Artifact: Extracting Instruction Semantics Via Symbolic Execution of Code Generators

Niranjan Hasabnis
Intel
niranjan.hasabnis@intel.com

R. Sekar
Stony Brook University
sekar@cs.stonybrook.edu

1. SCORECARD

1.1 Insightful

Binary analysis and instrumentation form the basis of many tools and frameworks for software debugging, security hardening, and monitoring. Accurate modeling of instruction semantics is paramount in this regard, as errors would cause instrumented programs to crash, or worse, bypass security checks and/or monitoring. Semantic modeling is a daunting challenge since modern instruction sets are large and complex: both x86 and ARM support more than a thousand opcodes, with instruction manuals stretching to about a thousand pages. The task of semantic modeling is made even more complicated by the fact that most existing binary analysis/instrumentation systems, including DynamoRio [2], Pin [4], QEMU [1], and Valgrind [5] rely on manual efforts to encode semantics of target ISA into a specification. Manual modeling not only makes updates to specifications complicated (when new instruction set extensions are released), but also restricts applicability to a limited number of ISAs. We are only aware of LISC [3] that use compilers to automate semantic modeling effort. Unfortunately, LISC relies on learning semantic models from code generator logs, which does not guarantee completeness (all target ISA instructions are covered) of those models.

In our artifact, we demonstrate a novel approach, called `EISSEC`, that automates the task of semantic modeling by relying on an insight that compiler authors model semantics of target ISA in code generator specifications. By relying on a new symbolic execution based technique to extract the encoded semantics from a compiler's source code, `EISSEC` guarantees completeness of the extracted semantic model.

1.2 Useful

An approach of extracting instruction semantic model from compiler code generators offers several advantages. Not only does this approach avoid manual labor, but it also ensures that the approach is architecture-neutral, i.e., it can support any of the numerous architectures already supported by a compiler such as GCC. Additionally, since compilers model most (if not all) target instructions including advanced instructions, `EISSEC` ensures that the extracted models are complete with respect to the advanced instructions. We found that many existing systems such as Valgrind lack support for advanced instructions, even from a popular ISA such as x86.

We performed symbolic execution of the GCC-4.6.4 x86 code generator which models all the x86 advanced instruction sets such as SSE, AVX, etc. C code built by GCC for the x86 code generator specification is about 120KLoC. `EISSEC` performed symbolic execution of this code to extract x86 semantic model. It took around 14 days for extraction, which were further reduced to roughly 6 days with various hardware/software optimizations that we implemented in `EISSEC`.

We evaluated the completeness of extracted x86 semantic model by comparing against all x86 binaries on Ubuntu-14.04. This test tells us if we covered all instructions that are contained in applications typically distributed with the OS. On this test, we found that the semantics of 99.66% of instructions from Ubuntu-14.04 binaries is already encoded in the extracted model. A manual approach to encode semantics of 99.66% of instructions from Ubuntu-14.04 binaries would require significant time and effort.

1.3 Usable

The artifact includes complete source code of `EISSEC` along with a test code generator and full x86 code generator in a Docker image. The Docker image comes with all the required packages preinstalled, which enables user to start using `EISSEC` directly. In addition to Docker image, we also provide our artifact as a VM image. All images can be downloaded from http://http://seclab.cs.sunysb.edu/seclab/eissec/.

The artifact also includes README file with instructions to compile the source, build an executable and test it on a dummy and full code generator. The instructions also describe input/output of every step and makes understanding the steps and their output easier.

2. REFERENCES

[1] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Conference on Annual Technical Conference*, 2005.

[2] Derek L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Cambridge, MA, USA, 2004.

[3] Niranjan Hasabnis and R. Sekar. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, 2016.

[4] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, 2005.

[5] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07. ACM, 2007.

3. ARTIFACT 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 start 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 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.

```
_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**

- **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.