MO601 - Project 3 Report
Learning Instruction Semantics From Code Generators

093311 - Alberto Arruda de Oliveira[1]

[1]Institute of Computing – University of Campinas (UNICAMP)

{alberto.oliveira}@ic.unicamp.br

Abstract. *Instruction lifting, the process of translating machine instructions into a higher form of intermediate language is crucial for many systems focused on binary analysis and code instrumentation. Hasabnis and Sekar [Hasabnis and Sekar 2016] proposed the LISC (Lifting Instruction Semantics from Code Generators) tool to automatically perform instruction lifting, using information from code generators employed by compilers. The purpose of code generators is to translate an intermediate language (IL), such as GCC's RTL, into machine instructions. Thus, the authors argue that by employing machine learning techniques using the output of code generators as training data, it is possible to build a transducer that performs the inverse, translating instructions into IL. Currently, for most systems this translation is done manually, a process which is very time consuming given the large number of instructions present in different architectures, leading to limited support. By eliminating or reducing the need for manual translation, it is possible to quickly adapt systems that require it for new architectures. In this project, e tested and tried to reproduce the results of LISC for the x86 platform, in the completeness test scenario.*

1. Introduction

Program monitoring and debugging systems, such as Pin [Pin], virtualization systems, as well as software security techniques, all rely on binary analysis and instrumentation. The modeling of instruction semantics into a higher level intermediate language is one of the main steps required by tools and algorithms that rely on binary analysis. Currently, such modeling is mostly done manually. However, considering the large amount of instruction present on modern architectures (x86, ARM v7), which surpasses the thousands, as well as the complexity of such instruction sets, makes the manual translation a hard and very time consuming process. This, in turn, leads to limited architecture support by popular programs that rely on such models.

Hasabnis and Sekar [Hasabnis and Sekar 2016] propose a system to automatically build the semantic models, based on knowledge that can be obtained from modern compilers such as GCC. Since compilers employ code generators to convert an *Intermediate Language*(IL) into binary instructions, the authors argue that it is reasonable that by learning from such those generators, it is possible to perform the inverse. The main advantages of using this approach, besides its automated nature, is that the resulting code is *architecture neutral*, and the binary lifting relies on compiler code, which is very robust and well tested. Besides the development of the LISC tool, Hasabnis and Sekar [Hasabnis and Sekar 2016] also proposed a evaluation framework for their tool, which considers areas such as completeness and correctness.

The objective of this project was to reproduce the results obtained by table 4 of [Hasabnis and Sekar 2016], which displays the completeness results of the LISC tool. The objective of this type of evaluation is to assess how many of the instructions of a testing set of programs are lifted, given a set of training programs. However, as we will show in section 4, the reproduction of such experiments is not so simple. It requires very particular system configurations which cannot be easily duplicate, unless the original training and resting files are present.

2. Problem Overview

The main idea behind LISC is to employ machine learning as a way to given a translation from Intermediate Language (IL) to assembly produced by code generators, learn a mapping from assembly to intermediate language. However, the machine learning algorithm for this problem should be carefully chosen, as no translation errors can be tolerated. Plus, it should be an algorithm that supports a tree-like output, used for translation. Those two constraints lead the authors to the *Finite State Transducers* (FSTs).

Another important issue the authors tackle is the translation of assembly languages into intermediate language. Because there are multiple possible translations, lifting a binary to IL could result in an exponential problem. Another issue is the fact that multiple instructions, instead of a single one, have to be translated to a single IL snippet. To solve this problem, the authors propose a dynamic programming algorithms that assigns costs to larger groups of instructions that can be lifted by a single pass of the transducer.

The translation problem is formulated, according to the authors, as *learning a parameterized translation on trees*, each tree representing a IL snippet. The parameters correspond to operands in assembly and IL.

3. Methodology

In this section, the overall methodology used to reproduce the results shown in the paper is discussed, as well as the evaluation employed.

3.1. Execution Flow

Together with their paper, the authors made available a pack with the LISC tool [LIS], as well as other supporting tools. The general flow of execution to train an automata and lift the instructions of a target binary is displayed by Figure 1.

To extract the logs of GCC's code generator (.S), it is necessary to run GCC using a special plugin contained in the packages. This is done by including the following flags:

-fplugin=path_to_lisc/pprtl/plugin_dump_mapping.so
-fplugin-arg-plugin_dump_mapping-out_file=output_path/output_name.S -dP

and compiling afterwards. Such flags can be included after the variable **CC** in a program's Makefile. Although this process is straightforward for some programs, for others (those that employ tools specifically for building the project, like QMake), can be very difficult and result in unexpected errors, like failures to load plugins or file descriptor

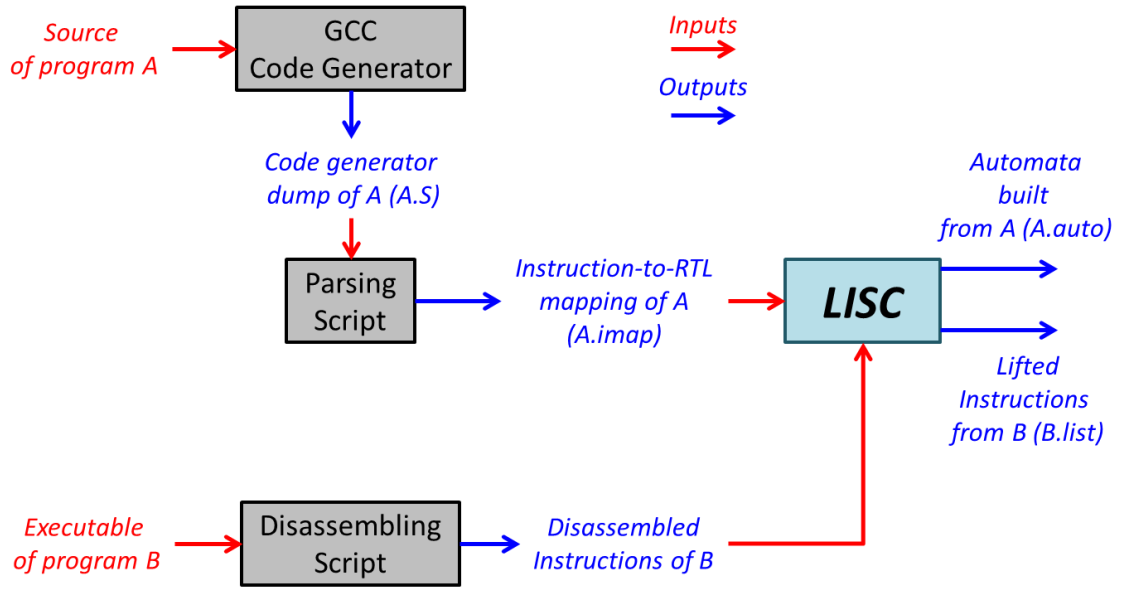

Figure 1. Flow of the LISC tool.

issues. The code logs for two of the programs listed as used to produce the completeness table in [Hasabnis and Sekar 2016], the linux kernel and QT, could not be produced correctly, thus they were skipped. Although the instructions were followed exactly as pointed in LISC's, it is very difficult to be completely sure if the files were correctly produced, as any changes in configuration could lead to a different dump being generated. There are several other important information that the authors skip as well, such as if only .c code dumps are generated or .cpp are as well.

Once the code generator dumps are obtained, they have to be parsed to a simpler formatting, with extension .imap. As the name suggest, this format maps each instruction to its corresponding RTL. The file is organized in pairs of line, of the following format:

movl 8(%esp), %edx ;
(set (reg:SI edx) (mem:SI (plus:SI (reg:SI esp) (const_int 8))))

where the first line is the instruction, and the second is the corresponding RTL, as per section 2. To produce a single .imap files for multiple source programs, their individual .imap files are simply concatenated. A script named *combined_imap.py* was developed to produced such combined .imap files.

The disassembling script uses *objdump* to produce a text file containing the sequence of instructions of an executable. If multiple executables are to be used, then the files containing their sequence of instructions are concatenated. A script named *run_disass.sh* was developed to run the disassembling script for multiple programs and concatenate their results.

Once the .imap file is obtained, we train and save the automata by running LISC. The command to run and and save the automata, saving a .dot file of the automata figure is:

Table 1. Original Results for LISC Completeness

P_{train}	% Instructions Lifted		LISC (%)		Missing Mnemonics (Absolute)
	Exact Recall	LISC	Missing Mnemonics	Missing Operands	
openssl-1.0.1f + binutils-2.22	63.72	98.46	1.05	0.49	464
+ffmpeg-2.3.3 (Non opt)	68.21	98.74	1.03	0.23	377
+glibc-2.21	68.74	98.80	1.01	0.19	346
+ffmpeg-2.3.3 (Opt)	69.07	98.89	0.88	0.23	303
+gstreamer-1.4.5	71.07	99.10	0.79	0.11	221
+qt-5.4.1	72.45	99.21	0.69	0.09	161
+linuxkern-3.19	73.97	99.49	0.44	0.07	49
+Manual	74.04	100.00	0.00	0.00	0

./learnopt -tr input.imap -sa output.automata -dotf output.dot

3.2. Evaluation

The evaluation proposed by Hasabnis and Sekar [Hasabnis and Sekar 2016] consists of four areas: completeness, architecture neutrality, performance, and correctness. Completeness was the focus of this project, with the attempt to reproduce Table 4 of [Hasabnis and Sekar 2016]. The completeness experiment consists in using a training set of programs, or P_{train} to create automatas, and then check its capability to lift the instructions of a testing set of programs, or P_{test}. In this scenario, P_{train} is always a subset of P_{test}, and is expanded by adding a new program in each execution round (or line in table 4 of [Hasabnis and Sekar 2016]).

First, the authors compare the % of instructions lifted by LISC with the *Exact Recall* (an instruction of P_{test} is lifted only if it also belongs to P_{train}. Then, they list the % of missing instructions, divided in mnemonics and operands, of the lifting performed by LISC.

4. Experiments

Although our objective was to reproduce Table 4 of [Hasabnis and Sekar 2016] (Displayed as Table 1 here), it was not possible to do so, as accurate information about the complete procedure was not provided, even after managing to get in contact with the original authors of the paper. The authors managed to provide some of the .imap files used, but they were obsolete files, so they could not be used to generate the required automatas. In addition, the paper does not make it clear how the P_{test} set was generated, an alternate methodology was used instead. Finally, the .imap files for the linux kernel and QT could not be correctly generated, even though the exact same steps provided by the authors were followed. It is important to keep in mind that reproducing the results without the original files is extremely difficult, as any changes in configuration of source files used or the machine employed can lead to changes in the set of train and testing instructions.

Instead, a new experiment was performed, trying to follow as best as possible the framework provided by [Hasabnis and Sekar 2016]. The experiment consisted in generating imaps for *openssl-1.0.0f*, *binutils-2.24*, *ffmpeg-2.3.3*(without and with some optmization flags), *glibc-2.19*, and *gstreamer-1.2.4*, to create P_{train}, and use as script to disassemble all executable files available in a virtualized 32-bit Ubuntu 14.04 (after installing the

aforementioned libraries) to create P_{test}, which resulted in a P_{test} with around 4 million unique instructions. The exact recall could not be computed, since the script provided by the authors lacked a dependency and did not work properly. Instead, we display only the percentage of lifted instructions by LISC and the percentage of missing mnemonics and operands. Table **??** display such results.

5. Conclusion

In this project, the LISC [Hasabnis and Sekar 2016] was studied and tested. The objective of such system is the to automate the process of translating assembly instructions into a intermediate representation, as process required by many binary analysis and instrumentation tools. This process is usually done manually, requiring a great deal of labor and limiting the reach of tools that need it. A machine learning approach based on Finite State Transducers is employed to learn the semantics involved in the translation of Intermediate Language to assembly done by a Compiler's code generator, to perform the inverse translation.

Our objective was to reproduce the completeness experiment proposed by the authors, which is displayed in Table 4 of [Hasabnis and Sekar 2016]. However, we argue that it is very difficult to perfectly reproduce such results without having access to the original files used. This is due to multiple factors, such as the difficulty in completely duplicating the environment used to generate the training and testing files, as well as failure in a precise description of the methodology used to dump the code generator logs of some programs employed. Instead, we opted to perform a new experiment based in the experimental setup proposed by the authors, in a smaller scope.

References

Lisc - learning instruction semantics from code generators. `http://seclab.cs.sunysb.edu/seclab/lisc/`. Accessed: 2016-11-17.

Pin - a dynamic binary instrumentation tool. `https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool`. Accessed: 2016-09-15.

Hasabnis, N. and Sekar, R. (2016). Lifting assembly to intermediate representation: A novel approach leveraging compilers.