DEL: Dynamic Symbolic Execution-based Lifter for Enhanced Low-Level Intermediate Representation

Hany Abdelmaksoud†,Zain A. H. Hammadeh†, Goerschwin Fey‡, Daniel Lüdtke†
† German Aerospace Center (DLR), Braunschweig, Germany ‡ TU Hamburg, Hamburg, Germany
Email: hany.abdelmaksoud@dlr.de, zain.hajhammadeh@dlr.de, goerschwin.fey@tuhh.de, daniel.luedtke@dlr.de

***Abstract*—This work develops an approach that lifts binaries into an enhanced LLVM Intermediate Representation (IR) including indirect jumps. The proposed lifter combines both static and dynamic methods and strives to fully recover the Control-Flow Graph (CFG) of a program. Using Satisfiability Modulo Theories (SMT) supported by memory and register models, our lifter dynamically symbolically executes IR instructions after translating them into SMT expressions.**

I. Introduction

Many tools has been developed to analyze the Intermediate Representation (IR) of a program and not the source code to make the analysis independent of the programming language and to make analyses possible if only the binary is available. One type of analysis tools lifts a binary to an IR as an intermediate step, e.g., S2E [1]. However, some tools are dedicated to lifting binaries to different IRs and are known as binary lifters, e.g., McSema[1]. Our work presents a lifter that outputs an IR that can be useful for applying different types of analyses, especially timing analyses [2].

Lifting approaches are classified as either static or dynamic. Indirect jumps present a huge challenge for static lifters when it comes to fully recovering the Control-Flow Graph (CFG). On the other hand, for dynamic lifters, full code coverage becomes an issue as they lift only the executable code for a given input set. Listing 1 shows a C++ program that follows the inversion control programming paradigm used by software frameworks to develop embedded software. The class `Task` has a pure virtual method, namely `execute` in Line 5. Static lifting approaches used by, e.g., RetDec[2] and Angr [3] cannot resolve the indirect jump caused by the `execute` function call in Line 12.

In this work, we develop a lifter that combines both static lifting and Dynamic Symbolic Execution (DSE) to generate enhanced IR modules. Hence, we call our lifter DEL (Dynamic symbolic Execution Lifter). Our lifter takes a binary file and lifts it into an LLVM IR module. Figure 1 shows the detailed structure of our lifter. DEL guarantees full coverage of the code under analysis, unlike purely dynamic lifters. Also, DEL resolves indirect jumps in the program guaranteeing a full control flow recovery. Leveraging the DSE engine and the static lifting, DEL can resolve the indirect jump presented in Listing 1 with 100% code coverage.

[1]https://github.com/lifting-bits/mcsema
[2]https://github.com/avast/retdec

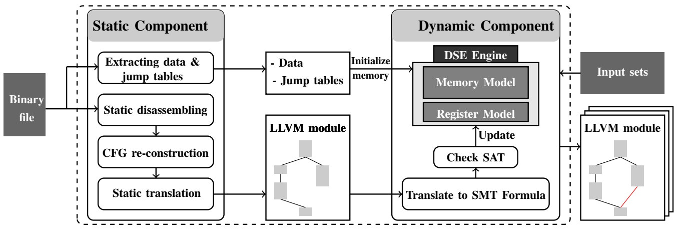

Fig. 1: The structure of DEL

Listing 1: C++ example of indirect jumps

```
\\ framework.hpp                                              1
class Task{                                                   2
        public: explicit Task( int wcet ): wcet_{wcet}{}      3
        virtual ~Task();                                      4
        virtual void execute(void) = 0;                       5
        private: int wcet_{};};                               6
class Scheduler {                                             7
        public: void perform(Task& task);};                   8
                                                              9
\\ framework.cpp                                              10
#include "framework.hpp"                                      11
void Scheduler::perform(Task& task) {task.execute();}         12
Task::~Task() {}                                              13
                                                              14
\\ App.cpp                                                    15
#include "framework.hpp"                                      16
class T1: public Task{                                        17
        public: T1(int wcet) : Task(wcet) {};                 18
        void execute(void) override;};                        19
class T2: public Task{                                        20
        public: T2(int wcet) :Task(wcet) {};                  21
        void execute(void) override;};                        22
        void T1::execute(void){int x = 6; x++;}               23
        void T2::execute(void){int y = 8; y--;}               24
int main(int argc, char** argv){                              25
        T1 firstTask(9);                                      26
        T2 secondTask(4);                                     27
        Scheduler sched;                                      28
        if (argc < 2){sched.perform(firstTask);}              29
        else{sched.perform(secondTask);}                      30
        return 0;}                                            31
```

II. DEL: DSE-based Lifting

The lifting process proposed in this paper combines static and dynamic approaches. Specifically, this combination enables DEL to produce a more accurate CFG than previous approaches. DEL consists of a static component and a dynamic component. DEL's dynamic component takes as input the output provided by its static component. DEL generates one *basic* LLVM IR module and multiple *enhanced* LLVM IR modules. The basic LLVM module is generated from the static component. This module has a full coverage of the input code. However, it might have an incomplete CFG missing indirect jumps. The enhanced LLVM IR modules are generated using the DSE from the dynamic component. Each enhanced LLVM IR module resolves indirect jumps and covers only the executable code for a given (set of) input(s).

A. Static component

The static component carries out the following steps:

TABLE I: Example of translating assembly into LLVM.

ARMv7-M instruction	LLVM instructions
add r3, #4	%55 = load i32, i32* %R3, align 4 %56 = add i32 %55, 4 store i32 %56, i32* %R3, align 4

1) Static disassembling: We use the *objdump*[3] static disassembler to disassemble an input binary into ARM assembly.

2) Extracting data and jump tables: The static component extracts the data and the jump tables from the binaries. The extracted data and jump table information are used to initialize DEL's memory model.

3) CFG re-construction: The static component iterates through the input assembly code and identifies the basic blocks. Once basic blocks have been identified, DEL reconstructs a preliminary CFG illustrating the predecessor and successor relationships between the different basic blocks. Here, the indirect jumps (*bx* and *blx*) are detected but not resolved. We link the jump tables extracted from the previous step to the indirect jump instructions. Each indirect jump instruction now has a range of potential addresses that it can resolve to.

4) Static translation: DEL once more iterates through the assembly instructions and translates each assembly instruction into a set of LLVM IRs. For each assembly instruction in the ARMv7-M ISA, DEL implements a C++ API that translates it into its equivalent set of LLVM IR instructions. Each assembly instruction in the ARMv7-M ISA is translated into an average of three to five LLVM instructions. The flag checking and updating functionalities of a lifted assembly instruction are also broken down into their own set of LLVM instructions during lifting. For the example in Listing 1, the static component generates an IR module that comprises 734 instructions, 3 of which are indirect jumps.

Table I shows how to lift the **add** ARMv7-M instruction into LLVM instructions.

B. Dynamic component

Abaza et al. proposed in [2] a DSE engine using an SMT solver, namely Z3. They proposed establishing a memory and register model using the Z3 vector array, translating each IR instruction into a Z3 expression, checking the satisfiability of the generated Z3 expression, and updating the memory and register model accordingly. The dynamic component is inspired by the DSE engine presented in [2]. Hence, we have memory and register models, translation to SMT expressions, and dynamic symbolic execution. Also, we use Z3 as an SMT solver.

1) Memory and register models: We used Z3 bit vectors to build a memory model and a register model. The memory model (register model) is a map where the key is the memory address (register name) and the value is a Z3 bit vector. In the DSE, each LLVM instruction could either update the memory model or the register model.

[3]https://sourceware.org/binutils/docs/binutils/objdump

2) Translation to SMT expressions: Static Single Assignment (SSA) facilitates using Z3 for the DSE of IR instructions [2]. IR instructions are translated into formulas that imply the mathematical effect of the IR instruction on the engine state. Using the array and bit vector theories of Z3 enables direct mapping of the SSA form of the Low Level Intermediate Representation (LLIR) to a Z3 expression. Equation 1 shows the translation of an LLVM IR *add* instruction into a Z3 expression.

$$[\%r1 = add\ i32\ \%r0,\ 1] \Rightarrow \\ BitVec(r1, c) = BitVec(r0, c) + BitVec(1, c) \quad (1)$$

We translate each LLVM instruction that we encounter during the DSE into a Z3 expression.

3) Execution: The DSE is input based, which means different input sets to the program might result in different execution paths explored during the DSE. An input set is used in the DSE to discover the execution paths and resolve the encountered indirect jumps to concrete jump targets. The LLVM instructions are dynamically symbolically executed by evaluating the equivalent Z3 expressions by the Z3 solver and updating the memory or the register models. The state of the memory and register models after execution of the IR instruction are logged and attached to the instruction. The execution ends once DEL reaches the exit block of the program. At the end, an enhanced LLVM IR module is generated. The module has no non-resolved indirect jumps and covers only the executed paths for the considered input set. Ideally, using comprehensive input sets would result in all indirect jumps being resolved and all potential jump targets being visited during the DSE.

III. Conclusion

In this paper, we presented a new approach to lifting binaries into LLVM IR modules that aims to fully recover the CFG while providing a full code coverage using static lifting and DSE. The presented lifter seamlessly combines the static lifter with the DSE engine, unlike other tools that loosely chain components for disassembling, CFG recovery, and translation to IR with the need for many interfaces between the components.

References

[1] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 265–278.

[2] H. Abaza, Z. A. Haj Hammadeh, and D. Lüdtke, "DELOOP: Automatic flow facts computation using dynamic symbolic execution," in *20th International Workshop on Worst-Case Execution Time Analysis (WCET 2022)*, ser. Open Access Series in Informatics (OASIcs), C. Ballabriga, Ed., vol. 103. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, pp. 3:1–3:12.

[3] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel *et al.*, "Sok:(state of) the art of war: Offensive techniques in binary analysis," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 138–157.