Advanced Static Analysis for Decompilation Using Scattered Context Grammars

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: Reverse program compilation (i.e. decompilation) is a process heavily exploited in reverse engineering. The task of decompilation is to transform a platform-specific executable into a high-level language representation, which is usually the C language. Such a process can be used for source code reconstruction, compiler testing, malware analysis, etc. In present, there are several existing decompilers that are able to decompile simple applications. However, we can see a drop-off in terms of the quality of the generated code when the decompiled code is highly optimized (e.g. usage of instruction idioms) or obfuscated (e.g. dead code insertion, register renaming). Optimized or obfuscated applications are usually generated by highly optimizing compilers or metamorphic engines (used by malware authors). In this paper, we present several innovative decompilation methods based on scattered context grammars. These methods are able to effectively decompile optimized or obfuscated code. For demonstration, we used these methods for enhancement of the static analysis phase of an existing decompiler. Experimental results of our solution are presented at the end of the paper.

Key–Words: decompilation, Lissom, static analysis, LLVM IR, scattered context grammars

1 Introduction

Machine code decompilation into a high-level language (HLL) code is the most traditional type of program decompilation. We can find several existing decompilers that have been created during the 50 years history of this reverse-engineering discipline. In present, the most advanced decompilers are *dcc* [1], *Boomerang* [2], *REC Decompiler* [3], and *Hex-Rays Decompiler* [4]. They are more or less bounded to a particular target processor architecture, mainly to Intel x86 [5]. However, some of them also support additional architectures, like ARM [6] or MIPS [7].

These decompilers are able to decompile compiler-generated applications and they can produce well-readable C code. However, we can find several code optimizing and obfuscating techniques (e.g. instruction substitution, dead code insertion), used by advanced compilers or malware generators. Such non-standard code is hard to decompile for existing decompilers, mainly because of lack of appropriate static code analyses.

In this paper, we present new decompilation techniques, that are used for non-standard code detection and its transformation into a more suitable form of representation. These techniques are based on a formal model—scattered context grammars, and they are platform independent. Therefore, they can be used within an existing retargetable decompiler, which is being developed in the Lissom project [8].

As will be seen in experimental results, usage of these techniques has a very good impact on the readability of the generated HLL code.

The paper is organized as follows. Section 2 introduces a retargetable decompiler developed within the Lissom project. Then, Section 3 describes scattered context grammars. A description of several techniques used in code optimization and obfuscation are presented in Section 4. This section also contains a comparison of existing decompilation methods and design of our own ones used in the static analysis phase. Experimental results are given in Section 5. Section 6 closes the paper by discussing future research.

2 Lissom Project Retargetable Decompiler

In this section, we will briefly describe the concept of an automatically generated retargetable decom-

ISBN: 978-1-61804-051-0

piler, which is developed in the Lissom project [8]. Our solution exploits architecture description language *ISAC* [9] for describing target architectures, and it is build on the top of the *LLVM Compiler System* [10]. The LLVM assembly language, called *LLVM IR*, is used as an internal representation of decompiled applications in particular decompilation phases. See [11] for a detailed description.

The decompiler's core consists of three basic parts—a *front-end*, a *middle-end*, and a *back-end*, see Figure 1.

Howbeit, at first, a *binary file converter* is used for transforming decompiled applications from a platform-specific executable format into an internal, uniform COFF-based file format. Currently, we support conversions from Windows PE, Unix ELF, Symbian E32, Apple Mach-O, and Android DEX file formats.

The decompiled application is further processed by the front-end, which is the only platform-specific part of the decompiler, because it is automatically generated based on the target architecture model. The architecture model is realized in the architecture description language ISAC [9], which is also developed in this project.

Each ISAC processor model describes processor resources (e.g. registers, memory hierarchy), and its instruction set (i.e. assembler language syntax, binary encoding, and semantics of each instruction). The model is transformed by a *semantics extractor* [12], which transforms semantics description of each instruction into a sequence of LLVM IR-like instructions, which properly describes its behavior. The extracted semantics is used for translation of the decompiled application into an LLVM IR instruction sequence, which characterize its behavior in a platform-independent way.

This intermediate program representation is further analysed and transformed in the static analysis phase of the front-end. This part is responsible for detection of functions, detection of application binary interface (ABI), elimination of statically linked code, etc.

Afterwards, the LLVM IR program representation is optimized in the middle-end using many built-in optimizations available in LLVM and our own passes (e.g. optimizations of loops, constant propagation, control-flow graph simplifications).

Finally, the program intermediate representation is emitted as the target HLL. Currently, a Python-like language is used for this purpose. However, a support for different back-ends (e.g. a C back-end) is planned.

In present, we have a complete ISAC model of the MIPS architecture. ISAC models of the ARM and Intel x86 architectures are in development. Therefore, our decompiler is able to decompile just MIPS applications yet.

In comparison with existing decompilers, our solution is not bound to any particular architecture. It should be noted that in present, there is no other competitive method of automatically-generated retargetable decompiler.

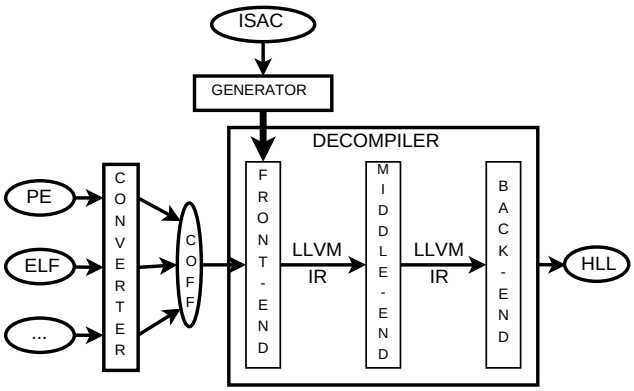

Figure 1: The concept of our retargetable decompiler.

3 Scattered Context Grammars

In this section, we assume that the reader is familiar with the basics of formal language theory (see [13]).

A *scattered context grammar* (an SCG for short, see [14]) represents a semi-parallel extension of a context-free grammar used to describe processing of information containing related elements, intuitively referred to as *scattered information*. As it is usual in formal language theory, an SCG comprises of a finite sets of *nonterminals* and *terminals*, a *starting nonterminal*, and a finite set of *scattered context rules* (abbreviated *rules*). Every scattered context rule is formed by a tuple of ordinary context-free rules, i.e. every rule is of the form

$$(A_1, A_2, \ldots, A_n) \rightarrow (x_1, x_2, \ldots, x_n)$$

where every A_i is a nonterminal and every x_i is a string of terminals and nonterminals. Such a rule can rewrite a string u if u is of the form $u = u_0A_1u_1A_2u_2 \cdots A_nu_n$, where A_1 through A_n represent scattered information and strings u_0 through u_n represent parts of u irrelevant to the current computational step. Such a computational step, referred to as a *derivation step*, has the form

$$u_0A_1u_1A_2u_2 \cdots A_nu_n \Rightarrow u_0x_1u_1x_2u_2 \cdots x_nu_n$$

where x_1 through x_n represent the new piece of information derived from A_1 through A_n. The *language*

ISBN: 978-1-61804-051-0

generated by an SCG consists of all strings of terminals which can be derived in this way from the starting nonterminal.

As an example, consider the non-context-free language $L = \{a^k b^k c^k \mid k \geq 1\}$. This language can be generated by an SCG with the following three rules:

$$\begin{array}{lcl} (S) & \rightarrow & (ABC) \\ (A,B,C) & \rightarrow & (aA,bB,cC) \\ (A,B,C) & \rightarrow & (a,b,c) \end{array}$$

(S, A, B, and C are nonterminals, S is the starting nonterminal.) For example, the string $aabbcc$ is generated as follows:

$$S \Rightarrow ABC \Rightarrow aAbBcC \Rightarrow aabbcc$$

When empty strings are permitted to appear on the right-hand side of a rule, this formalism is computationally complete, meaning that any "reasonable" computational process can be defined using an SCG. These grammars are applicable in many different areas, ranging from compilers through computational linguistics up to bioinformatics, see [14, 15]. An *LL SCG* (see [15]) represents a variant of an SCG suitable for efficient parsing.

4 Advanced Static Analysis

Static analysis can enhance decompilation results. It can also provide some additional information about the processed code. There are several issues that are appropriate to be solved by such an analysis. Our analyses are aimed to codes which are widely used by metamorphic engines or virus generators. In the following subsections, we will introduce various kinds of unwanted phenomena and solutions for their suppression.

4.1 Unwanted Phenomena in Binary Code

We took three types of classic code obfuscations which change code while preserving its semantics. Some of them, like instructions substitution, are also used by compilers for optimization purposes.

- *instructions substitution* – A lot of operations can be performed by various instructions or groups of instructions. Such instructions are called *idioms*. The simplest example is assigning zero to a register, which can be made by `MOV reg, 0`, but also by `XOR reg, reg` or `SUB reg, reg`.

- *register renaming* – A majority of instructions is able to work with more than a single register. During obfuscation, registers can be exchanged in some piece of code. This can break ABI for the given architecture, which causes complications for the decompilation process. We can illustrate it on code for the MIPS processor architecture [7], where the first argument is stored in the register `a0`, the calling code and code of functions can be adjusted to use e.g. the temporary register `t0` for this purpose [16]. This represents an obvious problem for a decompiler which relies on standard ABI conventions and assumes that the first argument is stored in `a0`.

- *dead code insertion* – Dead code is a term for code which can be removed without any effects on the behavior of the original code. Its usual aim is to hide another potentially dangerous code. A challenge for a decompiler is a detection of such code and its elimination from the output, which has very beneficial impact on the readability of the resulting code.

4.2 State of the Art

In 2003, M. Christodorescu and S. Jha introduced SAFE—a static analyzer for executables [17]. It uses static analyses of several obfuscation techniques to recognize malware. They constructed a detector, which takes annotated call flow graph of an executable and a malicious code automaton, and it is able to find the pattern described by this automaton in this executable. This algorithm is context-insensitive because of using a finite state automaton.

We should also take a look on how static analyses are used in other decompilers. We made experiments with two decompilers. We chose Boomerang and REC Decompiler, because they are available for free. Two other decompilers were not included in the test because dcc can decompile code only for MS-DOS and Hex-Rays Decompiler is a commercial product. We created a simple code which uses the idiom with `XOR` for zeroing a register, and useless addition and subtraction. The tested code in the C language is shown in Figure 2. It uses a static library named `libsum` with a single function `sum()`. An advantage of this approach is a possibility to create and use signatures for detecting statically linked code. We transformed this C code into an assembly code and obfuscated it. A snippet of the obfuscated code in the MIPS assembler is shown in Figure 3. Division by 5 was replaced by multiplication by 3435973837. This idiom uses multiplication by a large number to emulate division, see [18]. The idiom works only for a subset of numbers, so it has to be used cleverly. It utilizes the fact that the result of a multiplication on a 32b architecture is truncated to 32 bits. The second

ISBN: 978-1-61804-051-0

```
#include "sum.h"

int main(int argc, char *argv[]) {
        int a = argc / 5;
        int b = sum (a, a ^ a) + 1;
        return b - 1;
}
```

Figure 2: The original code of our example.

```
        lui     $3,52428
        addiu   $3,52429
        mult    $2,$3
        mflo    $2
        sw      $2,28($fp)
        lw      $7,28($fp)
        xor     $8,$2,$2
        jal     sum
```

Figure 3: The obfuscated code of the example.

obfuscation was based on switching registers which are used for passing arguments to functions. This results into non-standard ABI. The significance of this test was monitoring how decompilers manage to handle idioms, dead code, and broken ABI.

The first tested decompiler was Boomerang. This decompiler unfortunately does not support MIPS, so we rewrote the example to x86 assembler. As can be seen from the result in Figure 4, this decompiler does not run analyses for removing dead code and idioms (the `XOR` instruction was not replaced by zero). From the return statement, it is clear that expressions are not partially evaluated, because otherwise, the expression `+ 1 - 1` would be removed. It does not have any trouble with broken ABI, because it does not try to recognize which values are arguments of a function (there are no arguments in the `sum()` call). Also, we see that the advanced idiom that replaced `DIV` by `MULT` is not handled in any special way, except displaying the number in a hexadecimal base format.

The REC decompiler was tested on the MIPS code because it supports that architecture. From its result in Figure 5, we see that it did not decode the instruction `MULT`, so some analysis for used idiom was not possible. In contrast to Boomerang, it replaced `XOR` by zero. A good point is an analysis for recognizing function call arguments, since we used non-standard registers `a3` and `t0` for passing arguments. The created function declares that it takes four arguments, but the first one is a memory value from the address `(_t25 + 28)`, which is actually `argc`, and the second one is zero. Partial evaluation of expressions is also not run because adding and subtracting of 1 remained in the code.

```
int main(int argc, char *argv[]) {
  unsigned int eax; // r24
  void *esp;        // r28

  eax = *((esp - 4) + 8);
  tmp = (eax) * ((unsigned int)0xcccccccd);
  *(unsigned long*)(esp - 4 - 32 + 24) = tmp;
  *(__size32*)(esp - 4 - 32 + 4) = tmp ^ tmp;
  eax = *(esp - 4 - 32 + 24);
  *(__size32*)(esp - 4 - 32) = eax;
  eax = sum();
  return eax + 1 - 1;
}

__size32 sum() {
  unsigned int eax; // r24
  __size32 edx;     // r26
  __size32 esp;     // r28

  eax = *((esp - 4) + 12);
  edx = *(esp - 4 + 8);
  return edx + eax * 1;
}
```

Figure 4: The result from Boomerang.

4.3 Design of Our Analyses

A retargetable decompiler should use analyses which are generic for more platforms. Therefore, analyses have to be created on the level of the internal IR code. Such analyses can be used without regard to the origin platform of the processed code.

In the first analysis, we focused on idioms. We created a set of IR codes and rules for simplifying these codes. The simpler part of this analysis replaces more obvious constructions, such as `XOR reg, reg` or `SUB reg, reg` by `MOV reg, 0`. Also, shifts to the left and to the right by n bits are changed to a multiplication or division by 2^n.

A little bit more advanced analysis is aimed on using multiplication instead of division. We created a table for this idiom. The table contains two types of values: a multiplier and a divisor. The divisors are numbers less than 20. This idiom is also used for optimization by compilers, because division is usually much more slower than multiplication. Our analysis exchanges multiplication by multipliers from the table to division by the corresponding divisor, what is more readable. This analysis also serves as code normalization, what has positive effect on the next analysis.

The second, advanced, analysis tries to fix broken ABI. Our decompiler supports ABI conventions as recognizing and generating function arguments, but at this time, ABI has to be standard. The main idea is to recognize switched registers and fix them back. This analysis is based on the concept of recognizing

```
main(_unknown_ r2, intOrPtr r30) {
    _unknown_ _t25;

    _t25 = __sp;
     *((intOrPtr*)(_t25 + 40)) = __a0;
     *((intOrPtr*)(_t25 + 44)) = __a1;
    asm("mult␣v0,v1");
     *((intOrPtr*)(_t25 + 28)) =
*((intOrPtr*)(_t25 + 40));
    __a3 =  *((intOrPtr*)(_t25 + 28));
    __t0 = 0;
    _t18 = L400790( *((intOrPtr*)
           (_t25 + 28)), 0, __sp, _t25);
     *((intOrPtr*)(_t25 + 24)) = _t18 + 1;
    return  *((intOrPtr*)(_t25 + 24)) + -1;
}

L400790(intOrPtr r7, intOrPtr r8,
        _unknown_ r29, intOrPtr r30) {
    _unknown_ _t12;
    __sp = __sp;
    __t0 = __t0;
    __a3 = __a3;
    _t12 = __sp;
     *((intOrPtr*)(_t12 + 8)) = __a3;
     *((intOrPtr*)(_t12 + 12)) = __t0;
    __sp = _t12;
    return  *((intOrPtr*)(_t12 + 8)) +
            *((intOrPtr*)(_t12 + 12));
}
```

Figure 5: The result from REC decompiler.

register realignment by LL SCGs [19]. The presented solution was designed for an x86 assembler, which is insufficient for us. Therefore, we extended the algorithm from the paper to work with the LLVM IR code. We are able to describe pieces of code by an SCG. The rules of the used grammar exactly determine possible combinations of used registers in this code while preserving code semantics. We built a database with grammars for commonly occurred pieces of codes, used by metamorphic engines, virus generators, and similar applications. Such codes use to be adjusted by register realignment. SCGs can detect such code and we can fix registers to follow standard ABI. The fixation is also done using SCGs, but this part is not finished yet. In Figure 6, a subset of used rules is shown (Note that simplified LLVM IR syntax was used for lucidity). The grammar works with LLVM IR code, which was preprocessed by the first presented analysis. Therefore, there is `LOAD` of zero to a register instead of `XOR`.

$S \rightarrow$ `load` X_1`,5`
`op_arith_div` X_2, X_3, X_4
`store` X_5`,28(r29)`
`load` X_6`,28(r29)`
`load` X_7`,0`
`call sum`

$(X_1, X_2, X_3, X_4, X_5, X_6, X_7) \rightarrow$
(r3,r2,r2,r3,r2,r7,r8)

$(X_1, X_2, X_3, X_4, X_5, X_6, X_7) \rightarrow$
(r4,r3,r3,r4,r3,r5,r6)

Figure 6: A snippet of an LL SCG for the presented obfuscated example.

5 Experimental Results

In this section, we show how our decompiler, using the analyses introduced in the previous section, manages to decompile the example from Figure 2. Analyses are implemented in the static analysis phase of the decompiler's front-end, see Figure 1.

The output from our decompiler can be seen in Figure 7. Notice the following key aspects of the decompiled code. (1) Dead code, such as adding and subtracting 1 from the result, was eliminated. Also, `XOR`ing a variable with itself was replaced with 0. (2) The division-by-multiplication idiom was correctly detected. So, instead of multiplying `argc` by 3435973837, it is divided by 5, which is more readable. (3) Our analysis detected register renaming and fixed it, so ABI detection was successfully used. Hence, we were able to correctly detect how many and which arguments should be passed to the `sum()` call. (4) Since `sum()` is from a statically linked library, our decompiler removed its definition from the resulting output and just emitted its declaration. Compare all of these points with Figures 4 and 5.

We measured the impact of these analyses on the execution time of the decompiler. The analyses increased the running time by approximately 10% according to used input. For the presented example, the overall time is still less than 1 second. We consider such increase as acceptable.

6 Conclusion

We presented some advanced techniques for static analysis usable in decompilation, namely (1) detection and replacement of idioms, (2) detection and removal of inserted dead code, and (3) detection and

```
# ------------ Functions ---------------
def main(argc, **argv):
    result = argc / 5
    result = sum(result, 0)
    return result

# ------- Function Declarations --------
#  int sum(int, int)

# ------------ Entry Point --------------
if __name__ == '__main__':
    import sys
    sys.exit(main(len(sys.argv), sys.argv))
```

Figure 7: The result from our decompiler.

correction of register renaming using SCGs. We created an example requiring each of the mentioned analyses, and we tested two currently existing decompilers on this example. Since none of them was able to fully tackle the presented obfuscations and optimizations, we provided a design of new analyses. As a proof of concept, we added an implementation of the designed analyses into our decompiler, and showed that it successfully managed to cope with the presented problems. The introduction of the new analyses resulted in a more precise and readable code.

We close this paper by suggesting several topics for future research. Many new analyses concerning instruction substitution can be designed and implemented (see, for example, the idioms given in [18]). To decrease the number of rules needed in an SCG to handle register renaming, one may consider to use *SCGs with priority* (see [20]). Also, an automatic generator of SCG databases using SCGs would obviously decrease the amount of time needed to implement a suitable SCG.

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/rec.htm.

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

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

[6] ARM Limited, *ARM Architecture Reference Manual*. ARM DDI 0100I, 2005.

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

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

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

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

[11] 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.

[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 *6th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*. Brno, CZ: Masaryk University, 2010, pp. 84–91.

[13] A. Meduna, *Automata and Languages: Theory and Applications*. Springer, London, 2000.

[14] A. Meduna and J. Techet, *Scattered Context Grammars and their Applications*. WIT Press, Southampton, 2010.

[15] D. Kolář, "Scattered context grammars parsers," in *Proceedings of the 14th International Congress of Cybernetics and Systems of WOCS*. Wroclaw, PL: Wroclaw University of Technology, 2008, pp. 491–500.

[16] D. Patterson and J. L. Hennessy, *Computer organization and design: the hardware/software interface*. Elsevier/Morgan Kaufmann, 2005.

[17] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. Berkeley, CA, USA: USENIX Association, 2003, pp. 169–186.

[18] H. Warren, *Hacker's delight*. Boston: Addison-Wesley, 2003.

[19] L. Ďurfina and D. Kolář, "Generic detection of register realignment," in *SCLIT'11: Proceedings of the Symposium on Computer Languages, Implementations and Tools*, Kassandra, Greece, 2011.

[20] J. Křoustek, S. Židek, D. Kolář, and A. Meduna, "Scattered context grammars with priority," *International Journal of Advanced Research in Computer Science*, vol. 2, no. 4, pp. 1–6, 2011.