RUSTINA: Automatically Checking and Patching Inline Assembly Interface Compliance (Artifact Evaluation)

Accepted submission #992 – "Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine"

Frédéric Recoules
Univ. Paris-Saclay, CEA, List
Saclay, France
frederic.recoules@cea.fr

Sébastien Bardin
Univ. Paris-Saclay, CEA, List
Saclay, France
sebastien.bardin@cea.fr

Richard Bonichon
Tweag I/O
Paris, France
richard.bonichon@gmail.com

Matthieu Lemerre
Univ. Paris-Saclay, CEA, List
Saclay, France
matthieu.lemerre@cea.fr

Laurent Mounier
Univ. Grenoble Alpes, VERIMAG
Grenoble, France
laurent.mounier@univ-grenoble-alpes.fr

Marie-Laure Potet
Univ. Grenoble Alpes, VERIMAG
Grenoble, France
marie-laure.potet@univ-grenoble-alpes.fr

I. GOAL

The main goal of the artifact is to support the experimental claims of the paper #992 "*Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine*" [3] by making both the prototype and data **Available** to the community. The expected result is the same output as the figures given in Table I and Table IV (appendix C) of the paper. In addition, we hope the released snapshot of our prototype is simple, documented and robust enough to have some uses for people dealing with inline assembly.

II. CONTEXT

Inline assembly is still a common practice in low-level C programming, typically for efficiency reasons or for accessing specific hardware resources. Such embedded assembly codes in the GNU syntax (supported by major compilers such as GCC, Clang and ICC) have an *interface* specifying how the assembly codes interact with the C environment. For simplicity reasons, the compiler treats GNU inline assembly codes as blackboxes and relies only on their interface to correctly glue them into the compiled C code. Therefore, the adequacy between the assembly chunk and its interface (named *compliance*) is of primary importance, as such compliance issues can lead to subtle and hard-to-find bugs. We propose RUSTINA [3], [1], the first automated technique for formally checking inline assembly compliance, with the extra ability to propose (proven) patches and (optimization) refinements in certain cases. RUSTINA is based on an original formalization of the inline assembly compliance problem together with novel dedicated algorithms. Our prototype has been evaluated on 202 Debian packages with inline assembly (2656 chunks), finding 2183 issues in 85 packages – 986 significant issues in 54 packages (including major projects such as ffmpeg or ALSA), and proposing patches for 92% of them. Currently, 38 patches have already been accepted (solving 156 significant issues), with positive feedback from development teams.

RUSTINA is part of the TINA toolset for the automatic analysis of low-level C code with inline assembly [2]. It is built on top of the binary-level code analyzer BINSEC [4] and the C code analyzer Frama-C [5].

III. MISC

Contact. Frédéric Recoules is the corresponding author: frederic.recoules@cea.fr.

Requirements. Hardware requirements are low. Any personal 64bit processor laptop should be able to replay the benchmark. Depending on the reviewer situation, it will be needed to have an access to *either*: **Ubuntu 18.04** with make, python2 and python-pandas installed, **docker** or **VirtualBox**. Replaying the benchmark requires no particular skill. Yet, basic knowledge of shell, python scripts, CSV file format and GNU inline assembly could help the reviewer experimenting on their own.

Availability. The artifact is made publicly available on Github at: https://github.com/binsec/icse2021-artifact992/, and more information on RUSTINA can be found at https://binsec.github.io/new/publication/1970/01/01/nutshell-icse-21.html.

Archive. The artifact is archived in a persistent Zenodo repository at https://doi.org/10.5281/zenodo.4601172.

IV. Contents

RUSTInA. This is the software part of the artifact, packaged inside a standalone AppImage (equivalent of `.exe` on Windows). This prototype is able to digest C preprocessed (i.e where macro are resolved) source files that contain inline assembly chunks to either:

Print important information about the compliance of the chunks in a human readable fashion;

Log individual issue information in a CSV format.

Instruction sets currently supported are x86-32bit and ARMv7.

Trimmed data. This sample contains all assembly chunks studied in the experimental evaluation (section VII.) of the prototype. That is *2656* relevant chunks over *202* projects for the x86 architecture and *392* relevant chunks of *3* projects for the ARM architecture. The chunks have been extracted from their original source and gathered in one place by project to save space and ease repeatability. Data contain only assembly related slices of the original codes, but location information allows to link them back to the place they were extracted.

Scripts. The artifact comes with predefined rules in order to automatically replay the benchmark and output the formatted key figures presented in the paper (Table I). The given python scripts can be used to format the log of any subpart of the benchmark or custom experiments.

In addition, the repository comes with detailed instructions, documentation, examples and a preprint of the accepted paper.

V. Replication process

Replaying the benchmark is fully automatic. The material can be found on the GitHub repository hosted at https://github.com/binsec/icse2021-artifact992/.

It is available on 3 formats:

1) the repository itself can be cloned using git when running an Ubuntu 18.04;
2) a docker container can be downloaded in the release panel for docker users;
3) an Appliance (`.ova`) can be downloaded in the release panel for VirtualBox users.

The 3 formats are equivalent since the containers are just a wrapper around the content of the repository (including the AppImage) with resolved dependencies. Only space disk usage changes.

At the root of the directory (containers already start here), a single command launches the entire benchmark:

```
$ make all
```

The command will call RUSTInA on each sample file of the `samples/` directory and output a log (`.csv`) in the corresponding `data/` directory. Each file will be processed twice, once without precision enhancer and once with bit level liveness and symbolic expression folding enabled. The special log files `data/debian.csv` gather all the results. Then the python scripts `script/table1.py` and `script/table2.py` digest the log to output the formatted figures of, respectively, Table I and Table IV.

Subsequent call to `make all` will directly output the result. In order to restart from scratch, use the command:

```
$ make clean
```

VI. Origin of the data-set

The data-set is exactly the one used in the paper. The snippets of assembly code come from the sources of Debian Jessie (*8.11*) packages. We ran a long session of package installations during which each call to a C compiler was intercepted. All C preprocessed sources were scanned for assembly chunks and relevant ones were saved. Files were merged by project and then sliced to keep only functions containing assembly chunks and their syntactic dependencies. All credits belongs to their original authors.

VII. Other uses

RUSTInA (`bin/rustina-x86_64.AppImage`) can be used as other command line tools.

It takes 3 optional parameters and a list of preprocessed C files (ending with `.i` extension).

The optional parameters are:

```
-b --batch <path>     Logs issues in the file <path>.
-v --verbose          Outputs issues and patches on <stdout>.
-x --basic            No precision enhancer.
```

Note. Due to AppImage internal behavior, every path given in the command line has to be fully qualified. This can be easily achieved by prefixing relative file path with "`$(pwd)/`".

In order to process the file, RUSTInA requires the C source to have no macro. If your test file does not contain any, simply rename the `.c` or `.h` extension by `.i`. In other case, the compiler option `-E` should work for gcc or Clang to output file after macro substitution.

Additional instructions and examples are included in the repository to experiment with RUSTInA.

References

[1] F. Recoules, S. Bardin, R. Bonichon, M. Lemerre, L. Mounier, M. Potet. RUSTInA in a nutshell. https://binsec.github.io/new/publication/1970/01/01/nutshell-icse-21.html

[2] F. Recoules, S. Bardin, R. Bonichon, L. Mounier, M. Potet. Get Rid of Inline Assembly through Verification-Oriented Lifting. 34th IEEE/ACM International Conference on Automated Software Engineering (ASE'19) IEEE

[3] F. Recoules, S. Bardin, R. Bonichon, M. Lemerre, L. Mounier, M.-L. Potet. Interface Compliance of Inline Assembly: Automatically Check, Patch and Refine. 43rd International Conference on Software Engineering (ICSE 2021). ACM

[4] A. Djoudi and S. Bardin. BINSEC: Binary Code Analysis with Low-Level Regions. 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'15). Springer

[5] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles and B. Yakobowski. Frama-C: A software analysis perspective. Formal Asp. Comput, 27(3), 2015