SoK: Demystifying Binary Lifters Through the Lens of Downstream Applications Supplementary Materials

I. Patching & Augmenting Binary Lifters

This section discusses patches and extensions we made for each binary lifter. Overall, all static binary lifters evaluated in this research are developed and maintained by companies (e.g., Microsoft, Traits of Bits Inc.). They show very solid engineering quality. The dynamic lifter, `BinRec` (published at EuroSys '20 [2]), is also of good engineering quality. We now list a few changes and augmentations we made to each lifter to support our research.

Changes on `BinRec`. We made a number of changes and fixed some small issues in the current codebase of `BinRec`. We now list all the augmentation and changes we made as follows:

- *SPEC Test Inputs*: `BinRec` provides a set of inputs to execute SPEC test cases. We modify the `scripts/create_spec_bins.sh` script and select a proper set of inputs, such that the anticipated symbolic execution time is close to around 50 hours (changed 51 LOC).
- *Skip Failed Traces and Empty Folders*: we write a script to detect traces that cause IR lifting errors (in 189 LOC). We also modify the `scripts/merge_captured_dirs_multi.sh` script, skipping a directory in case it does not contain any successfully logged traces (changed 28 LOC).
- *Bug Fix*: we fix a bug at line 289 of `scripts/make_loadable_segments.py`. To generate a functional executable by compiling lifted IR code, `BinRec` leverages a "replica-based" instrumentation method [6] which stitches the original executable and the new executable compiled from the lifted IR code. The gaps between two ELF binary sections were somehow incorrectly calculated, although that should not affect the primary behavior of the final outputs.
- *POJ-104 Test Cases*: `BinRec` employs $\mathtt{S}^2\mathtt{E}$ [5] to perform symbolic execution and discover program paths. To launch experiments on POJ-104 programs, we write a script to set proper symbolic inputs for all 104 classes of POJ-104 programs. In short, we managed to properly prepare four kinds of symbolic inputs for different programs as follows:

1) For programs which take only one input (`int` or `char`), we create four byte symbolic inputs.
2) For programs take multiple inputs or a string (i.e., `char*`): we create eight byte symbolic inputs.
3) For test programs that repeatedly read inputs within a loop, we create 12 byte symbolic inputs. POJ-104 programs, denoting typical entry-level college programming assignments, frequently use such patterns to read user inputs.
4) For test programs that iteratively read inputs within a nested loop, we create 20 byte symbolic inputs.

We spent **considerable manual efforts** at this step to prepare proper symbolic inputs for the POJ-104 test cases. To the best of our knowledge, these four situations subsume all input types of POJ-104 programs.

Changes on `McSema`. Typically, in addition to user-defined functions, compilers will add a number of helper functions into the Linux ELF executable, e.g., preparing the memory stack before entering `main`. `McSema` generates more lengthy IR code by lifting every function (including those helper functions). We debloat those helper functions to better support downstream code comprehension tasks.

Changes on `mctoll`. When facing an external function call which do not have the corresponding implementation in the binary being lifted, `mctoll` requires to at least provide the type of this external function. We extend `ExternalFunctions.cpp` [4] with 20 new external functions encountered during our experiments.

II. Downstream Applications

A. LLVM IR-Based Discriminability Analysis Tool

For discriminability analysis, we employ a recently released LLVM IR embedding and analysis framework, `ncc` (published at NeurIPS'19 [3]) to measure the discriminability of generated IR code. We strictly follow the setup provided by the `ncc` paper to launch evaluations on the POJ-104 dataset [7].

Nevertheless, the currently released version of `ncc` seems buggy [1]. We also find that `ncc`, using a legacy version of `TensorFlow`, can throw "out of memory (OOM)" errors when processing the POJ-104 dataset. Hence, we use `Pytorch` [8] to completely rewrite `ncc`. `Pytorch` seems to have better memory managements, and we report all experiments can be smoothly finished without incurring any OOM errors.

B. LLVM IR to C Decompiler

This section lists the major patches and extensions on our employed LLVM to C decompiler, `llvmir2hll`. Developers can fix the issues with the following information.

1) **Array Allocation**:
   - *Problem*: As discussed in our paper, `McSema` uses an array to represent the memory stack and all local variable accesses are converted into array lookup. `McSema` indeed uses a rarely used LLVM statement to create array, which is not supported by the LLVM to C decompiler, `llvmir2hll`, used in the evaluation.
   - *Solution*: Let `llvmir2hll` skip converting array allocation.
   - *Patched Functions*: `isDirectAlloca()` starting from line 176 of `llvm_support.cpp`.
2) **Inline Assembly**:
   - *Problem*: Inline assembly can be found in the outputs of `McSema` and `BinRec`.
   - *Solution*: Skip converting inline assembly.
   - *Patched Functions*: `isInlineAsmCall()` starting from line 84 of `llvm_support.cpp`.
3) **Unsupported Vector Type**:
   - *Problem*: `McSema` generates a few VectorType instances (e.g., <2 x i32>) that are not supported by `llvmir2hll`.
   - *Solution*: Convert VectorType into ArrayType.
   - *Patched Functions*: `convert()` starting from line 130 of `llvm_type_converter.cpp`.
4) **Unsupported Vector Type for LLVM zeroinitializer**:
   - *Problem*: store <2 x i64> zeroinitializer is not supported by `llvmir2hll`.
   - *Solution*: Convert VectorType into ArrayType.
   - *Patched Functions*: `convertZeroInitializer()` starting from line 274 of `llvm_constant_converter.cpp`.
5) **Unsupported Instruction**:
   - *Problem*: The `extractelement` instruction, which extracts a single scalar element from a vector at a specified index, is not supported by `llvmir2hll`.
   - *Solution*: Skip `extractelement` instruction when handling the corresponding store instructions.
   - *Patched Functions*: `shouldBeConvertedAsInst()` starting from line 279 of `llvm_value_convert.cpp`. `shouldBeConverted()` starting from line 75 of `basic_block_converter.cpp`.
6) **Unsupported Instruction**:
   - *Problem*: The `insertelement` instruction, which inserts a scalar element into a vector at a specified index, is is not supported by `llvmir2hll`.
   - *Solution*: Skip `insertelement` instruction when handling the corresponding instructions.
   - *Patched Functions*: `shouldBeConverted()` starting from line 76 of `basic_block_convert.cpp`.
7) **Unsupported Instruction**:
   - *Problem*: The `shufflevector` instruction constructs a permutation of elements from two input vectors, returning a vector with the same element type as the input and length that is the same as the shuffle mask. This is is not currently supported by `llvmir2hll`.
   - *Solution*: Skip `shufflevector` instruction when handling the corresponding instructions.
   - *Patched Functions*: `shouldBeConvertedAsInst()` starting from line 285 of `llvm_value_converter.cpp`.
8) **Unsupported Instruction**:
   - *Problem*: `getelementptr` instructions perform address calculation of LLVM aggregate data structures (e.g., array). When types of certain elements in the aggregate data structures are missing (e.g., due to imprecise binary lifting), `llvmir2hll` cannot proceed further.
   - *Solution*: Skip `getelementer` instructions with no element data type information.
   - *Patched Functions*: `shouldBeConvertedAsInst()` at line 285 of `llvm_value_converter.cpp`.
9) **Undefined Metadata**:
   - *Problem*: `McSema`[0] generates certain undefined metadata as function parameters, for instance, `declare !remill.function.type !1240 void @llvm.dbg.declare(metadata, metadata, metadata)`.
   - *Solution*: Convert "metadata" into integers to smooth decompilation.
   - *Patched Functions*: `convert()` at line 78 of `llvm_type_converter.cpp`.
10) **Segmentation Fault**:
   - *Problem*: During decompilation, we have observed that several recursive calls (e.g., on line 576 of `ordered_all_visitor.cpp`) could potentially run out of the stack memory space of `llvmir2hll` and lead to its segmentation fault.
   - *Solution*: Compiling the decompiler with release option can help to alleviate some cases. We further change stack size to 800MB with `ulimit -s 819200`. As reported in our paper, all decompilation can be smoothly processed except processing the `gcc` and `perlbench` LLVM IR programs lifted by `BinRec`, and `gcc` lifted by `RetDec`.

III. Errors in McSema Output Patterns

Our study in the main paper shows that `McSema` mostly generates incorrect wrappers for the `main` function when processing ARM64 cases. We present a case study and comparison in Fig. 1. Note that for the first case (x86 platform

```
define dllexport void @main() #5 !remill.function.type
{

  tail call void asm sideeffect \
  "pushq $0;pushq $$0x4005f0;jmpq *$1;", "*m,*m,~{dirflag},~{fpsr},~{flags}"\
  (%struct.Memory* (%struct.State*, i64, %struct.Memory*)** nonnull @6, void ()** nonnull @1)

  ret void
}
```

(a) Main wrapper in IR code lifted from x86 code by RetDec

```
define dllexport void @.init_proc() #4 !remill.function.type
{

  tail call void asm sideeffect \
  "nop;", "*m,*m,~{dirflag},~{fpsr},~{flags}"\
  (%struct.Memory* (%struct.State*, i64, %struct.Memory*)** nonnull @6, void ()** nonnull @1)

  ret void
}
```

(b) Main wrapper IR code lifted from AArch64 code by RetDec

Fig. 1. Errors in the wrapper function for the `main` function when processing ARM64 cases.

lifting), the wrapper function will transfer the control flow to a function named `_mcsema_attach_call`, and further transfer to the `main` function. However, the wrapper function in Fig. 6(b) actually did nothing (see the `nop` instruction in Fig. 6(b)) before finishing the execution. This primarily leads to the functionality testing failure of `McSema` on the ARM64 platform. We have reported this issue to the developer of `McSema`.

References

[1] NCC Errors. https://github.com/spcl/ncc/issues/19, 2019.

[2] Anil Altinay, Joseph Nash, Taddeus Kroes, Prabhu Rajasekaran, Dixin Zhou, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, Cristiano Giuffrida, et al. BinRec: dynamic binary lifting and recompilation. In *EuroSys*, 2020.

[3] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. NIPS 2018, pages 3588–3600, 2018.

[4] bharadwajy. Externalfunctions.cpp. https://github.com/microsoft/llvm-mctoll/blob/master/ExternalFunctions.cpp, 2020.

[5] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *Acm SIGPLAN Notices*, 46(3):265–278, 2011.

[6] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. BISTRO: Binary component extraction and embedding for software security applications. ESORICS. 2013.

[7] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *NIPS*, 2019.