Artifact: Extracting Instruction Semantics Via Symbolic Execution of Code Generators

1. ARTIFACT DETAILS

- **Author Contact Details**: Niranjan Hasabnis, nirhasabnis@gmail.com
- **Artifact License**: GPL
- **URL**: https://hub.docker.com/r/seclab/eissec/
- **Scorecard**
  - **Insightful**: Extracting instruction-set semantics is a pressing problem faced by many binary analysis or binary instrumentation systems. Artifact describes a design of a symbolic execution system to extract semantic model from GCC's x86 code generator. The symbolic execution system is designed in such a way that it can be applied to handle complex source code such as GCC's code generator.
  - **Useful**: Existing approach to extract instruction-set semantics is manual one. Manual extraction is a tedious task and takes considerable manual efforts. Our artifact describes an automated approach to extract semantics model from GCC's code generator. Moreover, by relying on compiler authors' efforts to support advanced instructions in code generators, our artifact ensures that the extracted model is complete with respect to advanced instructions.
  - **Usable**: Artifact includes complete source code of the system along with a test code generator and full x86 code generator in a Docker image. Artifact also includes instructions to compile the source, build an executable and test it on a dummy and full code generator. Instructions also describe input/output of every step and makes understanding the steps easier.

2. DESCRIPTION

Artifact comes as a Docker image that can be installed on any host on which Docker can be installed (Linux, Windows, MacOS). Instructions for using artifact are as follows:

- **Docker installation.** Please refer to steps at https://docs.docker.com/engine/installation/ to install Docker on your machine. EISSEC docker image has been tested on Ubuntu-14.04 x86_64.
- **Using EISSEC Docker image.** Execute commands below to pull EISSEC Docker image, create a container from the image and get a shell access to the container.

```
$ docker pull seclab/eissec
$ docker create -it --name eissec seclab/eissec
$ docker exec -it eissec bash
```

After executing above commands, we get shell access to the EISSEC container. Inside the container, in `eissec` directory, we can notice the layout of EISSEC source package. `README` file under `eissec` directory explains the layout. Now execute commands below inside the container to build the EISSEC executable and extract the model from x86 code generator.

```
$ cd eissec
$ source env_setup.sh
$ make
$ cd test/x86
$ ./testmodel dummy.c
$ ./fullmodel dummy.c
```

- **Output of testmodel.**

```
$ ./testmodel dummy.c
```

(This command will dump assembly to RTL mapping for a sample code generator.) One of the dumped rules can look like Figure 1.

In the Figure, $\_G$ are Prolog variables. The figure shows the mapping rule for `cmpb` x86 instruction and the operand of `cmpb` is represented by a variable $\_G2101$. It is a register (can be recognized by the fact that % is dumped before $\_G2101$). RTL corresponding to the assembly specifies how it maps $\_G2101$ in RTL variables. For variables in assembly instruction, RTL variables are obtained by solving constraints dumped for RTL variables.

Conceptually, rules are functions (say f), where $f(x,y,z)=o$, where x, y, and z are variables in assembly and o is output produced by f (RTL). Relations between x, y, z and o is captured by the set of constraints for every

```
_G2101 in 21..28,
_G653 in 36..138, _G653 #>= _G588, _G184 #>= _G653,
_G462 in -1..32, _G462 + 10 #= _G494,
_G494 in 9..42, _G494 + _G543 #= _G588, _G462 + 10 #= _G494, _G494 * 3 #= _G543,
_G543 in 27..126, _G494 + _G543 #= _G588, _G494 * 3 #= _G543,
_G588 in 36..137, _G653 #>= _G588, _G588 #=< _G184 + -1, _G494 + _G543 #= _G588,
_G184 in 37..138, _G184 #>= _G653, _G588 #=< _G184 + -1,

map:  cmpb    %_G2101 -> insn(_G9, rtl(15,_G27,union_u([rtl(_G184,_G191,_G198,_G205),
                                                        rtl(_G462,_G469,_G470,_G471),
                                                        rtl(_G653,_G660,_G667,_G668)|_G393],
                                  _G39,_G46,_G53,_G54),_G73))
```

Figure 1: **Example of mapping rule for `cmpb %reg` x86 instruction**

rule. There could be more than one possible instantiations that solve a set of constraints for a single rule. These instantiations correspond to different operands that a rule will support. For instance, for `cmpb` rule, value of $\_G2101$ is between 21 and 28. So this rule maps *cmpb* for 8 different registers (having value between 21 and 28 — GCC represents x86 registers such as `eax`, `ebx`, etc, by number 21, 22, so on till 28, resp.)

Steps involved in extracting the model are:

- *Transform GCC's code generator code for symbolic execution.*

  EISSEC uses CIL-based source-to-source transformation. Transformed code when executed perform symbolic executions of the code generator. `pkgs/cil-1.4.0/bin/cilly` is the source-to-source transformer. The transformer transforms insn-recog.c (from `test/x86`) while compiling (`test/x86`). Transformed version of `insn-recog.c` can be found in `/tmp/insn-recog.cil.c`. Reading `insn-recog.cil.c` may not be easy, but for a curious reader, the transformed code for `insn-recog.c` is found in this file.

- *Constraints sent to Constraint solver.*

  When `testmodel` or `fullmodel` is executed, constraints sent to `cons_solve` are dumped in `/tmp/logcs` file. If you would like to change the location, please modify `driver.c` in `test/x86` and recompile. `/tmp/logcs` dumps all the constraints sent to the constraint solver and the solver's response to those constraints.

- **Maturity.** Artifact describes EISSEC symbolic execution system used to extract Assembly-to-IR model from compiler's code generator. Extracting assembly-to-IR semantic model is the one of the complex tasks in the process of assembly-to-IR translation. Typical approach to this problem is to manually extract assembly-to-IR semantic model. EISSEC automates this task, and the *artifact demonstrates how EISSEC automatically extracts the model*.

  The artifact does not describe *how to apply extracted semantic model for assembly-to-IR translation process*. We definitely have a prototype for this purpose, but it is not fully mature and requires manual intervention.

- **Source code.** EISSEC's source code is fully contained inside `eissec` directory inside the Docker container. `src` directory inside `eissec` contains the source code of source-to-source transformer (`transformer`) (for symbolic execution), helper functions for symbolic execution (`symhelper`) and interface to constraint solver (`csolve`). Additionally, EISSEC source code can also be downloaded from http://seclab.cs.sunysb.edu/seclab/download.html.