LELAPE

GHADIR Group
Universidad Complutense de Madrid

Contents

Chapter 1

Introduction

1.1 The reason of using LELAPE

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 *physical 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.

LELAPE can be found on its GitHub website (https://github.com/fjfrancopelaez/LELAPE).

1.2 Acnowledgments

This tool was supported by the Spanish "*Ministerio de Ciencia e Innovación (MICINN)*" by means of the PID2020-112916GB-I00 project.

1.3 How to reference LELAPE

If you have successfully used this tool and the results are worth for academic publications, we ask you for including the following references:

- Obviusly, this site.
- **The Julia Language**: J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "*Julia: A fresh approach to numerical computing*," SIAM review, vol. 59, no. 1, pp. 65–98, 2017 (DOI: 10.1137/141000671).

1.4 License of use

Chapter 2

Theoretical Background

2.1 Memory devices as a metric space-like sets

First of all, we are going to depict a memory as a set of ordered elements, which can contain only two possible values. Clearly, this elements are the memory cells, and values are **0** & 1.

As the set is ordered, we can assign an index number to every cell, which will be a positive natural number. For example, in FPGAs, this index is the position of the bit in the bitstream. In SRAMs, we can assign any cell the following index:

$$Index = ADD \times W + k \tag{2.1}$$

ADD being the word address, W the wordwidth, and k the bit positiion in the word. From now on, we will call this index as *pseudoaddress* in SRAMs. As this definition is arbitrary, anyone can postulate others for the *pseudoaddress*, keeping in mind that any new definition must be bijective. Thus,

$$Index_2 = ADD \times W + (W - k)$$

could also be valid.

It is immediate that the memory size, L_N is the number of available cells. If this memory is divided in L_A W-bit width words, $L_N = L_A \times W$ holds. In the simplest scenario, cell indexes are distributed between 0 and $L_N - 1$, but more complicated situations can occur:

- In some microcontrollers, we can download its content as a binary file and the SRAM memory is placed between A_0 & $A_0 + L_N - 1$ positions. Other memories such as the Flash memory are in other parts of the file.
- In Xilinx FPGAs, the downloaded bitstream contains information about the configuration memory, flipflops, and BRAM. The content of these parts are split in several pieces and inserted in the bitstream. It is necessary to perform some reverse engineering studies to gather information about how to disassemble the bitstream. Therefore, there will be gaps in the indexes.

The first case is easy to solve just correcting the offset, although under some circumstances this is not strictly necessary for reasons to be explained later. In the second case, some ideas are remove gaps one by one, keep the indexes as they are, etc. It is not so obvious how to proceed.

We are going to defina a *pseudodistance* or *pseudometric* as any funcition $d : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that:

1. $d(a, a) = 0 \quad \forall a \in \mathbb{N}$
2. $d(a, b) = 0 \Leftrightarrow a = b \quad \forall a, b \in \mathbb{N}$
3. $d(a, b) = d(b, a) \quad \forall a, b \in \mathbb{N}$

In this list, the triangular property ($d(a,c) \leq d(a,b) + d(b,c) \quad \forall a, b, c \in \mathbb{N}$) has not been included. If the function fulfills this fourth condition, it would become a true distance or metric.

We will focus on two functions, which have proven to provide interesting results to analyze events. These are:

- **Bitwise XOR**: Addresses are expressed in binary format and the distance is the result of making XOR operation on pairs of bits in the same position.
- **Positive subtraction**: Or *absolute subtraction*. It is defined as $d(a,b) = |a-b|$. This function is a true metric.

Now, we can briefly return to the case of memory blocks in microprocessors, exposed in previous paragraphs. In spite of the fact that the addresses values could range from A_0 to $A_0 + L_N - 1$, values of the positive subtraction are restricted to $0, \cdots, L_N - 1$, and the same occurs for bitwise XOR is, e. g., A_0 is a power of 2 higher than L_N.

2.2 Characteristics of subsets with randomly picked elements

Let us come back to the simplest case: cell indexes are distributed between 0 & $L_N - 1$. Let us suppose that we randomly pick N_{BF} cells, and we can choose a cell twice or more times. This is exactly what happens when a memory is irradiated: cells are randomly flipped (at least if multiple events do not occur) so, when the memory is read back, the addresses of flipped cells should be randomly distributed. Let us call this set of addresses ADD

In this case, it is possible to demonstrate several properties. For example, assuming that a cell can be hit twice, hence undetected, the expected actual number of flipped cells is:

$$N^*_{BF} = N_{BF} + \frac{N^2_{BF}}{L_N} \tag{2.2}$$

This correction is not necessary for typical experiments [1] so we will assume hereafter that the number of observed bitflips (N_{BF}) is just the total number, N^*_{BF}.

Now, we will build a new set, which we will call *"`Difference Vector (DV)"*, as the pseudodistance values of all possible pairs of addresses of flipped cells:

$$DV = [x = d(a,b) \quad \forall a, b \in ADD, b > a] \tag{2.3}$$

Some properties of this set are:

- In DV, there are
$$N_{DV} = \frac{1}{2} \cdot N_{BF} \cdot (N_{BF} - 1) \tag{2.4}$$
elements, but this is valid only for results with one writing & reading cycle. If there were several rounds, as it occurs in pseudostatic or dynamic tests, N_{DV} should be computed as the sum of the sizes of the partial sets.
- No element is 0. The reason is that every address appears once and only once.
- Some values can randomly appear several times in the DV set.

This last statement is extremely interesting. It is possible to deduce that, if only single bit upsets occur and the bitwise XOR is used [1], the expected number of values appearing k times in the DV set is:

$$N_{R,XOR}(k, N_{DV}) = (\, N_{\ DV} k) \cdot \frac{(L_N - 1)^{N_{DV} - k}}{L_N^{N_{DV}-1}}$$

[1]Nevertheless, we have found it necessary for Monte-Carlo tests

In the case of using the positive subtraction [2], the expression is more complicated but computable:

$$N_{R,POS}\left(k, N_{DV}\right) = \binom{N_{DV}}{k} \cdot \sum_{i=0}^{N_{DV}-k} \frac{(-1)^i}{i+k+1} \cdot \binom{N_{DV}-k}{k} \cdot \frac{2^{i+k}}{L_N^{i+k-1}}$$

Whichever expression we choose, the expected number of repetitions fades away as k grows. Thus, it is possible to calculate the value of k_{th} from which the number of expected elements repeated $k > k_{th}$ times is lower than ε, with $1 \gg \varepsilon > 0$. Therefore, it is almost impossible to find elements in the DV set repeated more the k_{th} times.

But this is true only if there are SBUs. If besides SBUs there are multiple cell upsets (MCUs), it is possible to find elements in the DV set repeated a forbidden number of times. These values are the marks of multiple events, that can be used to reconstruct the number and size of multiple events.

For example, in Example 5 of Jupyter notebooks, we discovered that there was over-abundance of 1, 2, 3230–3234 in the DV set derived from results in an FPGA with positive subtraction. Pairs of flipped bit addresses differing in one of these values are very likely members of a single multiple event. A later study showed that the FPGA configuration memory is organized in columns of 101 32-bit words, hence the physical meaning of 3232 = 32 $\times$ 101, so cells differing 3232 are just in the same row.

2.3 Methods to find and reject anomalously repeated value

2.3.1 The Self-Consistency Rule

Experiments on actual devices show that some values appear in the DV set far more than expected. However, not all of them are marks to relate adjacent cells.

Let us put an example. We have a hypothetical memory where cells are distributed along a simple chain, as in the bitstream of FPGAs. Now, let us suppose that 1 is a true mark to detect events, and that there are 2-bit MCUs in positions (100, 101), (350, 351), and (1500, 1501). As there are 6 cells, we can calculate 15 distances, the values of which appear the following number of times:

- 1 : 3 times
- 150, 1150, 1400: 2 times
- 149, 151, 1149, 1151, 1399, 1401: once

As expected, 1, the true mark, appears more times than the rest of elements. However, three nonsense numbers, 150, 1150 and 1400, appear also several times since they measure the relative distance between identical-shape groups.

This phenomenon is exacerbated if larger events are present. In order to discard false events, we have proposed to include the "*self-consistency rule*". The group of confirmed anomalous values are picked one by one from the set of anomalies following the number of occurrences. When the size of the predicted events is larger than the number of collected critical elements, the process takes a step back to discard the recently added elements and exits. The advantage of this rule is that nonsense anomalies are rejected, with the penalty of possibly discarding genuine anomalies. However, previous experiences led us to consider that this decision is better than to be too permissive and to accept false values, that eventually lead to the detection of unrealistic very large events, fruit of the artificial union of two or more unrelated small ones.

2.3.2 The MCU rule

Let us suppose that we have performed the self-consistency test on a set of addresses of flipped cells and that we have discovered several anomalies that allowed us reconstructing the possible multiple events.

Let us focus now on those events with a size of 3 or more. Cells in every event are obviously adjacent, and perhaps, studying the relation between pairs cells inside every MCU we can discover new critical anomalies that were not discovered during the first check.

In LELAPE, only those anomalies found in MCUs that also appear more than expected are included in the list of genuine marks of events.

2.3.3 The Trace rule

We call "*trace*" of a natural number as the number of ones present in the binary expression of the number. When the bitwise XOR operation is used, discovered anomalies are usually values with only one to three values in binary format.

Therefore, the rule is simple: let us check all the possible natural numbers lower than the memory size with low values of trace (1–3) to verify if they appear too often in the DV set. If so, they will be added to the group of confirmed anomailes.

However, our experience shows that only values with traces of 1 or 2 are good candidates. Trace 3 is risky and 4 or higher are forbidden in LELAPE.

2.3.4 The Shuffle rule

The idea is quite simple. Let us suppose that, after applying previous rules we have discovered a set of anomalies. The idea behind this rule is to combine them in pairs using the distance function in order to obtain new values, and adding them to the set of anomalies if they appear in the DV set more than predicted by the statistical model.

2.3.5 The History rule

This is not exactly a rule, but a sign of being sensible. Let us suppose that you test a device and get a set of anomalies, A_1. Later, you test it again and the obtained values are A_2. As the device is the same, or at least of the same model and the anomalies are linked to the internal structure, it is evident that elements in A_1 are valid for the second experiment, and vice versa. Thus, the union of both sets, $A_1 \cup A_2$ can be used to classify results from both experiments.

Even more, if you had previously tested the device and identified a significant set of anomalies, it is possible to skip the process of obtaining anomalies and straightly apply that set on the test results, saving analysis time.

However, this rule is only applicable if the address pins are correctly identified. For example, in some synchronous memories, these pins are fully configurable so unless the same setup does not change from experiment to experiment, previous results will be useless. Another option is to reorganize the address bits by software before analyzing events.

2.4 Number of false events

Sometimes, two or more single bit upsets may occur in adjacent cells in such a way that a later analysis can erroneously conclude that both belong to a unique multiple cell upset [3]. This is completely false, and the number of expected false events increases with the number of bitflips. It is possible to deduce that that number increases with the number of flipped bits, and also depends on the method to detect the multiple cell upsets [4].

Concerning the statistical methods, we will defina N_{AN} as the number of detected anomalies in the DV set. Thus, the number of expected false 2-bit multiple cell upsets is:

$$N_{F,2BMCU} = M \cdot N_{DV} \cdot \frac{N_{AN}}{L_N} \tag{2.5}$$

M being 1 for bitwise XOR, 2 for positive subtraction. N_{DV} the size of the DV set.

It is also possible to find expressions for the number of false 3-bit events but, unlike in the previous case, only upper and lower boundaries can be calculated. See [4] for a deeper discussion.

Sometimes, the researchers only checks the presence of multiple bit upsets in the bulk of experimental results. The accumulation of hits can lead to the existence of independent bitflips in the same word, which can be erroneously taken as an MBU. The expected number of false 2-bit MCUs is [4]:

$$N_{F,2BMBU} = N_{DV} \cdot \frac{W-1}{L_N} \tag{2.6}$$

W being the wordsize. Expressions for false 3-bit MCUs can be found in [4] and are implemented in LELAPE.

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"*: 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 [5]. 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 tool 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:`

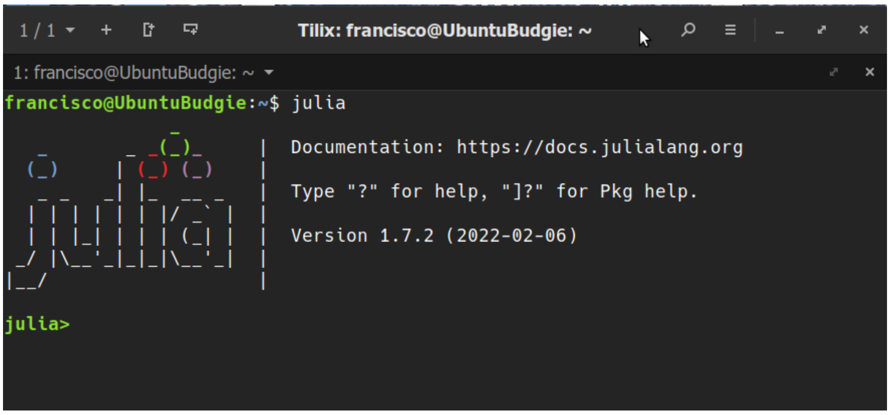
- **Visual Studio Code**: Popular IDE developed by Microsoft with plugins for Julia. It can be downloaded from https://code.visualstudio.com/ and the plugins installed as an extension. See https://code.visualstudio.com/docs/languages/julia for further information.

- **Atom**: Just like VS Code, it is a general-purpose IDE with plugins for Julia. It can be downloaded from https://atom.io/, with Julia plugins on https://junolab.org/. Thus, Atom becomes Juno, an IDE for Julia.

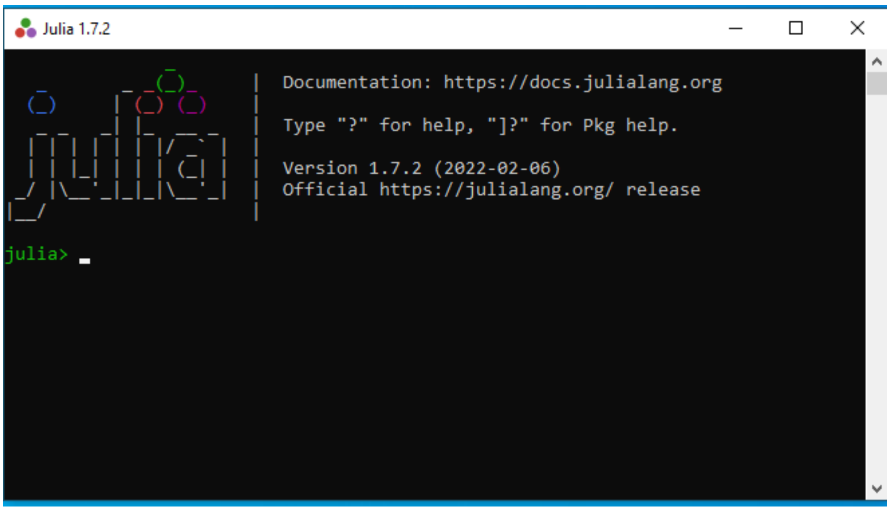
- **Jupyter**: Although it is typically used for Python, it is also appropriate for Julia. Indeed, Jupyter is an acronym for **Ju**lia-**Pyth**on-**R**. There are different ways of installing Jupyter. In systems with Microsoft Windows OS, the most simple way is to install Anaconda https://www.anaconda.com/products/individual. It can be setup to use Julia 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.

  Modern versions of Julia allow skipping this step, as we will see later. If not Jupyter is not found, Julia downloads and locally installs the software.

- **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.



(a)



(b)

Figure 3.1: Example of the REPL's welcome screen for Julia on a machine running Ubuntu Budgie (a) and Microsoft Windows (b).

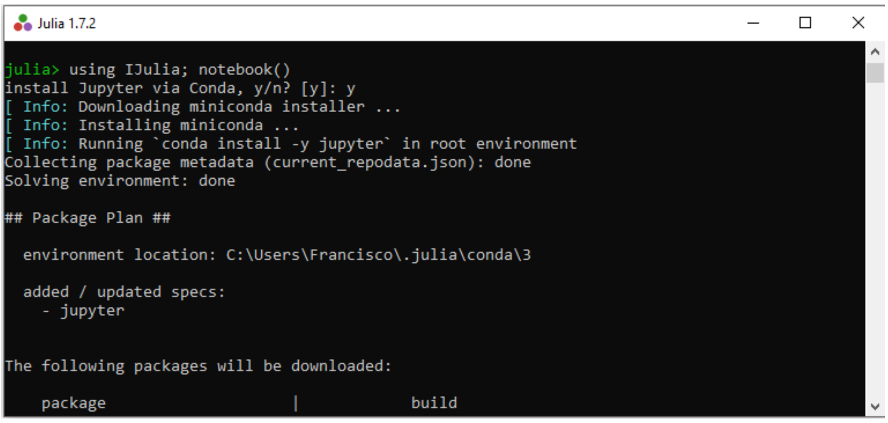


Figure 3.2: If Jupyter is not detected, Julia installs a minimal version the first time IJulia is used.

Atom, Jupyter and Pluto require the installation of additional packages to link Julia to the IDE. 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 package IJulia to the basic Julia installation. However, a previous step is to search for Jupyter on your computer. In the case of not finding it, Julia suggests to download a minimal version of Conda, thus installing Jupyter, as shown in Fig. 3.2. Finally, it is launched inside Julia as follows:

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

Pluto package is installed following a similar procedure.

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

and launched with:

```
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. Perhaps this flaw can be fixed in the future, but, nowadays, the tool is not as powerful as the others.

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`. The instruction is `git clone https://github.com/fjfrancopelaez/LELAPE.git`
2. Find the `LELAPE/src` folder where a file called LELAPE.jl is located.

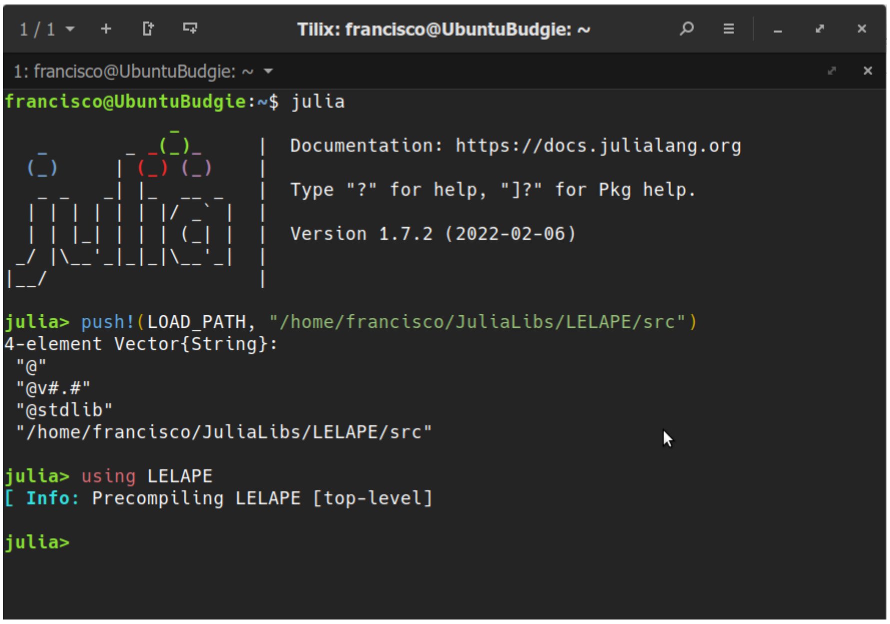


Figure 3.3: How to indicate Julia where LELAPE is installed, and how to load it.

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. `LOAD_PATH` is a string vector that contains the list of folder where Julia must look up external libraries. `push!` is a function that adds a new element at the end of any vector, keeping the name. Therefore, we have just added a new entry to the original list. Fig. 3.3 is a snapshot of the Julia terminal in GNU/Linux.

   A warning for users of Microsoft Windows: in this operating system, folders in the path are marked with the symbol \. For technical reasons, this is not recognized in Julia, so the path to LELAPE must be modified with one of the following tips:

   - Replacing \ with /, emulating the Unix style (GNU/Linux and Mac OS X).
   - Replacing \ with \\.

   Fig. 3.4 shows how to sucessfully add a new element to `LOAD_PATH` with the first option[1]

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

```
using LELAPE
```

   After a few seconds to precompile the library, the functions are loaded. Figs. 3.3 & 3.5 show practical examples.

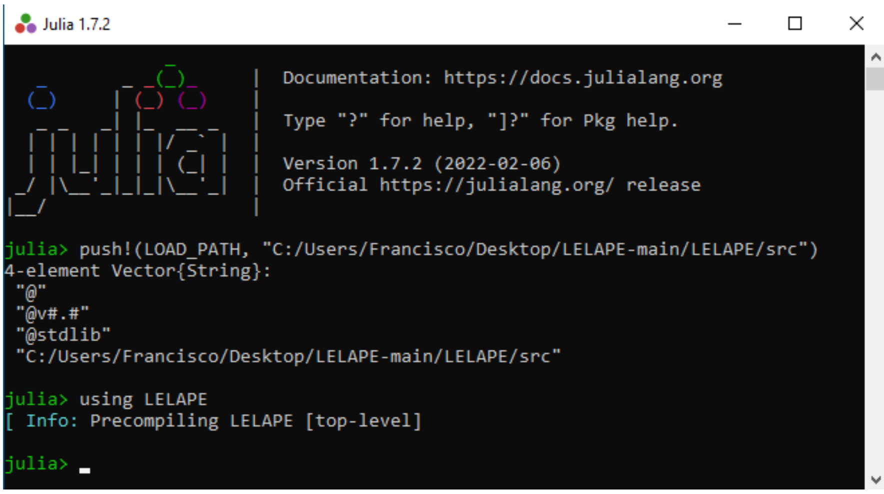


Figure 3.4: The folder containing LELAPE is added to the admitted paths in Microsoft Windows.

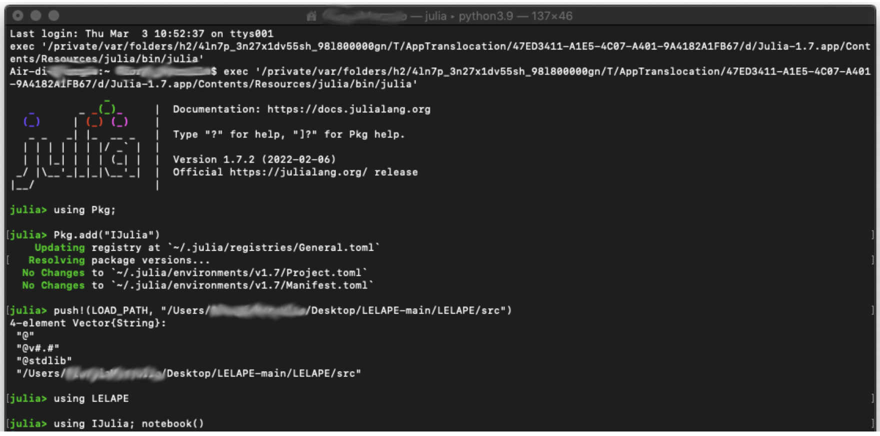


Figure 3.5: Executing Julia on macOS and loading LELAPE.

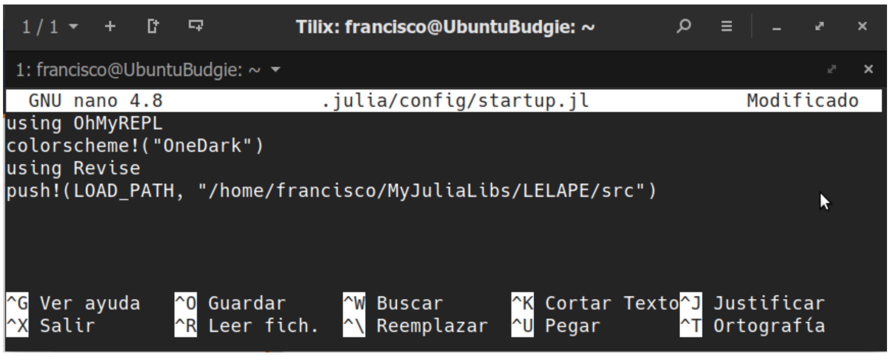


Figure 3.6: Example of startup.jl file in GNU/Linux.

Jupyter users should be aware of a detail. Even if it is loaded on the terminal, it does not inherit loaded packages or modules, so they must be loaded again and independently in Jupyter. Even more, the LOAD_PATH variable is initialized with its default value, with only three elements, so the instruction push!(LOAD_PATH, "PATH_TO_FOLDER") must be executed again before loading LELAPE.

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**: Inteligent highlighting of elements in REPL. Figs. 3.1 & 3.3 are using this package to show function names and strings.

Both are installed with Pkg.add().

Another interesting package to have is **DelimitedFiles**, which allows reading and writing CSV files. It is an essential package in Julia, available in a fresh installation, but not loaded by default. It is not necessary to fetch it and it is loaded as:

```
using DelimitedFiles
```

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

A tip for new users: after working with Julia for a long time, you may eventually discover that only a few packages are frequently used and you may get bored of loading them every time you start a new session. Thus, the typical solution is to create the following folder and text file in your home directory:

- In GNU/Linux & Mac OS X: /home/<USER>/.julia/config/startup.jl
- In Microsoft Windows: C:\Users\<USER>\.julia\config\startup.jl

Fig. 3.6 is an example of the startup.jl. In this file, one can see that LELAPE placement is automatically loaded when the session begins. However, if you wish to load these packages in Jupyter, it is necessary to create a new file, startup_ijulia.jl, with identical information. However, a soft link to startup.jl is enough.

[1]The instruction was push!(LOAD_PATH, "C:/Users/francisco/Desktop/LELAPE-main/LELAPE/src"), although push!(LOAD_PATH, "C:\\Users\\francisco\\Desktop\\LELAPE-main\\LELAPE \\src")

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 fly [6].
- **Dynamic**: A typical experiment consists in performing a continuous reading & writing process while the memory is irradiated with a predefined algorithm to detect failures [7, 8].

In any of these cases, the researcher saves information about the bitfilps: Word address, read content, initial pattern, ... 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.

  This calumn is also necessary for dynamic tests. In each step of the algorithm, the memory is fully explored and the discrepancies registered. So, in practice, for analysis with LELAPE the only differences between dynamic and psedostatic tests are that, in the former, the pattern changes from cycle to cycle, and that the addresses are recorded in different orders, sometimes increasing, sometimes decreasing, etc. These details are irrelevant for the tool.

LELAPE needs this elements to be converted to a UInt32 matrix[1] 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 `readdlm`, 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.

[1]In 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.

  If you had registered your data directly with the cell address, disregarding any organization in words, just set `WordWidth = 1` and `LA =` L_N, this being the memory size in bits.
- **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:

  $$PSA = NWA \cdot W + k \tag{4.1}$$

  This value is full of meaning in FPGAs since just returns the position of the cell in the bitflips. It is completly 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.

  Before going on, a little tip to treat data from **field tests** where a large number of similar devices are exposed to natural radiation. An option to analyze data would have been define a new pseudoaddress adding the position of the device in the bank, k_{MEM}, and the memory size in bits, L_N:

  $$PSA^* = k_{MEM} \cdot L_N + NWA \cdot W + k$$

  This may work, but is strongly computationally inefficient for LELAPE. Instead of it, we recommend to redefine the reading cycle index. Let us suppose that there are N_{MEM} identical memories in the bank, and that they are indexed from $k_{MEM} = 0$ to $k_{MEM} = N_{MEM} - 1$. If the bank reading cycle is k_{BNK}, the cycle to be included in the CSV file should be:

  $$Cycle = k_{BNK} \cdot N_{MEM} + k_{MEM} \tag{4.2}$$

  In other words, we are redefining cycles at device level, not bank level. This solution is much more efficient for LELAPE.

- **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, but, if you do not expect such a large event, reduce its value in order to unburden the computer memory usage.

4.3 Functions in the module

The functions depicted in this section are included in LELAPE and also accessible from the REPL, Jupyter, etc. Other functions are for internal use in LELAPE and are excluded from this list. The reader can just open the individual `.jl` files and check the comments.

Available functions are grouped in several categories:

- Preparation of experimental data
- Multiple Bit Upsets
- Statistical predictions
- Search of anomalies
- Classification of events from anomalies
- False events due to accumulation of bitflips

In all the items you will find the accepted input arguments, the kind of output as well as an explication of its purpose.

4.3.1 Preparation of experimental data

These functions just adapt the original test data to be used by LELAPE, or provide information about the data set. In this section, the following functions are included:

- ConvertToPseudoADD
- AddPatternColumn
- ExtractFlippedBits
- Npairs
- NTriplets

ConvertToPseudoADD

- **Input arguments**:
  - *Method 1:* **DATA**::Array{UInt32}, **WordWidth**::Int
  - *Method 2:* **DATA**::Array{UInt32}, **WordWidth**::Int, **KeepCycle**::Bool

  In the case of not providing **KeepCycle**, it is assumed to be `false`.

- **Output**: Array{UInt32, 2}, or Matrix{UInt32}.

- This function looks for the flipped bits between words in the same row but in the second and third columns of the **DATA** matrix. It does not matter if there are several bitflips, since they are independently counted. The pseudoaddress of each bitflip, defined as

  $$\text{WORDADDRESS} \times \text{Wordwith} + \text{Bitposition}$$

  (Eq. 2.1) is returned as the first column of the output.

  If there is information about the different cycles, it can be kept in the optional second column in the output with the condition of previously declaring **KeepCycle** as true. If cycle information is absent, the second column is filled with 1's.

  Fig. 4.1 shows an example of use in the REPL. There are more rows in the output matrix than in the input one since there are words with several flipped bits.

AddPatternColumn

- **Input arguments**:
  - *Method 1:* **DATA**::Matrix{UInt32}, **PATTERN**::UInt32
  - *Method 2:* **DATA**::Matrix{UInt32}, **PATTERN**::UInt16
  - *Method 3:* **DATA**::Matrix{UInt32}, **PATTERN**::UInt8

  DATA must have 2 or 3 columns, as explained later. If the matrix does not accomplish this condition, the function returns an error.

- **Output**: MatrixUInt32

- Sometimes, **DATA** are just a matrix with only two columns: *Word Address* & *Content*, assuming that the PATTERN is constant. This function just expands the matrix to include a column in the third position with the used **PATTERN**.

  Sometimes, in pseudostatic tests, there is a third column with cycle information. In this case, this column is shifted to the fourth position, and the void third column filled with the **PATTERN**.

  Fig. 4.2 shows an example of use of this function for a simple case.

ExtractFlippedBits

- **Input arguments**:
  - *Method 1:* **WORD**::UInt32, **PATTERN**::UInt32, **Wordwidth**::Int
  - *Method 2:* **WORD**::UInt16, **PATTERN**::UInt16, **Wordwidth**::Int
  - *Method 3:* **WORD**::UInt8, **PATTERN**::UInt8, **Wordwidth**::Int

- **Output**: :Array{Int,1}, or Vector{Int}

```
julia> using LELAPE

julia> DATA = [0x1234 0x44 0x55 1
               0x4567 0x75 0x55 1
               0x789A 0x05 0x55 2]
3×4 Matrix{Int64}:
  4660   68  85  1
 17767  117  85  1
 30874    5  85  2

julia> DATA = convert.(UInt32, DATA)
3×4 Matrix{UInt32}:
 0x00001234  0x00000044  0x00000055  0x00000001
 0x00004567  0x00000075  0x00000055  0x00000001
 0x0000789a  0x00000005  0x00000055  0x00000002

julia> WordWidth = 8; KeepCycle = true;

julia> ConvertToPseudoADD(DATA, WordWidth)
5×2 Matrix{UInt32}:
 0x000091a0  0x00000001
 0x000091a4  0x00000001
 0x00022b3d  0x00000001
 0x0003c4d4  0x00000001
 0x0003c4d6  0x00000001

julia> ConvertToPseudoADD(DATA, WordWidth, KeepCycle)
5×2 Matrix{UInt32}:
 0x000091a0  0x00000001
 0x000091a4  0x00000001
 0x00022b3d  0x00000001
 0x0003c4d4  0x00000002
 0x0003c4d6  0x00000002

julia> 
```

Figure 4.1: Example of use of `ConvertToPseudoADD`.

- This function allows discovering the position of different bits between **WORD** and **PATTERN**. It also verifies that both values are coherent with the **Wordwidth**, meaning that neither of them are higher than $2^{Wordwidth} - 1$. A vector, never larger than **Wordwidth** is returned. If this condition is not fulfilled, the functions returns an error. If **WORD** and **PATTERN** are equal, the output is a void vector.

  Fig. 4.3 shows some examples of use of this function. Take into account that, even when there is a single bitflip, a vector is always returned, albeit with an only value inside.

NPairs

- **Input arguments**:
  - *Method 1*: **DATA**::ArrayUInt32
  - *Method 2*: **DATA**::ArrayUInt32, **UsePseudoAdd**::Bool
  - *Method 3*: **DATA**:: ArrayUInt32, **UsePseudoAdd**::Bool, **WordWidth**:: Int
  - *Method 4*: **DATA**::ArrayUInt32, **UsePseudoAdd**:: Bool, **WordWidth**:: Int, **KeepCycle**:: Bool
  - *Method 5*: **N**::Int

```
julia> DATA
832×2 Matrix{UInt32}:
 0x00001ee6  0x00000057
 0x0000289f  0x00000057
           ⋮
 0x001feeef  0x000000d5
 0x001ff4ce  0x00000045

julia> PATTERN =0x55
0x55

julia> DATA2=AddPatternColumn(DATA, PATTERN)
832×3 Matrix{UInt32}:
 0x00001ee6  0x00000057  0x00000055
 0x0000289f  0x00000057  0x00000055
           ⋮
 0x001feeef  0x000000d5  0x00000055
 0x001ff4ce  0x00000045  0x00000055

julia> 
```

Figure 4.2: Example of use of `AddPatternColumn`.

```
julia> using LELAPE

julia> ExtractFlippedBits(0xFF, 0x0F, 8)
4-element Vector{Int64}:
 4
 5
 6
 7

julia> ExtractFlippedBits(0xFF, 0xFF, 8)
Int64[]

julia> ExtractFlippedBits(0xFF, 0xEF, 8)
1-element Vector{Int64}:
 4

julia> ExtractFlippedBits(0xFF, 0xEF, 4)
ERROR: WORD and/or PATTERN inputs are not representable with
the present wordwidth.
Stacktrace:
 [1] error(s::String)
```

Figure 4.3: Example of use of `ExtractFlippedBits`.

In Methods 1–4, default values for parameters are **`UsePseudoAdd`** `= false`, **`WordWidth`** `= 1`, **`KeepCycle`** `= false`.

- **Output**: Int

- **DATA** is a 3 or 4-column matrix derived from the loaded CSV file and each row containing the word address (#1), the read word after the tests (#2), the initial pattern (#3) and the number of reading cycle when the error was observed. If this last column is not provided or **KeepCycle** is false, the system works as if only one cycle was done.

  The function provides the number of pairs of addresses taken during each cycle regarding predictions of the Only-SBU model (Eq. 2.4). For example, let us suppose that we have done 2 cycles, observing in the first one 30 events, and 40 in the second. The number of possible pairs is the addition of the pairs in each cycle:

$$\frac{1}{2} \cdot 30 \cdot (30 - 1) + \frac{1}{2} \cdot 40 \cdot (40 - 1) = 1215$$

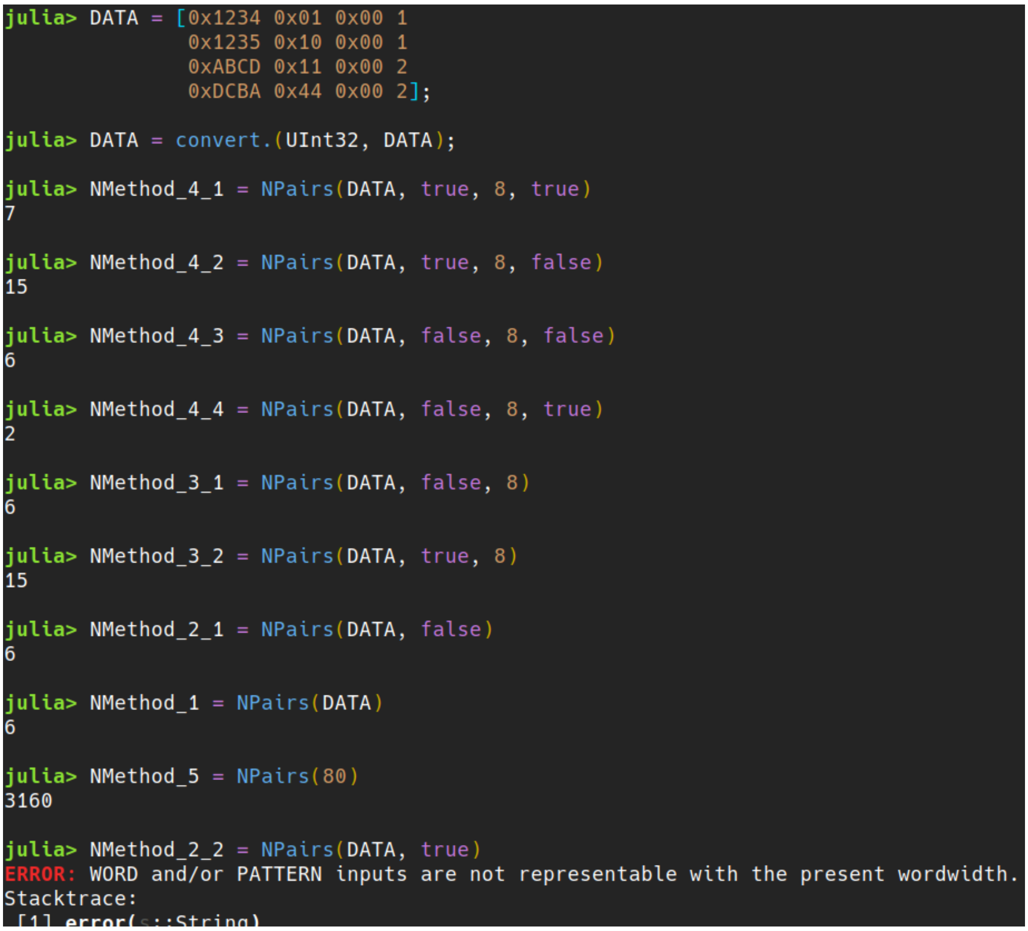
Thus, 1215 pairs can be formed. If we had not taken into account the existence of cycles, the number of pairs would have been

$$\frac{1}{2} \cdot (30 + 40) \cdot (30 + 40 - 1) = 2415$$

however, many of them are unreal since were taken in different times!

If **UsePseudoAdd** is set to true, the system looks for the position of the bitflips inside the word and uses the pseudoaddress (WordAddress $\times$ WordWidth $+$ Position). Thus, it is necessary to provide the **WordWidth** value (8, 16, 32, ...). Using **DATA** as the only argument is appropriate to analyze values directly in pseudoaddress format (or just the word address) taken during one only cycle. This is the case, for example, of FPGAs configuration memory.

There is a final method, just saying the number of observed pairs, **N**. In this case, the function returns **N**$\cdot$(**N**-1)/2.



```
julia> DATA = [0x1234 0x01 0x00 1
               0x1235 0x10 0x00 1
               0xABCD 0x11 0x00 2
               0xDCBA 0x44 0x00 2];

julia> DATA = convert.(UInt32, DATA);

julia> NMethod_4_1 = NPairs(DATA, true, 8, true)
7

julia> NMethod_4_2 = NPairs(DATA, true, 8, false)
15

julia> NMethod_4_3 = NPairs(DATA, false, 8, false)
6

julia> NMethod_4_4 = NPairs(DATA, false, 8, true)
2

julia> NMethod_3_1 = NPairs(DATA, false, 8)
6

julia> NMethod_3_2 = NPairs(DATA, true, 8)
15

julia> NMethod_2_1 = NPairs(DATA, false)
6

julia> NMethod_1 = NPairs(DATA)
6

julia> NMethod_5 = NPairs(80)
3160

julia> NMethod_2_2 = NPairs(DATA, true)
ERROR: WORD and/or PATTERN inputs are not representable with the present wordwidth.
Stacktrace:
 [1] error(s::String)
```

Figure 4.4: Example of use of `NPairs`.

Fig. 4.4 shows an example of use of this function.

NTriplets

- **Input arguments**:

- *Method 1*: **DATA**::ArrayUInt32
- *Method 2*: **DATA**::ArrayUInt32, **UsePseudoAdd**::Bool
- *Method 3*: **DATA**:: ArrayUInt32, **UsePseudoAdd**::Bool, **WordWidth**:: Int
- *Method 4*: **DATA**::ArrayUInt32, **UsePseudoAdd**:: Bool, **WordWidth**:: Int, **Keep-Cycle**:: Bool
- *Method 5*: **N**::Int

- **Output**: Int

- Similar to *Npairs(...)*, but calculating the expected number of triplets instead of pairs. Thus, instead of using Eq. 2.4 as the basis for calculations, this function uses:

$$N_{Triplets} = \frac{1}{6} \cdot N_{BF} \cdot (N_{BF} - 1) \cdot (N_{BF} - 2)$$

4.3.2 Multiple Bit Upsets

Sometimes, the researcher just wants to know how many multiple bit upsets occurred during the experiments. There is only one function in this section, `CheckMBUs`.

CheckMBUs

- **Input arguments**:
  - *Method 1*: **WORD**::UInt32, **PATTERN**::UInt32, **WordWidth**::Int
  - *Method 2*: **WORDS**::Vector{UInt32}, **PATTERN**::Vector{UInt32}, **WordWidth**::Int
  - *Method 3*: **WORDS**::Vector{UInt32}, **PATTERN**::UInt32, **WordWidth**::Int

- **Output**:
  - *Method 1*: Tuple{Int, Vector{Int}}
  - *Methods 2 & 3*: Tuple{Vector{Int}, Vector{Any}}

- The first method is quite easy to understad. It just takes to unsigned integer 32-bit numbers, **WORD** & **PATTERN**, of which only the last **WordWidth** bits are significant, and looks for equivalent bits with different value with the function `ExtractFlippedBits`. Then, it returns the number of biflips and a vector containing the flipped positions. Fig. 4.5 shows how this function behaves with these inputs.

```
julia> WORD = UInt32(0b0011_0000_1100_0000); PATTERN = UInt32(0x0000); Width=16;

julia> N, Positions = CheckMBUs(WORD, PATTERN, Width)
(4, [6, 7, 12, 13])

julia> N
4

julia> Positions
4-element Vector{Int64}:
  6
  7
 12
 13
```

Figure 4.5: Example of use of `CheckMBUs` with unsigned integers as inputs.

For Method 2, the idea is to provide as arguments two similar-length vectors with the **WORD** and **PATTERN** values in UInt32 format, as well as the **WordWidth**. Then, the function checks the values in both vectors with the same index and returns two vectors.

The first one is a typical vector of integers, containing in the k-th position the number of different bits between **WORD**[k] y **PATTERN**[k]. The second is a vector of vectors. Thus, the element in the k-th positiion is also a vector with the positions of the flipped bits.

Method 3 is quite similar to Method 2, but **PATTERN** is no longer a vector but a constant value for all the elements of **WORDS**. Fig. 4.6 shows how this method and the previous one work.

```
julia> DATA = [0x1234 0x01 0x00 1
               0x1235 0x10 0x00 1
               0xABCD 0x11 0x00 2
               0xDCBA 0x44 0x00 2];

julia> DATA = convert.(UInt32, DATA);

julia> N, Positions = CheckMBUs(DATA[:,2], DATA[:,3], 8)
([1, 1, 2, 2], Any[[0], [4], [0, 4], [2, 6]])

julia> N
4-element Vector{Int64}:
 1
 1
 2
 2

julia> Positions
4-element Vector{Any}:
 [0]
 [4]
 [0, 4]
 [2, 6]

julia> N, Positions = CheckMBUs(DATA[:,2], UInt32(0x00), 8)
([1, 1, 2, 2], Any[[0], [4], [0, 4], [2, 6]])
```

Figure 4.6: Example of use of `CheckMBUs` with vectors as inputs. Last line uses Method 3, where the pattern is provided as an unsigned integer.

4.3.3 Statistical predictions

The goals of this set of functions is to cast predictions about the characteristics of the data according to the Only-SBU model. The following functions are included in this section:

- MaxExpectedRepetitions
- TheoAbundance_POS
- TheoAbundance_XOR
- TheoAbundance

Related to these functions are those used to determine the expected number of false errors, but they will be depicted in the corresponding section.

MaxExpectedRepetitions

- **Input arguments**:

  - *Method 1*: **NDV**::Int, **LN**::Int, **Operation**:: String, ε:: AbstractFloat
  - *Method 2*: **NDV**::Int, **LN**::Int, **Operation**::String

- **Output**: Int

- The purpose of this funcion is to determine the maximum number of expected repetitions. in a DV set taken from a memory with size equal to **LN** . In general, it is the first integer such that its theoretical abundance is lower than ε. If this threshold is not provided, it is assumed to be $\varepsilon = 0.01$.

TheoAbundance_POS

- **Input arguments**:
  - *Method 1*: **NR**: :Int, **NB**::Int, **LN**::Int, **UsingDV**::Bool
  - *Method 2*: **NR**: :Int, **NB**::Int, **LN**::Int

- **Output:** AbstractFloat

- This function allows calculating the expected number of values repeated **NR** times after several bitflips in a memory with **LA** words with **W** bits per word supposing to have used the POSITIVE subtraction.

  NR must be an integer number higher than or equal to 0. **NB** is an integer number supposed to be higher than 1. **LN** is an integer number, and indicates the size of the memory. If the researcher uses bit address for calculations, **LN** is **LA**$\times$**W**. However, if he/she uses the word address instead, this parameerter is **LA**.

  UsingDV determines how **NB** must be interpreted. If that boolean variable is `true`, **NB** is the size of the DV set, called elsewhere N_{DV}. If false, **NB** is the number of bitflips and N_{DV} must be calculated from it. By default, **UsingDV** is `false`.

  Equations were got from Eq.2 of the Appendix in J. C. Fabero et al., ”*Single Event Upsets Under 14-MeV Neutrons in a 28-nm SRAM-Based FPGA in Static Mode,*” in IEEE Transactions on Nuclear Science, vol. 67, no. 7, pp. 1461-1469, July 2020, doi: 10.1109/TNS.2020.2977874.

  Lawfully avalaible for free download on https://eprints.ucm.es/id/eprint/59496/

TheoAbundance_XOR

- **Arguments:**
  - *Method 1*: **NR**: :Int, **NB**::Int, **LN**::Int, **UsingDV**::Bool
  - *Method 2*: **NR**: :Int, **NB**::Int, **LN**::Int

- **Output:** AbstractFloat

- Equivalent to `TheoAbundance_POS`, but referred to the bitwise XOR operation.

  Equations were got from Eq.12 of the Appendix.C in F. J. Franco et al., ”*Statistical Deviations From the Theoretical Only-SBU Model to Estimate MCU Rates in SRAMs,*” in IEEE Transactions on Nuclear Science, vol. 64, no. 8, pp. 2152-2160, Aug. 2017, doi: 10.1109/TNS.2017.2726938.

  Lawfully avalaible for free on https://eprints.ucm.es/id/eprint/43874/

TheoAbundance

- **Input arguments**:
  - *Method 1:* **NR**::Int, **NB**::Int, **LN**::Int, **Operation**:: String, **UsingDV**::Bool
  - *Method 1:* **NR**::Int, **NB**::Int, **LN**::Int, **Operation**:: String
- **Output**: AbstractFloat
- This is an Alias for `TheoAbundance_POS()` or `TheoAbundance_XOR()`. The definition of arguments are similar, with an additional parameter, **Operation**, which only can be `"XOR"` or `"POS"`.

4.3.4 Search of anomalies

This is an important set of functions that determine the statistical anomalies in the data set, and discard those that are not trustworthy, perhaps due to interaction between events.

The functions included in this section are:

- DetectAnomalies_SelfConsis
- DetectAnomalies_Shuffle_Rule
- DetectAnomalies_Trace_Rule
- DetectAnomalies_MCU_Rule
- DetectAnomalies_FullCheck

Last function just calls the previous four.

DetectAnomalies_SelfConsis

- **Input arguments**:
  - *Method 1:* **DATA**::Array{UInt32, 2}, **WordWidth**::Int, **LN0**::Int, **Operation**::String, **UsePseudoADD**::Bool, **KeepCycle**::Bool, ε::AbstractFloat, **LargestMCUSize**::Int
  - *Method 2:* **DATA**::Array{UInt32, 2}, **WordWidth**::Int, **LN0**::Int, **Operation**::String, **UsePseudoADD**::Bool, **KeepCycle**::Bool, ε::AbstractFloat
  - *Method 3:* **DATA**::ArrayUInt32, 2, **WordWidth**::Int, **LN0**::Int, **Operation**::String, **UsePseudoADD**::Bool, **KeepCycle**::Bool

  Default values for ε & **LargestMCUSize** are `0.05` and `200` respectively.
- **Output:** Array{UInt32, 2}
- This function will calculate the anomalies in the set of addresses using the Self Consistency rule. First of all, let us know the inputs:
  - **DATA**: A matrix with 3 or 4 columns.
    * The first column contains the word addresses in UInt32 format.
    * The second one shows the content read in the memory after the irradiation.
    * The third one, the pattern that should be inside.
    * The fourth one is optional and shows the number of the read cycle if the memory was read and corrected several times during the irradiation.
  - **WordWidth**: The size of each word in bits, usually 8, 16. 32, etc. No default value is provided.

– **LN0**: The memory size in words (not in bits!!!). In many cases, a natural power of 2.
– **Operation**: A string variable to indicate the preferred operation to calculate the DV set. Only two operations are allowed:
  * `"XOR"`: bitwise XOR.
  * `"POS"`: positive subtraction
– **UsePseudoADD**: A boolean variable. Its purpose is to indicate that the user wants to use the word addresses when this parameter is false). If true, the pseudoaddress, calculated as WORADDRESS $\times$ WordWidth $+$ BitPosition, is used instead. Full of sense in FPGA since it is just the position of the bit in the bitstream, it has not physical interpretation in memories BUT works!!!!
– **KeepCycle**: If true, the function looks for the fourth column and uses it to calculate the DV Set.
– ε: A small positive integer number to determine the threshold that defines when a number of repetitions are impossible to occur. Set by default to 0.05 if not provided among the input arguments.
– **LargestMCUSize**: This value indicates the largest possible size for MCUs. It has not physical sense and is only used to stop the program if unrealistic events occur. Set to 200 if not given as an input.

The function returns an $N \times 2$ UInt32 matrix. The first column contains the anomalously repeated values of the DV SET compatible with the Self Consistency test. The second one contains the number of times they appear in the DV set. Due to format integrity reasons, this column is expressed in unnatural `UInt32` format.

It is advisable a latter conversion into Int to make this column more readable. There are several examples of this function or equivalent in the Jupyter folder.

If the function does not find any anomaly, it returns a void matrix. Or, more exactly, a 0×2 one.

DetectAnomalies_Shuffle_Rule

TBD

DetectAnomalies_Trace_Rule

TBD

DetectAnomalies_MCU_Rule

TBD

DetectAnomalies_FullCheck

TBD

4.3.5 Classification of events from anomalies

MCU_Indexes

- **Input arguments**:

- *Method 1:* **DATA**::Matrix{UInt32}, **OPERATION**::String, **Markers**:: Vector{UInt32}, **UsePseudoADD**::Bool, **WordWidth**::Int, **LimitMCUSize**:: Int
- *Method 2:* **DATA**::Matrix{UInt32}, **OPERATION**::String, **Markers**:: Vector{UInt32}, **UsePseudoADD**::Bool, **WordWidth**::Int
- *Method 3:* **DATA**:Matrix{UInt32}, **OPERATION**::String, **Markers**:: Vector{UInt32}, **UsePseudoADD**::Bool
- *Method 4:* **DATA**::Matrix{UInt32}, **OPERATION**::String, **Markers**:: Vector{UInt32},

* **Output**: Matrix{Int}
* This functions uses the **DATA** set to look for pairs of addresses which treated with **OPERATION** yield one of the **MARKERS**. If an **ADDRESS** is related to other two addresses, a 3–bit MCU appears (and so on.) The rest of parameters are used to provide necessary information to use the **PSEUDOADDRESS** instead of the **WORD ADDRESS**.

  More information about the inputs:

  - DATA: A matrix with 3 or 4 columns.
    * The first one contains the word addresses in UInt32 formata.
    * The second one shows the content read in the memory after the irradiation.
    * The third one, the pattern that should be inside.
    * The fourth one is optional and shows the namber of the read cycle if the memory was read and corrected several times during the irradiation.
  - **OPERATION**: A string variable to indicate the preferred operation to calculate the DVSET. Only two operations are allowed:
    * "XOR": XORing bit to bit.
    * "POS": abs(a-b)
  - **UsePseudoADD**: A boolean variable. It allows to indicate that the user wants to user word addresses (false). If true, a pseudoaddress is assigned to each bit and calculated as

    $$\text{WORDADDRESS} \times \text{WordWidth} + \text{BitPosition}.$$

    Full of sense in FPGA since it is just the position of the bit in the bitstream, it has not physical interpretation in memories BUT works!!!!
  - **WordWidth**: The size of each word in bits, usually 8, 16. 32, etc.
  - **LargestMCUSize**: This value indicates the largest possible size for MCUs. It has not physical sense and is only used to stop the program if unreallistic events occur. Initially set to 200.

  Concerning the OUTPUT: It provides an integer NMCU$\times$LMCU matrix, NMCU being the number of detected MCUs and LMCU the size of the largest reconstructed MCU. Every value different than 0 must be determined as follows:

  1. UsePseudoADD = false: The index indicates the row in DATA with the address in the MCU.
  2. UsePseudoADD = true: It provides the index in the PSEUDOADDRESS derived SET. If the exact position of the bitcell is required, DATA should be treated with ConvertToPseudoADD() and the index used in the resulting matrix.

  In both cases, if the size of the MCU is smaller than LMCU, the row will be filled with zeros until reaching the desired length. For example, if the content of a row is [5 7 9 0 0], it must be interpreted as a 3-bit MCU involving addresses indexed with 5, 7 & 9 in an experiment in which at least a 5-bit MCU (and nothing larger) was observed.

  Finally, if the index of an address does not appear in the returned matrix, it should be interpreted as isolated and belonging to an SBU.

Classify_Addresses_in_MCU

- **Input arguments**:
  - *Method 1*: **DATA**:: Matrix{UInt32}, **Indexes**:: Matrix{Int}, **UsePseudoADD**:: Bool, **WordWidth**:: Int
  - *Method 2*: **DATA**:: Matrix{UInt32}, **Indexes**:: Matrix{Int}, **UsePseudoADD**:: Bool
  - *Method 3*: **DATA**:: Matrix{UInt32}, **Indexes**:: Matrix{Int}
- **Output**: Vector:: {Any}
- The purpose of this function is to classify the addresses (or pseudoaddresses) with bitflips transform the matrix of INDEXES got from MCU_Indexes() into a Vector of matrices, called SOLUTION, which is eventually returned as OUTPUT.

  The length of SOLUTION is the size of the largest observed MCU(s), NLMCU. Thus, SOLUTION[1] is a N x NLMCU matrix in which every row contains the addresses or pseudoaddresses of the NLMCU bitflips involved in this MCU. N is the number of observdd NLMCU-bit MCUs.

  SOLUTION[2] is devoted to events with M = NLMCU-1 bits. As before, it is a matrix with NLMCU-1 rows and an undetermined number of rows.

  Finally, SOLUTION[NLMCU] is just a simple vector with the addresses not involved in MCUs. Obviously, these are the SBUs.

4.3.6 False events due to accumulation of bitflips

When the number of bitflips is too high, it is possible that the interpretation of results can be affected by random phenomena, such as the disappearance of bitflips if a cell is hit twice, single bit upsets in adjacent cells that are misled with multiple events, etc.

The functions are:

- CorrectNBitFlips
- NF2BitMCUs
- NF3BitMCUs

CorrectNBitFlips

- **Input arguments**: **NBF**::Int, **LN**::Int
- **Output**: Float64
- This function tries to correct the number of bitflips to compensate cells hit twice that escape from inspection. It is just an implementation of the simple Eq. 2.2.
  - **NBF**: Number of bitflips. Theoretically, SBUs but they are impossible to be distinguished from other kinds of bitflips. Therefore, this simple approach is taken.
  - **LN**: Memory size in BITS!!!!!

  Fig. 4.7 is an example of use of this function.

  Expression is taken from Eq. 6 in F. J. Franco, J. A. Clemente, H. Mecha and R. Velazco, "*Influence of Randomness During the Interpretation of Results From Single-Event Experiments on SRAMs*," IEEE Transactions on Device and Materials Reliability, vol. 19, no. 1, pp. 104-111, March 2019, doi: 10.1109/TDMR.2018. 2886358.

```
julia> LN = 2^20; NBF = 2000;

julia> ActNBF = CorrectNBitFlips(NBF, LN)
2003.814697265625

julia> println("In a memory with $LN bits and where $NBF bitflips were registered, probably around $ActNBF bitflips actually occurred")
In a memory with 1048576 bits and where 2000 bitflips were registered, probably around
 2003.814697265625 bitflips actually occurred
```

Figure 4.7: Example of use of `CorrectNBitFlips`.

NF2BitMCUs

- **Input arguments**:
  - *Method 1*: **NSBU**::Int, **LA**::Int, **METHOD**::String, **D**::Int, **WordWidth**::Int, **UsePseudoAddress**::Bool
  - *Method 2*: **NSBU**::Int, **LA**::Int, **METHOD**::String, **D**::Int, **WordWidth**::Int
- **Output**: Float64
- It indicates the expected number of false 2-bit MCUs that will occur in a memory with **LA** words with **WORDWIDTH** bits each in which **NSBU** SBUs have occurred. In this analysis, MCUS are sought using some grouping method (**METHOD**) with a generalized distance **D**.

  If **UsePseudoAddress** is not provided, its default value is `false`.

  Unlike `NF3BitMCUs`, two values are provided as outputs, called "*optimistic*" and "*pessimistic*". The actual number of expected false events is somewhere between both values. So far, it is not possible to get more accurate value, as explained in the theoretical development. Use the values as you may wish.

  Admitted values for **METHOD** and **D** are the following:

  1. **METHOD**: `"MBU"` $\rightarrow$ Only MBUs are sought. In this case, **D** is the **WORDWiDTH**.
  2. **METHOD**: `"MHD"` $\rightarrow$ Only possible if the user has been able to place the flipped cell in the XY plane. Two cells are related if $|x_1 - x_2| + |y_1 - y_2|) \leq D$. This is the "*Manhattan distance*".
  3. **METHOD**: `"IND"` $\rightarrow$ Only possible if the user has been able to place the flipped cell in the XY plane. Two cells are related if max$(|x_1 - x_2|, |y_1 - y_2|) \leq D$. In mathematics, this is the *"infinite distance"*.
  4. **METHOD**: `"THD"` $\rightarrow$ Only valid if pairs of bitflips are located in a linear bitstream and if the distance between cells is smaller than **D**: $|x_1 - x_2| \leq D$.
  5. **METHOD**: `"XOR"`$\rightarrow$ Related pairs are got by means of statistical deviations. Addresses are XORed and only if the value is one of the **D** possible critical values. If the WORD Addresses is used instead of PSEUDOADDRESS, the memory size must be expressed in WORDs, **LA = LN/WordWidth**. IF SO, THE WORDWIDTH MUST BE PROVIDED.
  6. **METHOD**: `"POS"`$\rightarrow$ Identical to the previous one but with positive subtraction instead of XOR.

  For LELAPE, only the two last methods are of interest. However, the other methods are included in the tool in case you have used another strategy to combine bitflips and wish to know the background noise. Fig. 4.8 is an example of use.

  Everything can be found in Eq. 11 of F. J. Franco, J. A. Clemente, G. Korkian, J. C. Fabero, H. Mecha and R. Velazco, *"Inherent Uncertainty in the Determination of Multiple Event*

```
julia> LA = 2^17; WordWidth = 8; UsePseudoAddress = true; NCriticalValues = 5;

julia> NSBU = 1000; NMU2 = 100;

julia> NF2MCU_A=NF2BitMCUs(NSBU, LA, "XOR", NCriticalValues, WordWidth, UsePseudoAddress)
2.3818016052246094

julia> NF2MCU_B=NF2BitMCUs(NSBU, LA, "POS", NCriticalValues, WordWidth, UsePseudoAddress)
4.763603210449219

julia> NF2MCU_C=NF2BitMCUs(NSBU, LA, "MBU", WordWidth, WordWidth, UsePseudoAddress)
3.334522247314453

julia> print("In a memory with $LA x $WordWidth bits, $NSBU SBUs and $NMU2 2-bit MCUs, with $NCritic
alValues anomalies, $NF2MCU_A false 2-bit MCUs are expected with bitwise XOR, $NF2MCU_B with positiv
e subtraction, and $NF2MCU_C false 2-bit MBUs.")
In a memory with 131072 x 8 bits, 1000 SBUs and 100 2-bit MCUs, with 5 anomalies, 2.381801605224609
 false 2-bit MCUs are expected with bitwise XOR, 4.763603210449219 with positive subtraction, and 3.
334522247314453 false 2-bit MBUs.
```

Figure 4.8: Example of use of `NF2BitMCUs`.

Cross Sections in Radiation Tests," IEEE Transactions on Nuclear Science, vol. 67, no. 7, pp. 1547-1554, July 2020, doi: 10.1109/TNS.2020.2977698.

NF3BitMCUs

- **Input arguments**:
  - *Method 1*: **NSBU**::Int, **NMU2**::Int, **LN**::Int, **METHOD**::String, **D**::Int, WordWidth::Int
  - *Method 2*: **NSBU**::Int, **NMU2**::Int, **LN**::Int, **METHOD**::String, **D**::Int
- **Output**: TupleFloat64, Float64
- It indicates the expected number of false 3-bit MCUs that will occur in a memory with **LN** bits in which **NSBU** SBUs and **NMU2** 2-bit MCUs have occurred. In this analysis, MCUS are sought using some grouping method (**METHOD**) with a generalized distance D.

  MCUS are sought using some grouping method (**METHOD**) with a generalized distance **D**.

  Admitted values for **METHOD** and **D** are the following:

  1. **METHOD**: `"MBU"` $\rightarrow$ Only MBUs are sought. In this case, **D** is the **WORDWiDTH**.
  2. **METHOD**: `"MHD"` $\rightarrow$ Only possible if the user has been able to place the flipped cell in the XY plane. Two cells are related if $|x_1 - x_2| + |y_1 - y_2|) \leq D$. This is the *"Manhattan distance"*.
  3. **METHOD**: `"IND"` $\rightarrow$ Only possible if the user has been able to place the flipped cell in the XY plane. Two cells are related if max$(|x_1 - x_2|, |y_1 - y_2|) \leq D$. In mathematics, this is the *"infinite distance"*.
  4. **METHOD**: `"THD"` $\rightarrow$ Only valid if pairs of bitflips are located in a linear bitstream and if the distance between cells is smaller than **D**: $|x_1 - x_2| \leq D$.
  5. **METHOD**: `"XOR"`$\rightarrow$ Related pairs are got by means of statistical deviations. Addresses are XORed and only if the value is one of the **D** possible critical values. If the WORD Addresses is used instead of PSEUDOADDRESS, the memory size must be expressed in WORDs, **LA = LN/WordWidth**. IF SO, THE WORDWIDTH MUST BE PROVIDED.
  6. **METHOD**: `"POs"`$\rightarrow$ Identical to the previous one but with positive subtraction instead of XOR.

  For LELAPE, only the two last methods are of interest. See Fig. 4.9 for a practical example. However, the other methods are included in the tool in case you have used another strategy to combine bitflips and wish to know the background noise.

```
julia> LA = 2^17; WordWidth = 8; UsePseudoAddress = true; NCriticalValues = 5;

julia> NSBU = 1000; NMU2 = 100;

julia> NF3MCU_A_OPT, NF3MCU_A_PES = NF3BitMCUs(NSBU, NMU2, LA*WordWidth, "XOR",
NCriticalValues, WordWidth)
(3.2452016603201628, 6.683847168460488)

julia> NF3MCU_B_OPT, NF3MCU_B_PES = NF3BitMCUs(NSBU, NMU2, LA*WordWidth, "POS",
NCriticalValues, WordWidth)
(7.7369523933157325, 16.344402101822197)

julia> print("In a memory with $LA x $WordWidth bits, $NSBU SBUs and $NMU2 2-bit
 MCUs and $NCriticalValues anomalies, it is expected to find between $NF3MCU_A_O
PT and $NF3MCU_A_PES false 3-bit MCUs with bitwise XOR, and between $NF3MCU_B_OP
T and $NF3MCU_B_PES with positive subtractions.")
In a memory with 131072 x 8 bits, 1000 SBUs and 100 2-bit MCUs and 5 anomalies,
it is expected to find between 3.2452016603201628 and 6.683847168460488 false 3-
bit MCUs with bitwise XOR, and between 7.7369523933157325 and 16.344402101822197
 with positive subtractions.
julia>
```

Figure 4.9: Example of use of `NF3BitMCUs`.

Everything can be found in Eq. 11 of F. J. Franco, J. A. Clemente, G. Korkian, J. C. Fabero, H. Mecha and R. Velazco, *"Inherent Uncertainty in the Determination of Multiple Event Cross Sections in Radiation Tests,"* IEEE Transactions on Nuclear Science, vol. 67, no. 7, pp. 1547-1554, July 2020, doi: 10.1109/TNS.2020.2977698.

In this paper, it was demonstrated that it is mathematically impossible to get an exact value. Therefore, optimistic and pessimistic results are provided.

Bibliography

[1] F. J. Franco *et al.*, "Statistical Deviations From the Theoretical Only-SBU Model to Estimate MCU Rates in SRAMs," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2152–2160, 2017.

[2] J. C. Fabero *et al.*, "Single Event Upsets Under 14-MeV Neutrons in a 28-nm SRAM-Based FPGA in Static Mode," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1461–1469, 2020.

[3] H. J. Tausch, "Simplified Birthday Statistics and Hamming EDAC," *IEEE Transactions on Nuclear Science*, vol. 56, no. 2, pp. 474–478, 2009.

[4] F. J. Franco, J. A. Clemente, G. Korkian, J. C. Fabero, H. Mecha, and R. Velazco, "Inherent Uncertainty in the Determination of Multiple Event Cross Sections in Radiation Tests," *IEEE Transactions on Nuclear Science*, vol. 67, no. 7, pp. 1547–1554, 2020.

[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[6] V. Gupta, "Analysis of single event radiation effects and fault mechanisms in SRAM, FRAM and NAND Flash : application to the MTCube nanosatellite project.," Master's thesis, Université de Montpellier, 2017. Electronics. English. ffNNT : 2017MONTS087ff. fftel-01954572f. Available on https://tel.archives-ouvertes.fr/tel-01954572.

[7] G. Tsiligiannis, L. Dilillo, A. Bosio, P. Girard, S. Pravossoudovitch, A. Todri, A. Virazel, H. Puchner, C. Frost, F. Wrobel, and F. Saigné, "Multiple Cell Upset Classification in Commercial SRAMs," *IEEE Transactions on Nuclear Science*, vol. 61, no. 4, pp. 1747–1754, 2014.

[8] G. Tsiligiannis, L. Dilillo, V. Gupta, A. Bosio, P. Girard, A. Virazel, H. Puchner, A. Bosser, A. Javanainen, A. Virtanen, C. Frost, F. Wrobel, L. Dusseau, and F. Saigné, "Dynamic Test Methods for COTS SRAMs," *IEEE Transactions on Nuclear Science*, vol. 61, no. 6, pp. 3095–3102, 2014.