

LELAPE

GHADIR Group
Universidad Complutense de Madrid

February 18, 2022

Contents

1	Introduction	1
2	Theoretical Background	3
3	How to install LELAPE	5
3.1	Why Julia?	5
3.2	How to install Julia	5
3.3	Installing LELAPE	6
3.4	Recommended packages	7
4	How to use LELAPE	9
4.1	Formatting input data	9
4.2	Setting up the analysis	10
4.3	Functions in the module	11

Chapter 1

Introduction

Probably, the reason of reading this document is that you are a researcher that tests electronica devices under radiation. Even more, perhaps you are not interested in any electronic devices but, at this moment, only in commercial-off-the-shelf (COTS) memories.

Under the umbrella of memories, different devices are comprised. Let us enumerate them:

- Static Random Access Memories (SRAMs)
- Dynamic Random Access Memories (DRAMs)
- Non-volatile memories (Flash, PRAM, MRAM, etc.)
- Configuration memory in FPGAs
- Cache memory in microprocessors and microcontrollers
- ...

It is well known that these elements will show bitflips if they are exposed to protons, neutrons, heavy ions, ... The common procedure is to write a pattern in the memory and look for errors during read-back. These errors will be labeled indicating the word address and the flipped-bit position in the word.

Unfortunately, this is the so-called *phisical address* and it is impossible to relate it to the exact physical position on the integrated circuit. And this leads to a serious problem at the time of interpreting results. Nearby bitflips are probably caused by pernicious multiple cell upsets (MCUs) but they will not discovered unless the researcher has somehow got the information to relate any logical address to its physical address (an X, Y pair on the silicon surface).

However, the presence of MCUs will leave a signature in the set of the logical addresses of bitflips that can be used to group pairs of related bitflips and classify them in single or multiple events. No matter are the multiple events hidden among the rest of bitflips, we can track and locate them.

LELAPE is the Spanish acronym for *Listas de Eventos Localizando Anomalías al Preparar Estadísticas* , which is equivalent in English to LAELAPS (*Lists of All Events Locating Anomalies at Preparing Statistics*). In Greek mythology, LELAPE, or LAELAPS, is Zeus' hound, with the magic skill to track and hunt any prey however hidden it may be. In a similar way, LELAPE is a software tool able to inspect sets of apparently random logical addresses of bitflips and discover those that are members of the same multiple event.

Chapter 2

Theoretical Background

TBD

Chapter 3

How to install LELAPE

3.1 Why Julia?

The Julia language (<https://julialang.org>) was released in 2012 as a possible solution the classical “Two Languages’ Problem”¹: a language easy to learn and develop in is usually slow and vice versa, so too often algorithms must be developed in one language and rewritten in another more efficient one. This language was specifically built to achieve speed, efficiency and clarity.

3.2 How to install Julia

Julia, an open software released with MIT license, can be downloaded and installed from <https://julialang.org/downloads/> for different operating systems and architectures. Depending on the system, the binary files will be installed (Microsoft Windows, macOS) or just uncompressed (GNU/Linux, FreeBSD).

If no additional tools is installed, Julia will be executed in a REPL as it is shown in Fig. 3.1. However, most of the users prefer to use the language in conjunction with IDEs or notebooks such as:

- **Visual Studio Code:** Popular IDE developed by Microsoft with plugins for Julia. It can be downloaded from <https://code.visualstudio.com/> and the plugins installed through the “Extensions” tool.

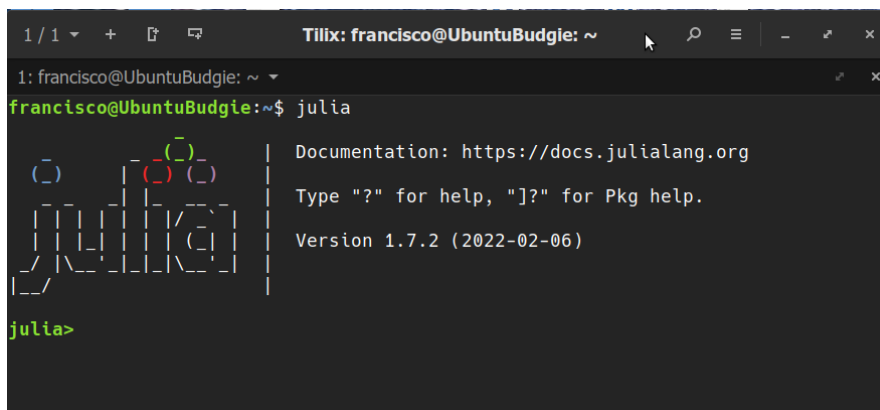
A screenshot of a terminal window titled "Tilix: francisco@UbuntuBudgie: ~". The terminal shows the command "julia" being entered at the prompt "francisco@UbuntuBudgie:~\$". The output is the Julia REPL welcome screen, which includes a stylized ASCII art logo of the letter 'J' on the left. To the right of the logo, the text reads: "Documentation: https://docs.julialang.org", "Type '?' for help, ']' for Pkg help.", and "Version 1.7.2 (2022-02-06)". The prompt "julia>" is shown at the bottom of the screen.

Figure 3.1: Example of the REPL’s welcome screen for Julia on a machine running Ubuntu Budgie.

-
- **Atom:** Just like VS Code, it is a general-purpose IDE with plugins for Julia.
 - **Jupyter:** Although it is typically used for Python, it is also appropriate for Julia. Indeed, Jupyter is an acronym for **Julia-Python-R**. There are different ways of installing Jupyter. In systems with Microsoft Windows OS, the most simple way is to install Anaconda <https://www.anaconda.com/products/individual>. Despite the fact of being designed for Python, Julia can be used as calculation engine. In GNU/Linux there are smaller packages to install Jupyter. For example, in Ubuntu the simple instruction `sudo apt install jupyter-notebook` will install the software in your computer. Later, it is necessary to install an additional package inside Julia but this will be studied a bit later.
 - **Pluto:** A notebook following the philosophy of Jupyter but specifically developed for Julia and easily extensible with JavaScript. Unlike the previous tools, it is installed inside Julia, not along with it.

Atom, Jupyter and Pluto require the installation of additional packages. As Jupyter is probably the most popular tool, it is installed inside Julia REPL with the following instructions:

```
using Pkg; Pkg.add("IJulia")
```

This adds the software, which is launched as follows:

```
using IJulia; notebook()
```

The other packages are installed following a similar procedure.

```
Using Pkg; Pkg.add("Pluto"); using Pluto; Pluto.run()
```

There are other options if you prefer cloud computing. For example, in spite of the fact that its primary use is running Python code, Google Colab is compatible with Julia language. For further information (and also learning a little Julia), you can read *Julia for Pythonists* and use the *Julia Colab Template*. However, this solution is not recommended due to some problems at installing external packages as well as at loading data files.

3.3 Installing LELAPE

LELAPE is built as a module. In Julia, a module is a set of elements such as variables, functions, etc. that can be loaded at will. The procedure is the following:

1. Download the ZIP system from the website and decompress it. Also, you can clone the site with git.
2. Find the folder where a file called `LELAPE.jl` is located.

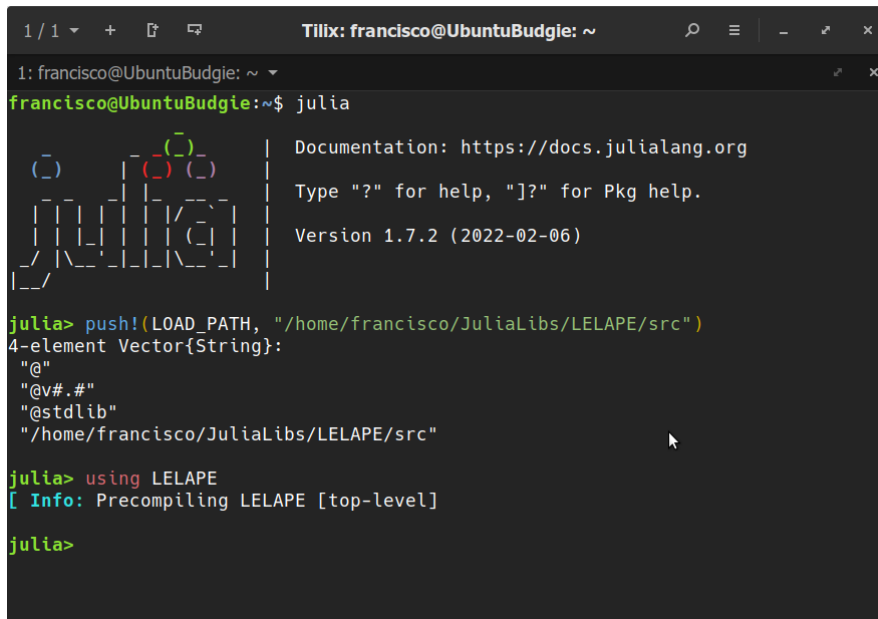
A screenshot of a terminal window titled "Tilix: francisco@UbuntuBudgie: ~". The terminal shows the Julia REPL prompt. The user enters `julia` at the prompt. The REPL displays the Julia logo, documentation link (<https://docs.julialang.org>), help instructions, and version `1.7.2 (2022-02-06)`. The user then enters `push!(LOAD_PATH, "/home/francisco/JuliaLibs/LELAPE/src")`, which outputs a 4-element `Vector{String}` containing `"@"`, `"@v#.#"`, `"@stdlib"`, and `"/home/francisco/JuliaLibs/LELAPE/src"`. Next, the user enters `using LELAPE`, which outputs `[Info: Precompiling LELAPE [top-level]`. Finally, the user enters `julia>` at the prompt.

Figure 3.2: How to indicate Julia where LELAPE is installed, and how to load it.

3. Copy the full path pointing to this folder (`PATH_TO_FOLDER`) and execute in REPL, Jupyter or the notebook you use the following command:

```
push!(LOAD_PATH, "PATH_TO_FOLDER")
```

For example, if `LELAPE.jl` is found in `/home/johndoe/Download/LELAPE/src/`, the instruction is:

```
push!(LOAD_PATH, "/home/johndoe/Download/LELAPE/src")
```

Thus, Julia knows where to find the module.

4. Now, just launch LELAPE with the following instruction:

```
using LELAPE
```

After a few seconds to precompile the library, the functions are loaded. Fig. 3.2 shows a practical example.

3.4 Recommended packages

There are many Julia packages at the user's disposal that can be found on <https://juliapackages.com/>. Some packages are extremely popular:

- **Revise**: Useful for code developers since it allows reloading user functions without restarting Julia and losing information.
- **OhMyREPL**: Intelligent highlighting of elements in REPL.

Both are installed with `Pkg.add()`. Another interesting package to have is **DelimitedFiles**, which allows reading and writing CSV files. It is installed through:

```
using Pkg; Pkg.add("DelimitedFiles")
```

This package is necessary to use the illustrative Jupyter notebooks that are provided along with LELAPE.

Chapter 4

How to use LELAPE

4.1 Formatting input data

You are supposed to have performed experiments on some memory element. Tests were performed as:

- **Static:** The device was written, sent to standby mode, irradiated and eventually read. The content after the irradiation was compared to the initially writing.
- **Pseudostatic:** Similar to static ones, but standby intervals are shorter than the irradiation time and the memory is read several times during the irradiation. Usually, flipped bits are corrected on the flight.

In both cases, the researcher saves information about the bitflips: Word address, read content, In order to use LELAPE, radiation test data must be converted to a matrix with three or four columns. The meaning of the columns is the following:

- First Column: Word Address where bitflips were observed.
- Second Column: Content in the word address after the radiation tests.
- Third Column: Content in the word address before the irradiation.
- Fourth Column: In pseudostatic tests, cycle in which the bitflip was observed. This column can be omitted in the static tests or replaced by a column full of ones.

LELAPE needs this elements to be converted to a UInt32 matrix¹ so it is important that all the elements of the matrix, including the cycle label, are in this format or, at least, in some kind of integer. This makes dangerous labelling the fourth column with words or letters.

A simple solution consists in grouping the data results in CSV format and read this text file with `readddlm`, included in the DelimitedFiles package. In the Jupyter folder, you can see some examples that can guide you to adapt your own data. One advantage of this function is that it can automatically convert the data to the required format, as shown in the Jupyter notebooks.

¹In Julia, typing variable is optional but encouraged to speed up calculations.

4.2 Setting up the analysis

Before starting the analysis, we must define some additional variables to indicate the software how to proceed. These variables are the followong:

- **LA:** Variable in integer format. It indicates the memory size in words (not in bits!). It is often a power of 2.
- **WordWidth:** Also an integer, it indicates the number of bits per word. Typically 8, 16, 32 but other values are possible.
- **Operation:** A string variable to indicate the mathematical operation used to create the DV set. So far, only two options are implemented:
 - *XOR:* Addresses are xored bit to bit. This mode is set with the ``XOR`` value.
 - *Positive subtraction:* The absolute value of the difference of addresses is returned. It is marked with "POS".

In practice, we have observed that the former is appropriate for SRAMs and the latter for FPGAs. However, this idea may be erroneous due to the use of few and partial radiation test data.

- **UsePseudoAddress:** The pseudoaddress is defined as follows: let us suppose that we have observed a bitflip in the k -th position of the NWA -th word address, The word width is W and $k = 0$ corresponds to the least significant bit, $W - 1$ to the most significant one. Hence, the pseudoaddress of the bitflip is:

$$PSA = NWA \cdot W + k$$

This value is full of meaning in FPGAs since just returns the position of the cell in the bitflips. It is completely artificial in some SRAMs but somehow analysis using the pseudoaddress instead of the word address are more accurate and efficient.

The researcher can set this variable to `true` or `false` at will.

- **KeepCycles:** In pseudostatic tests, this boolean variable indicates that the system must use the information about the cycles (fourth column) or just using the set of data as a whole.
- **TraceRuleLength:** This an integer variable with 1,2 or 3 as allowed values. LELAPE looks for anomalously repeated elements in the DV set with very few ones in binary representation. The user can decide if looks for elements with 1, 2 or 3 ones or less and include them as candidates to detect pairs.
- ε : A float number always positive but close to 0. If the expected number of elements repeated k times in the DV set is lower than ε , we must consider this number of repetitions impossible. If higher, we determine than at least an element can appear k times just due to randomness. Default value is 0.05.

A very low value of ε will exclude false positive but also genuine values relating pair of addresses in an MCU. On the contrary, if it is chosen too low, false positives might be taken as good ones.

-
- **LargestMCUSize:** During the search of critical DV values, LELAPE starts to group addresses in provisional MCUs that grow large and large as new possible critical DV values are tested. Unfortunately, sometimes this process does not find a stable solution and goes on looking for it despite being unrealistic. This parameter is used to stop the calculation since it informs the software of not considering MCUs with more than *LargestMCUSize* addresses. By default, it is set to 200.

4.3 Functions in the module

TBD

