a == b):`" is simplified into "`if a != b:`").

5 Experimental Results

To demonstrate the abilities and advantages of our solution, we now present a decompilation of a simple program for Sony PSP. The C source code for this program is given in Figure 5. It was compiled using `gcc` (version 4.3.2) from the PSP-SDK [14] with enabled optimizations (`-O2`).

First, we transform the executable from the PRX file format into our unified COFF file format. Then, the detections of a statically linked code and used compiler are utilized to eliminate boilerplate code (i.e. `startup` code and statically linked libraries `libc` and `psplibc`. In our example, the executable

```
#include <pspkernel.h>
#include "sum.h"
/* Initialization */
PSP_MODULE_INFO("template", 0, 1, 1);
PSP_MAIN_THREAD_ATTR(0x80004000);
int main(void) {
    volatile int a = 3;
    int b;
    for (b = 1; b < 100; b++)
        a = sum(a, b);
    return a;
}
```

Figure 5: The source code of the input program.

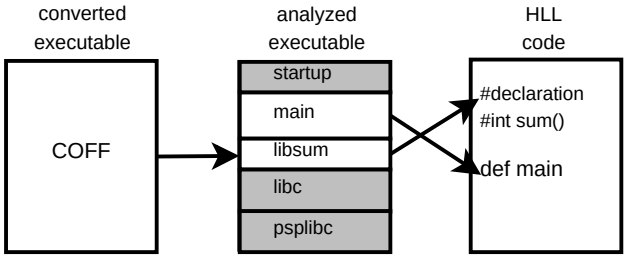

Figure 6: The decompilation process.

has 10 KB, while the stripped version of it has only 3 KB. After that, we disassembly the rest of the executable into LLVM IR using extracted semantics for the PSP description in the ISAC ADL. The disassembled code then undergoes several statical analyses, which transform platform-specific and hard-to-decompile constructs, such as function calls and argument passings, into more suitable forms. These analyses are followed by optimizations done in the middle-end. Finally, the back-end emits the HLL code, which is then optimized by the post-processor. The decompilation process is illustrated in Figure 6.

The resulting HLL code generated by our back-end can be seen in Figure 7. Observe the following key aspects of the resulting code. (1) Statically linked code (the `sum()` function in this example is from a static library) is removed. Startup code and other compiler-dependent parts of the code are also removed. (2) When known, we emit declarations of detected statically linked functions from header files to help analysts understand the code. (3) ABI description, specified in the ISAC model, is used to properly detect function calls, including argument passing and assignment of return values. (4) Variables are given more readable names. Clearly, `banana` is more readable than `gpregs_0x02`. Also, whenever possible, the variable containing the result of the current function is named `result`. (5) The `for` loop, including its induction variable, is successfully detected and transformed into a readable form. Furthermore, the induction variable is given a familiar name.

Finally, it should be noted, that decompilation of compiler-optimized applications can produce more

```
# ---------- Global Variables ----------
orange = 0
banana = 0
lemon = 0
# ----------- Declarations -------------
# int sum(int, int)
# -------- Defined Functions -----------
def main():
    global orange
    global banana
    global lemon
    orange = 3
    result = orange
    for i in range(0, 99):
        banana = result
        lemon = i + 1
        result = sum(banana, lemon)
    return result
```

Figure 7: The output from our decompiler.

readable code than decompilation of non-optimized code in some aspects. For example, compilers usually optimize frequent memory load and store of the same variable into its storage in register. This is much better for the decompiler, since it is easier to track the value of this variable in data-flow analyses.

6 Conclusion

This paper further extended the basic concept [5] and presented the implementation of our automatically generated retargetable decompiler of platform-specific machine code. The proof of the concept is presented on the Sony PSP handheld. Currently, we are able to decompile applications for this platform into a highly-readable HLL code.

However, there is still a lot of space for improvements. The output of the front-end can be further enhanced by the automatic detection of function boundaries and used ABI. With such information, it will be possible to automatically remove boilerplate code of functions and their callings.

As for the back-end, emission of some HLL constructs can be improved. For example, loops with no induction variable are currently emitted as `while True` loops. Furthermore, other HLLs can be considered, possibly including type information and conversions between types. Also, more complex dataflow analyses are required, since, for example, the `banana` variable in Figure 7 is not needed. This is planned to be improved in the future.

Due to space requirements, we were unable to present a more comprehensive example. Our solution properly decompiles most of commonly occurring HLL constructs. As mentioned earlier in this section, the main space for improvements lies in the detection and decompilation of functions.

Acknowledgements: This work was supported by the research funding TAČR, No. TA01010667, by the BUT FIT grant FIT-S-11-2, by the Research Plan MSM 0021630528, and by 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] Boomerang, `http://boomerang.sourceforge.net/`.

[3] Reverse Engineering Compiler, `http://www.backerstreet.com/rec/`.

[4] Hex-Rays Decompiler, `http://www.hex-rays.com/decompiler.shtml`.

[5] 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," in *5th International Conference on Information Security and Assurance*, Brno, CZ, 2011, pp. 72–86.

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

[7] The LLVM Compiler System, `http://llvm.org/`.

[8] A. Rahimzadeh, *Hacking the PSP™: Cool Hacks, Mods, and Customizations for the Sony® PlayStation® Portable*. New York: Wiley, 2006.

[9] N. Ramsey and M. F. Fernandez, "The new jersey machine-code toolkit," in *In Proceedings of the 1995 USENIX Technical Conference*, 1995, pp. 289–302.

[10] N. Ramsey and M. Fernández, "Specifying representations of machine instructions," *ACM Transactions on Programming Languages and Systems*, vol. 19, pp. 492–524, May 1997.

[11] Lissom Project, `http://www.fit.vutbr.cz/research/groups/lissom/`.

[12] A. Husár, M. Trmač, J. Hranáč, T. Hruška, K. Masařík, D. Kolář, and Z. Přikryl, "Automatic C compiler generation from architecture description language ISAC," in *MEMICS 2010*. Brno, CZ: Masaryk University, 2010, pp. 84–91.

[13] MIPS Technologies Inc., *MIPS32 Architecture for Programmers Volume II: The MIPS32 Instruction Set*, March 2010.

[14] Minimalist PSPSDK, `http://sourceforge.net/projects/minpspw/`.

[15] GNU Binutils, `http://www.gnu.org/software/binutils/`.

[16] Fast Library Identification and Recognition Technology, `http://www.hex-rays.com/idapro/flirt.htm`.

[17] Clang, `http://clang.llvm.org/`.