Assessing and Restoring Reproducibility of Jupyter Notebooks

Jiawei Wang
Faculty of Information Technology, Monash University
Melbourne, Australia
Jiawei.wang1@monash.edu

Tzu-yang KUO*
Hong Kong University of Science and Technology
Hong Kong, China
tkuo@connect.ust.hk

Li Li†
Faculty of Information Technology, Monash University
Melbourne, Australia
li.li@monash.edu

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.saarland

ABSTRACT

Jupyter notebooks—documents that contain live code, equations, visualizations, and narrative text—now are among the most popular means to compute, present, discuss and disseminate scientific findings. In principle, Jupyter notebooks should easily allow to reproduce and extend scientific computations and their findings; but in practice, this is not the case. The individual code cells in Jupyter notebooks can be executed *in any order,* with identifier usages preceding their definitions and results preceding their computations. In a sample of 936 published notebooks that would be executable in principle, we found that 73% of them would not be reproducible with straightforward approaches, requiring humans to infer (and often guess) the order in which the authors created the cells.

In this paper, we present an approach to (1) automatically satisfy dependencies between code cells to reconstruct possible execution orders of the cells; and (2) instrument code cells to mitigate the impact of non-reproducible statements (i.e., random functions) in Jupyter notebooks. Our *Osiris* prototype takes a notebook as input and outputs the possible execution schemes that reproduce the exact notebook results. In our sample, Osiris was able to reconstruct such schemes for 82.23% of all executable notebooks, which has more than three times better than the state-of-the-art; the resulting reordered code is valid program code and thus available for further testing and analysis.

ACM Reference Format:
Jiawei Wang, Tzu-yang KUO, Li Li, and Andreas Zeller. 2020. Assessing and Restoring Reproducibility of Jupyter Notebooks. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416585

*Contributed the same as the first author. This work was done when Tzu-yang KUO was a visiting student at Monash University.
†Corresponding author.

ASE '20, September 21–25, 2020, Virtual Event, Australia

ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3416585

1 INTRODUCTION

Jupyter notebooks—documents that contain live code, equations, visualizations, and narrative text—have become the most widely used system for interactive literate programming [49]. They are being used to compute, present, discuss and disseminate scientific findings; and have emerged as the de facto standard for data scientists to easily record and understand data analyses [32, 51]. In September 2018, more than 2.5 million Jupyter repositories were stored on GitHub—10 times more than in 2015 [33].

One of the promises of Jupyter notebooks is that they should make scientific findings *reproducible*—that is, readers should be able to reconstruct and assess the path from raw source data to abstractions and findings, as presented in the notebook [18, 36]. Unfortunately, this is rarely the case. Published notebooks suffer from lack of data, from lack of modules, from lack of metadata indicating tool and library versions, or bad packaging [6]. But even if all of this is given, only a small fraction of notebooks can be faithfully reproduced.

Why is that so? A central feature of Jupyter notebooks is that the individual *cells* they are made of can be executed interactively *in any order.* The language interpreter (typically Python) will execute the code in the cell as soon as a user "runs" it. While Jupyter provides a "run all cells" feature that runs all cells starting from the topmost one, authors do not need to ensure that this results in meaningful execution order. It is not uncommon that notebooks output and present a result at the very beginning, followed by the code that actually produces the result, making the notebook more akin to an article than a conventional program. The interactive nature of notebooks makes all of this possible.

Jupyter notebooks assign a monotonically increasing number to each cell as it is executed; readers may thus find that a cell at the top may have been executed after a cell further downwards. However, even with this information, notebooks are hard to reproduce. In a 2019 study by Pimentel et al. [33], with a straightforward setting, less than 25% of valid notebooks (with defined Python versions and recorded execution order) could be executed without errors; and *less than 5% of them would actually produce the same results.*

Given how many scientific results now are being produced using Jupyter notebooks [37], it is time for the program analysis and testing community to make their best approaches available to notebook authors. But with 75% of valid notebooks not even running without errors, this means that the majority of notebooks are actually inaccessible for any automated testing and analysis tool.

In this paper, we present an automatic approach to make Jupyter notebooks reproducible, and in consequence, available for analysis and testing. Our approach automatically identifies and satisfies dependencies between Jupyter notebook cells, reconstructing the possible execution orders that reproduce the exact notebook results without errors. The resulting ordered code can thus be subject to testing and analysis (e.g., enabling continuous regression testing for notebook contributors to ensure the reproducibility of their notebooks); our approach thus forms a necessary prerequisite for further analysis of notebook code. If a given notebook cannot be reproduced, our *Osiris* prototype provides detailed debugging messages explaining (to notebook users) why reproducibility is not achievable.

This paper is organized as follows. After detailing the problem (cf. Section 2), we make the following contributions:

A study on the causes of non-reproducibility. We conduct a large-scale reproducibility study about Jupyter notebooks and manually summarise the root causes making notebooks non-executable and non-reproducible (cf. Section 3).

Making notebooks reproducible again. We design and implement a prototype tool called Osiris (cf. Section 4), which combines static analysis and dynamic testing to explore the possibilities of reproducing given notebooks. The resulting reordered code faithfully reproduces the results in the notebook; since it is valid program code, it is available for further testing and analysis. Since different users may require different amounts of reproducibility (e.g. some users may wish to reproduce *all* results as stated, others may only wish to re-run the notebook, possibly with different data), Osiris supports a number of *matching* and *execution* strategies to achieve the best result.

Automatic diagnosis for non-reproducible notebooks. In case a notebook cannot be reproduced, Osiris features a targeted debugging module to infer and report failure causes.

Our approach is effective: In our sample, Osiris was able to reproduce 82.23% of executable notebooks (cf. Section 5), which is a large improvement over the 16.7% listed in state of the art [33]. After discussing the potential implication and limitations of our approach (cf. Section 6), we depict related work (cf. Section 7) and close this paper with conclusion (cf. Section 8).

2 MOTIVATION

Let us start with some background and terminology.

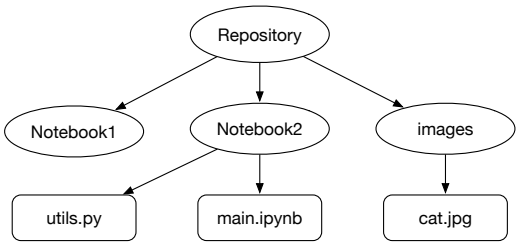


Figure 1: The file structure of a simple Jupyter notebook repository.

Jupyter is used to refer to the Jupyter application, which provides the computational environment to allow the execution of notebooks. **Notebook** (or Jupyter Notebook) refers to the literate programming document, which contains the actual content (e.g., main.ipynb in Figure 1) written by the **notebook authors**. Similar to the work of Pimentel et al. [33], Notebook and Jupyter Notebook will be interchangeably used in this paper. **Independent Python Code** will be used to refer to Python code that is not directly presented in a notebook but might be accessed by the notebook code. For example, the code shown in *utils.py* (cf. Figure 1) is regarded as independent python code. **Notebook Repository** refers to the project where the notebooks are written and managed. A notebook repository can contain multiple notebooks. For example, the repository shown in Figure 1 contains two Jupyter notebooks (i.e., Notebook1 and Notebook2).

Jupyter notebooks are sequences of *cells*, essentially *Code Cells* and *Text Cells* (cf. Figure 2). Code cells contain executable (Python) source code to generate results, while text cells contain text that enables programmers to state rationales behind the code logic; this text includes Markdown and HTML for rich text, images, formatting, and more. These combinations of these two cell types allows for *literate programming* [19], implemented by Jupyter in an *interactive computational notebook environment*. This environment allows parts of a notebook to be executed with immediate results, including formatted texts and visual graphs.

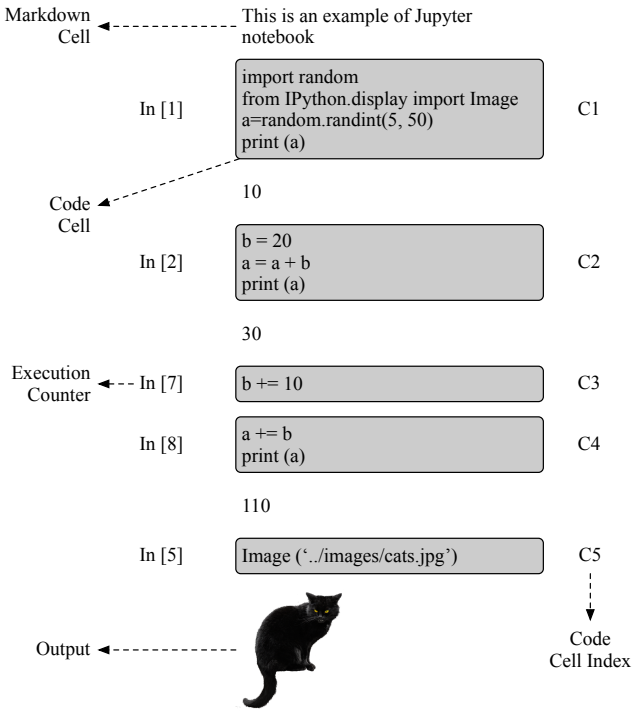


Figure 2: Jupyter notebook example.

Figure 2 illustrates a simple example of a notebook. It contains one (Markdown) text cell and five (Python) code cells. All the code cells have been executed as indicated by the execution mark (e.g., "In [1]") shown in front of the code cells. The execution mark tells

not only the information that the code cell is executed but also the order when the cell is executed (e.g., "In [5]" shows that the corresponding code cell is the fifth executed cell). The number of the execution mark is also known as the *execution counter* of the code cell. The output of a code cell, shown right under it, will always reflect the results of the latest execution (i.e., aligning with the execution counter).

It should be noted that a code cell can be either non-executed or executed in a given notebook. For non-executed cells, there will be no execution counters and outputs. Conversely, for executed cells, both execution counters and outputs (if any) will be generated for them. Moreover, the code cells can be executed by the user *in any order* and each of the cells can be executed multiple times, as long as the execution counter is respected. When skips of execution counters exist in a notebook, the actual execution order of the notebook cells becomes non-trivial to reproduce (as the counter records only the last execution). Subsequently, the skips make it difficult to automatically reproduce the results of the original notebook.

As an example, consider the notebook shown in Figure 2, in which three execution counter skips (i.e., 3, 4, and 6) are introduced. The notebook cannot be reproduced by simply following the top-down sequence of the code cells as well as the sequence of execution counters. For both cases[1], the reproducing process will fail at $C4$, where the output will be 90 rather than 110.[2] Consequently, we see that (1) Reproducing notebook executions can be a challenge; (2) There is a need for approaches that support users in reproducing notebook results; and (3) If a notebook cannot be reproduced automatically, there should be means to support users to do so manually.

3 REPRODUCIBILITY STUDY

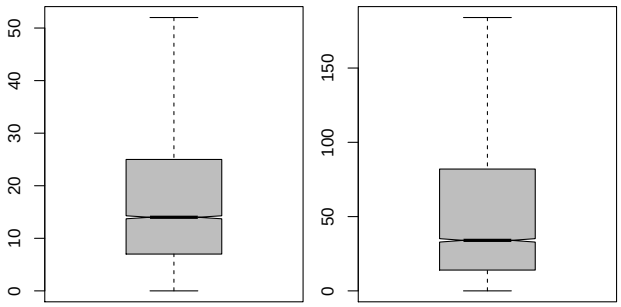
To understand the nature and extent of the reproducibility problem, we first conducted a large-scale reproducibility study, eventually guiding the design of our automated approaches.

3.1 Dataset Collection

How many Jupyter Notebooks are non-reproducible, and why? To answer this question, we apply the approach introduced by Pimentel et al. [33], collecting a dataset of notebooks from Github. Specifically, we have randomly cloned 10,000 Git repositories that have a file with "Jupyter Notebook" as identified language and have Python as the declared programming language. Among the 10,000 Git repositories, we further retrieve 10,000 notebooks for experiments and evaluations.[3]

Based on the dataset, we first look at the Python versions targeted by the notebooks. Among the 10,000 notebooks, 93% of notebooks specify detailed Python version requirements. The top-5 targeted versions are 2.7 (34.3%), 3.6 (31.4$), 3.5 (23.6%), 3.4 (3.4%), 3.7 (0.3%). The other 7% provide only vague information (such as *Python3*, *Python [Root]*).

[1]Let us assume at this point that the output of $C1$ is correct (i.e., the random function always returns 10).
[2]The correct execution sequence would be $C1$, $C2$, $C4$, $C4$, $C5$, $C3$, $C3$, $C4$.
[3]We select one notebook from each repository to avoid potential bias, where notebooks from the same repository may share similar problems for execution or reproducing. For such repositories that contain multiple notebooks, we simply choose the first one (by directory ordering) to form the dataset.



(a) # of cells (b) Max execution counter

Figure 3: Distribution of the number of cells (left) and the maximum number of original execution count(right) in the selected notebooks.

Figure 3 further presents the distribution of the number of cells and the maximum execution counter among the selected notebooks. The median and mean numbers are 14 and 20 for the number of cells; and 34 and 74 for the maximum execution counter, respectively. The fact that half of the notebooks have more than 15 code cells (executed more than 35 times) shows that the selected notebooks are actually "serious" notebooks suitable for this study.

3.2 Execution Environment Setup

To determine whether a notebook is reproducible, we need an execution environment for automated Python code execution.[4] Unfortunately, because of the infamous evolution of Python platforms (e.g., Python 2 and Python 3 are not compatible with each other), it is non-trivial to properly set up the execution environment.

As an example, the incompatibility of Python versions forces Python library contributors to maintain different versions of library modules (e.g., Numpy or Matplotlib) that are often imported by Jupyter notebooks. Specifically, a given notebook may import a large number of library modules, which per se also import many other library modules. It is hence non-trivial to know the necessary library modules beforehand when preparing the execution environment.

To produce consistent environments, we make use of Conda, an open-source library management system that supports automated library installation and updating [11]. Conda also implements an environment manager that allows different execution environments (such as Python 2 and Python 3) to be prepared and configured in the same system without involving conflicts between them. In this work, we set up four Conda virtual environments respectively for the top-5 leveraged major Python versions (except 2.7) discussed in the previous section, resulting in four pre-constructed environments for Python versions from 3.4 to 3.7.[5] For each of the four major Python versions, we install all the standard packages integrated into Conda by default (i.e., the Anaconda repository [2]), which leads to over 200 library packages included in the execution environment.

[4]In this work, we focus on Python-based Jupyter notebooks only.
[5]We decide not to pre-construct the environment for Python 2.7 as this version is now officially abandoned [35].

Table 1: Executable and reproducible notebooks. The execution is done based on the order of cells' execution counters.

Python	Notebooks	Executable	Reproducible
3.4	306	71(23.20%)	16(22.54%)
3.5	2,161	330(15.27%)	90(27.27%)
3.6	2,902	528(18.19%)	151(28.60%)
3.7	24	7(29.17%)	0(0.00%)
Total	5393	936(17.36%)	257(27.46%)

3.3 Experimental Study

Now that we have execution environments, we can execute all the notebooks in a suitable environment. Again following Pimentel et al. [33], we run the notebooks through the recorded execution counter of their code cells. For instance, regarding the notebook shown in Figure 2, the code cells will be executed via the following order: $C1 \rightarrow C2 \rightarrow C5 \rightarrow C3 \rightarrow C4$.

Because only four Python versions (i.e., 3.4–3.7) are configured in the execution environment at the moment, 4,130 out of the 10,000 notebooks (e.g., with Python 2.7 or without explicitly mentioning the depending version) are directly excluded from the execution. Among 5,870 notebooks, we further filtered out 477 notebooks that contain no execution records. For the remaining 5,393 notebooks, Table 1 summarizes the execution and reproducible results. The results are concerning. To start, *only 17.36% of notebooks can be fully executed* without error. Even worse, among the 936 executed notebooks, only 27.46% of them can be *exactly* reproduced. These results is more or less in line with the results reported by Pimentel et al. [33]—and still surprising, as we would have anticipated that the execution/reproducible rate would be high as Jupyter notebooks are likely used for education [7, 29].

3.4 Root Causes of Non-Reproducibility

Let us now explore the reasons (root causes) why the majority of notebooks (i.e., 679 = 936-257) cannot be executed or reproduced.

In this work, we summarize the root causes in two types: (1) Jupyter notebooks cannot be fully executed and (2) Jupyter notebooks can be fully executed but cannot reproduce the same results as of the original version. Recall that the objective of this work is to restore the reproducibility of notebooks. We will hence mainly focus on the root causes of the latter.

For the case where given notebooks can be fully executed but the results cannot be reproduced, our manual observation has identified the following reasons that may lead to different execution results compared to that of the original execution. We now summarize the representative ones, together with their absolute and relative prevalence.

R1: Randomness (276/679 = 40.6%). Many scientific computing programs require random functions, e.g. for sampling from Gaussian distributions or data shuffling. They produce different results after each execution (if no seed is given), making it hard to determine if the results can be reproduced. In our dataset, around 41% (276 out of 679) of non-reproducible notebooks contain random functions.

R2: Time and Date (87/679 = 12.8%). Time functions are recurrently used by notebook authors (roughly 13% of notebooks in our non-reproducible set) to achieve some specific functions such as evaluating time efficiency, logging for SQL operations, etc. Since time changes continuously, the outputs of the time function also vary every time, making it hard to ascertain the reproducibility of the code.

R3: Plots (352/679 = 51.8%). We see a considerable number of notebooks that cannot be reproduced because of differences in plotted images. Indeed, in some cases, images are generated based on input data or random numbers that cannot be ensured to remain the same each time when the notebooks are executed. Additionally, the running environment can also impact plotting results. Given the same inputs, if different library versions are used, the plotted images could also be different (because of API improvements[6] or change of default parameter values).[7] Furthermore, when the *matplotlib* module is not properly invoked, the image may not be properly displayed. Instead, there will be some warning texts related to *Python's magic functions* (e.g.,*[<matplotlib.lines.Line2D at 0x7f24b114c3c8>]* that could be different from time to time. Among the 352 image-related non-reproducible notebooks, around 10% of them are caused by incorrect usage of *matplotlib*.

R4: External Inputs (18/679 = 2.7%). Jupyter notebooks may rely on external inputs (data fetched from web servers such as web crawler demonstration) to execute. However, the external inputs are subject to change (such as URL decay [50]), causing inconsistencies between the reproduced results and the recorded original results.

R5: Floats (21/679 = 3.1%). In notebooks, floating point numbers may be printed differently depending on the running machines, or the targeted Python versions. Therefore, the reproduced results (relevant to floating point numbers) might be different from the original ones.

R6: Container Traversal (29/679 = 4.3%). In Python, the order in which sets and dictionaries are traversed is not fixed. Hence, this order may differ across execution environments, causing differing cell outputs. In our non-reproducible set, 4% of notebooks show this root cause.

R7: Execution Environment (181/679 = 26.7%). Notebooks may access the execution environment information (e.g., number of CPUs, Python package versions, the memory location of variables, etc.) that is usually specific to each setting and hence is different from one another. Moreover, in different execution environments, the same data might be printed in different formats. For example, the number 1 might be printed as 1.0. The matrix (i.e., two-dimensional array) might be printed with different spaces between the elements. All these differences will impact the reproducibility of notebooks.

Inappropriate execution order of cells. Apart from the errors raised by execution environments, we also observe a significant number of notebooks that fail to be executed due to poor code quality, e.g., containing *name errors* (undefined variables), *key error* (key not found in dictionaries), *syntax errors,* etc. We found this amount of errors surprising, as we would have expected code in notebooks to be of good quality. Indeed, the notebooks are collected from

[6] https://matplotlib.org/users/dflt_style_changes.html
[7] https://bit.ly/2QNO3P9

publicly released repositories, for which most of them are related to educational materials for teaching inexperienced developers.

This fact hence motivates us to go one step deeper to check the reasons behind this kind of errors. Our manual observation reveals that these errors are yielded because *the execution order of cells is inappropriate* (e.g., a variable is used before its definition). Since this problem is not caused by the execution environment, we decide to consider it as another root cause making notebooks non-reproducible. This root cause refers to problems that cause notebooks to be non-reproducible because the code cells are not executed in the intended order. Ideally, if the appropriate execution order of code cells is respected, the notebooks, falling into this category, would become reproducible.

In summary, all of these findings will impact *any approach* that attempts to reproduce, test, or analyze Jupyter notebooks. Indeed, even for executable notebooks, it may not always be possible to reproduce the exact results as originally obtained. In this work, we thus attempt to resolve all the aforementioned non-reproducible root causes (R1–R7 and Inappropriate execution order), aiming to design an automated approach for restoring the reproducibility of notebooks as much as possible.

4 OSIRIS

Now that we have identified causes for non-reproducibility, let us fix them. We have implemented a tool called Osiris, which adopts different strategies to resolve the aforementioned root causes of non-reproducibility, attempting to maximize the execution and reproducibility of notebooks. Osiris takes as input a Jupyter notebook and outputs the possible execution schemes that reproduce the exact notebook results. If Osiris fails to reproduce the notebook, it will highlight the location of failures (i.e., non-reproducible parts) that could be useful for understanding the root causes of non-reproducibility of Jupyter notebooks.

Figure 4 illustrates the working process of Osiris, which is mainly made up of four modules: (1) Cell Code Parsing, (2) Cell Dependency Graph Construction, (3) Strategy-based Reproducing, and (4) Targeted failure debugging. We now detail these four steps, respectively.

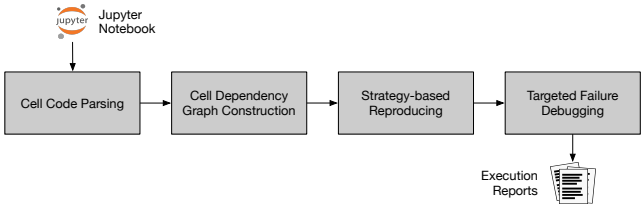


Figure 4: The working process of Osiris.

4.1 Parsing Cell Code

To analyze the code in Jupyter Notebook cells, we first have to *parse* it. For this purpose, Osiris makes use of built-in Python parsers, transforming Python code into abstract syntax trees (ASTs). Special care is taken to handle and resolve `import` statements as well as translating *syntactic sugar statements*[8] and *lambda statements*[9] into analyzable normal forms.

4.2 Resolving Cell Dependencies

The second module of Osiris aims at identifying and characterizing *dependencies* between notebook cells. By statically parsing the code cells, Osiris builds the so-called cell dependency graph (CDG) to model the relationships.

Technically speaking, a code cells *depends* on the set of variables, functions, classes, modules used, defined, or imported in the code snippets. A given code cell can be executed as long as all its used variables are defined in the Jupyter execution environment. In this work, we leverage the classical producer-consumer model to represent the define-use relationship of variables in the code cells of a notebook.

Given the parsed abstract syntax tree (AST) of a code cell C, as illustrated in Figure 5, we calculate the stored and loaded variables (including method calls and class initialization). The stored variables, defined functions/classes, imported modules are put into a producer set while loaded variables/functions/calls are added to a consumer set. Note that, within a code cell, if a variable is produced after being consumed (e.g., a in $C2$), this variable will be excluded from the producer set. The calls of the built-in functions (e.g., *hash(), ascii()*) will not be added into consumer set [24]. Hence, the final producer and consumer sets of $C2$ are $P(C2) = \{b\}$, $C(C2) = \{a, b\}$.

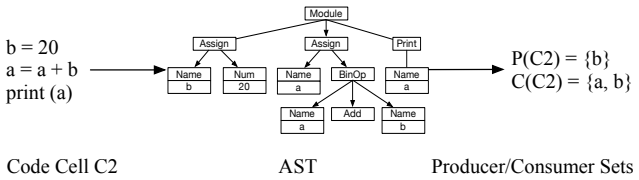


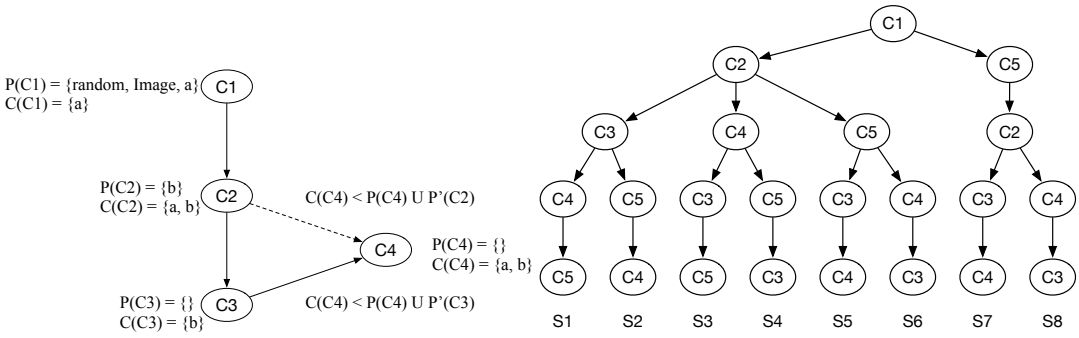
Figure 5: Generating producer/consumer sets.

After the producer/consumer sets are calculated for every code cell, Osiris then leverages this knowledge to construct a CDG for the notebook. In the CDG, every code cell becomes a node. Edges will be added if given two code cells (or nodes, say C_i and C_j) have no conflicts between each other. Concretely, an edge $C_j \rightarrow C_i$ can be added as long as the following constraint is respected: $C(C_j) \subset P(C_j) \cup P'(C_i)$, where $P'(C_i)$ is the accumulated producer set after C_i is executed (or reached from a root node in the CDG). Consequently, after the execution of C_j, $P'(C_j) = P(C_j) \cup P'(C_i)$ (i.e., the produced set is the combination of the newly produced variables and all the previously produced ones.).

It is worth mentioning that a given node in a CDG can be reached from the root node via multiple paths. Hence, the accumulated producer set may not be unique. Take the motivating notebook (cf. Figure 2) as an example, as demonstrated in Figure 6(a), C_4 can be reached from both C_2 and C_3.Subsequently, the accumulated producer set $P'(C_4)$ could be $P(C_4) \cup P'(C_2)$ or $P(C_4) \cup P'(C_3)$.

[8] Syntactic sugar statements (e.g.,*i += 5*, *a *= 5*) are designed to make things easier to read or express.

[9] Lambda statements are usually used to concisely define functions. For example, the following simple lambda statement *x = lambda a : a + 10* actually defines a function



(a) Example of multiple paths (b) Full paths

Figure 6: The possible execution orders of code cells w.r.t. producer/consumer constraints. The left sub-figure presents a simplified example of multiple paths while the right sub-figure illustrates all the possible execution paths for the motivating example shown in Figure 2.

4.3 Strategy-based Reproducing

As discussed in Section 1, we assume that Osiris users may have different expectations and requirements regarding reproducibility. Osiris thus supports two types of strategies to be configured: (1) match strategy and (2) execution strategy. We now detail these two types respectively.

4.3.1 Match Strategy. We have integrated three match strategies into Osiris:

Strong Match (Stage 1). The reproduced results should exactly match the results yielded by the original notebook. This matching strategy has been used in our preliminary reproducibility study as well as in the empirical study of Pimentel et al. [33]. This matching strategy is needed by users who want to exactly reproduce all results; we consider it as the baseline for reproducing Jupyter notebooks.

Weak Match (Stage 2). The reproduced results are the same when repeating the execution in the same environment but may not match exactly to that of the original notebook. This matching strategy targets users who are more interested in reusing and re-executing the code rather than reproducing the exact results.

With this matching strategy, most of the non-reproducible root causes categorized in R4–R7 are expected to be mitigated. Indeed, let us take R4 as an example, a given notebook dedicated to analyzing a Github project will be updated weekly. Since the Github project is continuously updated, the analyzing results will be different occasionally. However, if we launch the notebook twice (with a short time interval) in the same environment, the analyzing results should be identical.

Best-effort Match (Stage 3). For code cells that cannot be weakly matched (e.g., $R1 - R3$), Osiris will go one step deeper to explore alternative means to reproduce the code cells, aiming to achieve weakly match with special consideration (i.e., best-effort match). In particular, Osiris leverages a code instrumentation-based approach to achieve the aforementioned objective. Given a code cell, Osiris attempts to transform it into a semantically equivalent version. If the new version can be weakly matched, we consider the original code cells can be matched with best-effort. Again, this strategy targets users who want to reuse the notebook or parts of its code.

Osiris has three "antidotes" to achieve best-effort matches:

(1) Regarding **randomness,** we attempt to configure a seed before the random functions being called. As illustrated in Listing 1, random functions may be used by various library modules, which need to be addressed separately.
(2) For **time/date functions,** we select the "time-freeze" tactic to mock the time-related modules of Python [13]. Within this tactic, all the functions relying on the current timestamp such as *time.time()* will return constant values.
(3) (As for **plots** involving the `matplotlib` library, we force the notebook to directly display the image (to avoid the output of memory addresses) by explicitly inserting a magic function (i.e., %matplotlib inline) at the beginning of the notebook.

```
# For the random module of Python
random.seed(100)

# For the random module of numpy, sklearn and scipy
numpy.random.seed(100)

# For time functions
from freezegun import freeze_time
freezer = freeze_time("2019-01-01 00:00:00")
freezer.start()

# For matplotlib images
%maplotlib inline
```

Listing 1: Sample solutions for best-effort match.

4.3.2 Execution Strategy. The second type of strategies that can be configured in Osiris is related to the execution of the notebook. This strategy type allows users to specify how do they want to

named x. When statically parsing this statement, it should be interpreted as *def x(a): return a+10;*

execute the code cells in a given notebook. So far, Osiris has been equipped with three execution strategies.

Original Execution Counter (OEC). In this strategy, the code cells of a given notebook will be simply executed following the order of the cells' execution counters. Skipped counters (e.g., 3, 4, 6 in Figure 2) will be simply ignored. For the notebook presented in Figure 2, the execution order under this strategy will be solution $S5$ in Figure 6. All the experiments conducted in Section 3 are based on this strategy.

Top-down (Normal). In this strategy, the code cells will be executed following their natural order, from top to down. For the notebook illustrated in Figure 2, the execution order will be solution $S1$ in Figure 6.

Cell Dependency Graph (CDG). This strategy attempts to execute the code cells of a given notebook in all the possible orders with respect to the producer/consumer dependencies. Like the aforementioned two strategies, this strategy will only execute each code cell once. Regarding the notebook shown in Figure 2, this strategy will generate eight possible solutions for reproducing the notebook.

4.4 Targeted Failure Debugging

Using its strategy-based reproducing module, Osiris is now capable of exploring the reproducibility of a given notebook in different ways. Some of the attempts may not be able to reproduce the results (even with weak or best-effort matches). When the reproducing attempt fails, or even the execution fails, there will be, most likely, no logs or execution traces illustrating the reason why it fails, giving no clue for users to understand and refine their execution strategies. For this reason, Osiris features a *debugging module*. Given a notebook and a code cell where the reproducing process or execution fails, this module will then record the execution trace of the cell. To this end, we implement an *instrumentation-based approach* to record the execution status at the client code side. Particularly, for every code line of the specified cell, we inject logging statements to record the current execution state (e.g., the variables and their values) the code line will be executed with.

For each line executed in the cell, we record and compare a snapshot of all self-defined variables and their states. After cell execution, Osiris locates the first suspicious line in which the freshly computed variable values are inconsistent with published values.

Additionally, Osiris also highlights *non-repeatable code cells*, which may generate the same outputs but will yield variable values that would impact the subsequent execution of other cells. In our motivating example from Figure 2, all the first four cells (e.g., $C1$–$C4$) are non-repeatable ones. These non-repeatable cells may prevent notebook reproduction if execution counter skips are present.

5 EVALUATION

Our evaluation addresses the following research questions.

- **RQ1:** Can Osiris build sound cell dependency graphs for Jupyter notebooks?
- **RQ2:** To what extent can Osiris improve the reproducibility rate of Jupyter notebooks by varying the match strategies?
- **RQ3:** With different execution strategies, can Osiris improve the reproducibility rate of Jupyter notebooks?
- **RQ4:** Can the targeted failure debugging module of Osiris help pinpoint the root causes for non-reproducibility?

5.1 RQ1: Soundness of Cell Dependencies

Our first research question concerns the soundness of the cell dependency graph (CDG), which is important for Osiris to fully explore the reproducibility of Jupyter notebooks. Indeed, Osiris leverages the CDG to generate all the possible execution paths when *dependency* execution strategy is enabled. Ideally, if the CDG is sound, all the paths generated based on it (hereinafter referred to as *CDG paths*) would be executable (which may not be able to fully reproduce the notebook though). To this end, we transform the problem of checking the soundness of the generated CDG to the problem of checking the executability of CDG paths.

Since the number of CDG paths is huge, it is not practical to exhaustively execute all of them. Therefore, for each notebook to be tested, we randomly select 10 CDG paths to fulfill this experiment. To avoid potential biases, e.g., the non-executability is due to syntax errors in the Python code, this experiment should only be conducted on notebooks known as executable ones.

Therefore, we choose the 936 notebooks that are demonstrated to be executable in Section 3 as the input dataset to perform the experiment. Among the 936 notebooks, despite that only 10 CDG paths selected, our approach finds that (1) 79.81% of them have *all* the selected CDG paths successfully executed, (2) 14.32% of them have *at least one* CDG path failing; and (3) the remaining 5.88% have *all* CDG paths failed.

We further look into the reasons causing certain CDG paths to fail in our experiment. Our manual observation reveals that the fails are indeed related to the soundness of the CDG built by Osiris. The problems are mainly introduced by the static code analysis step of Osiris, where certain complicated Jupyter/Python features are overlooked. Listing 2 presents a common challenge that keeps Osiris from building a sound CDG. Because Osiris is not aware of dictionary keys, it will consider that cell 3 is reachable from cell 1 since the producer/consumer dependency is fulfilled. Unfortunately, accessing this unknown key will result in errors because the dictionary key is not registered yet. The overlook of the aforementioned Python features subsequently leads to incorrect producer/consumer sets and thereby resulting in unsound CDGs. Nonetheless, we do not observe any failures caused directly by the producer/consumer dependency algorithm. The fact that the majority of notebooks can be successfully processed to generate executable paths shows that the cell dependency graph construction process is generally sound.

```
21 # access unregistered dictionary key.
22 a = {} # cell 1: produce a
23 a['ase'] = "2020" # cell 2
24 x = a['ase'] # cell 3: consume a
```

Listing 2: An example of challenge that keeps Osiris from building a sound CDG.

Finally, at the end of this research question, we conduct one more experiment to check if 10 is a good number when randomly sampling the CDG execution paths. To this end, we randomly select 10 notebooks with CDG paths that can be successfully executed. Now, in addition to 10 paths, we further check the execution rate of

Table 2: Fraction of CDG paths that can be used to successfully execute notebooks.

Username/Repository Name	10	20	50	100
vuddagiri-mounica123/datascience	1.00	1.00	1.00	1.00
willengel88/Pandas-homework	0.50	0.60	0.54	0.51
rjsampa/Curso_python_PEC2018	1.00	1.00	1.00	1.00
carlorizzante/MITx6.00.1	1.00	1.00	0.98	0.99
fgnt/oaf	1.00	0.90	0.96	0.97
ratnakumar-nakka/Python	1.00	1.00	1.00	1.00
Novobura/2016-Election-Analysis	1.00	1.00	1.00	1.00
murrayLuke/dithering	1.00	1.00	1.00	0.99
svenchilton/springboard	0.20	0.30	0.34	0.25
mc-carthy/pyDSML	1.00	1.00	1.00	1.00

Table 3: Improvement of execution rate when varying the match strategies of Osiris.

Python	Notebooks	Strong	Weak	Best-effort
3.4	71	16(22.54%)	40(56.34%)	52(73.24%)
3.5	330	90(27.27%)	203(61.52%)	261(79.09%)
3.6	528	151(28.60%)	316(59.85%)	393(74.43%)
3.7	7	0(0.00%)	3(42.86%)	3(42.86%)
Total	936	257(27.46%)	562(60.04%)	709(75.75%)

20, 50, and 100 paths. Table 2 illustrates the 10 selected notebooks and additional execution results. No matter which threshold is set, the execution rates are kept more or less the same. This evidence suggests that 10 is a good threshold for testing the CDG paths.

5.2 RQ2: Match Strategies

The second research question concerns the effectiveness of the match strategies implemented in Osiris. Specifically, given a set of non-reproducible notebooks (strong match via original execution counter, as explained in Section 3), to what extent can our approach (with weak and best-effort strategies) improve the execution/reproducible rate? To this end, we launch Osiris on all the 936 notebooks that have been shown executable in our preliminary reproducibility study. Except for switching the match strategy to weak or best-effort strategies, we keep all the other parameters unchanged, i.e., the code cells are executed following the same order (i.e., original execution counter).

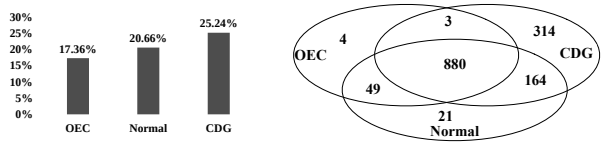
Table 3 summarizes the execution results. Among the 936 executable notebooks, with weak match strategy, the reproducible rate can be doubled from that of the weak match, increasing from 27.46% to 60.04%. When best-effort is enabled, an additional 15% of notebooks can be reproduced, resulting in 75.75% of reproducibility rate. This significant improvement of the reproducible rate shows that Osiris is useful and effective for assessing the reproducibility of Jupyter notebooks.

We further look into the breakdown of the number of notebooks that are additionally reproduced thanks to the best-effort match strategy. The majority of them are contributed by the random antidote. Among the 147 additionally reproduced cases, we find that the main improvement comes from the antidote to random functions, accounting for 113 (roughly 77%) notebooks. The improvements over time and image plot antidotes are 6 and 28 notebooks, respectively. Our in-depth analysis further reveals that our best-effort approach may have overlooked some notebooks. For example, we additionally find that there are 18 notebooks (out of the 227 non-reproducible ones) having also accessed into random functions. After manual investigation, we believe that these overlooked notebooks should not be considered as false negatives of our best-effort approach since the errors are likely caused by different reasons (i.e., not from the random function).

5.3 RQ3: Execution Strategies

We now look at the execution strategies introduced to Osiris when reproducing Jupyter notebooks. Instead of the original execution counter, which has been used in all the aforementioned experiments, we additionally launch Osiris via (1) normal (top-down) and (2) producer/consumer dependency (CDG) strategies. As discussed earlier, the possible CDG paths are generally too many to execute all. Therefore, as with RQ1, we randomly choose 10 CDG paths to fulfill this experiment. As long as one path from the randomly sampled 10 paths is capable of executing or reproducing the notebook, we will consider the notebook as such.

Since different execution strategies may be able to execute different notebooks[10], we resort to the original 5,393 notebooks for this experiment. As illustrated in Figure 7, interestingly, with best-effort match strategy, both normal and CDG execution strategies are able to improve the overall execution rate, giving 20.66% and 25.24%, respectively.



(a) Breakdown. (b) Venn diagram.

Figure 7: Executable rates and notebooks achieved by varying the execution strategies of Osiris.

In total, by counting the three execution strategies as a whole, Osiris can successfully execute 1,435 notebooks. Figure 8 further illustrates the distribution of reproduced notebooks over the three strategies. Among the 1,435 executable notebooks, 1,180 of them have been shown reproducible, giving a reproducibility rate of 82.23%, which has more than tripled the rate of the state-of-the-art and significantly increased from our results by leveraging OEC execution strategy along [33]. This result experimentally shows that the execution strategies of Osiris are indeed useful for restoring the reproducibility of Jupyter notebooks.

[10]We hypothesize that not all the non-executable notebooks are related to the inappropriate set up of the execution environment. Therefore, by leveraging different strategies to execute those notebooks, we might be able to find more executable notebooks, without updating our execution environment. Recall that executability is not the main focus of this paper.

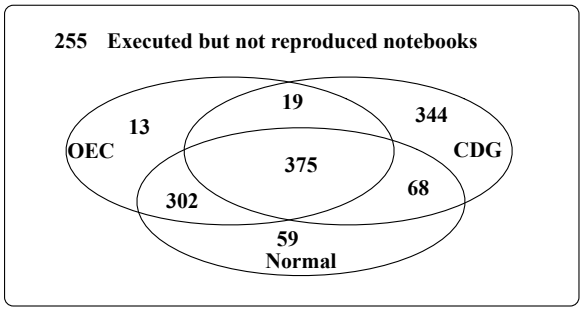


Figure 8: Reproducible notebooks among executable ones.

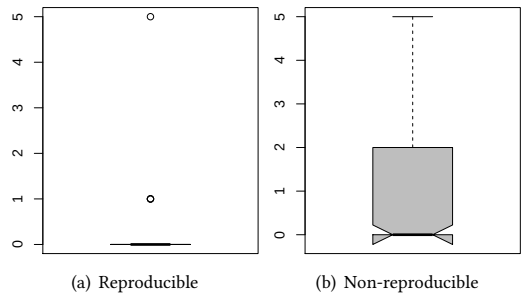


Figure 9: Distribution of the number of non-repeatable cells between reproducible and non-reproducible notebooks.

5.4 RQ4: Performance in locating faults via debug functionality

As revealed in the previous section, among the 1,435 executable notebooks, 225 can not be reproduced by Osiris. In this research question, we look at the targeted failure debugging module of Osiris, aiming at evaluating the usability of this module towards locating the problematic statements making notebooks non-reproducible (at least by Osiris). To this end, we apply Osiris again, with the targeted debugging module enabled and the first non-reproducible cell as input, to the 255 non-reproducible notebooks. For each line of code in the cell, Osiris will dump the execution status before and after the execution of the code line. If a suspicious line is identified, Osiris will further report it as such.

To evaluate the accuracy of the debugging module, we randomly select 20 notebooks with suspicious lines reported and manually check the results. All the selected notebooks are hence manually executed, using predefined execution strategies. Table 4 summarises the experimental results. For the 60 samples randomly selected, our manual validation confirms that 45 of them are correct, giving an accuracy at 75%. The false positives are mainly caused by two limitations of Osiris: (1) Some code lines (e.g., function calls) will only change the outputs of the cell but may not alter the execution states. Osiris cannot be aware of this. (2) Osiris misses some data types (such as the ones from third-party libraries such as Pandas DateFrame) when recording and debugging execution states.

Table 4: Performance of Osiris's targeted debugging module.

Execution Strategy	Random Samples	True Results	Non-repeatable Notebooks
OEC	20	12 (60%)	12 (60%)
Normal	20	16 (80%)	15 (75%)
CDG	20	17 (85%)	16 (80%)
Total	60	45 (75%)	43 (72%)

The last column of Table 4 lists the number of non-reproducible cells (reported by Osiris) that are also non-repeatable cells. As it is similar to that of non-reproducible cells, there could be a strong correlation between non-reproducible and non-repeatable cells. We hence conduct another experiment to evaluate this suggestion empirically. We randomly select two datasets (i.e., 100 reproducible notebooks and 100 non-reproducible notebooks) and launch Osiris to check the number of non-repeatable cells. Figure 9 presents the distribution of the results. A Mann-Whitney-Wilcoxon (MWW) test [12] confirms that the difference in terms of the number of non-repeatable cells between these two datasets is indeed significant (i.e., p-value is smaller than 0.001).

Cliff's *delta* effective size [26] further explains that the correlation between these two datasets is strong, indicating that non-repeatable cells could be one of the main reasons causing Jupyter notebooks to be non-reproducible. Therefore, we encourage notebook authors to mitigate the usage of non-repeatable cells to promote the reproducibility of their Jupyter notebooks.

6 DISCUSSION

Aiming at maximizing the reproducibility of Jupyter notebooks, our tool will be beneficial to both notebook authors and users. **Notebook authors** can leverage Osiris to check the reproducibility of their notebooks before committing. To ease the reproducibility for notebook users, the reproducing details can be dumped and included in the commit. Indeed, we believe that Osiris can be configured as a regression testing tool that can be regularly (e.g., nightly) executed to identify potential errors introduced by the development process. **Notebook users** can leverage Osiris to understand how a given notebook, which is released without giving any reproducing detail, can be reproduced. They can also gain the confidence that the notebook (if reproduced by Osiris) is correct (hence its code can be trusted and reused), even if the original output cannot be literally reproduced. Finally, **notebook researchers** can develop new testing and analysis approaches for Jupyter notebooks. For example, our fellow researchers could take the initiative to modify the Jupyter framework to record all the execution orders, including the skipped ones, which has caused difficulties (e.g., CDG construction) to properly restore the execution sequences of notebook cells.

6.1 Limitations

Insufficient Setup of Execution Environment. As discussed in Section 3, the majority of notebooks fail to be executed due to library dependencies (e.g., import error because of missing modules, or contain removed functions due to inappropriate library versions). This finding is similar to the one conducted by Pimentel et al. [33]. Although inferring a sufficient execution environment is not the focus of this work, we believe that having such an environment would

significantly complement our work towards restoring the reproducibility of Jupyter notebooks. Recently, Horton and Parnin [15] have presented an approach to automatically infer environment dependencies for Python code scripts. This work could be leveraged to set up appropriate environments for executing and reproducing Jupyter notebooks.

Unsound static Python code analysis. At the moment, Osiris has some drawbacks that keep it from performing sound static analyses of Jupyter notebooks. First of all, code cells in notebooks can be updated following successful execution (e.g., magic functions are likely to be removed after executing). This execution history is unfortunately not recorded and hence will introduce inconsistencies between the code and the outputs. Second, the variable value in notebooks can be updated in succeeding cells, which may cause runtime exceptions. For example, a matrix of $(M \times N)$ defined in one cell could be reshaped to a $1 \times (M \times N)$ array in another cell. Different CDG paths may cause exceptions of dimension mismatches in matrix multiplication.

Incomplete match/execution strategies. In all the current execution strategies, each code cell will only be executed once, which however may not be the cause in practice. Indeed, the current strategies cannot even reproduce the motivating notebooks shown in Figure 2. Therefore, we believe that there is a need to explore other strategies that take into account cell dependencies while respecting the exact execution counter of the cells. The length of the execution orders should be equal to the maximum value of the execution counter in the given notebook. In this work, we further implement an execution strategy fulfilling the aforementioned requirements. While respecting the exact execution counter of the cells, we additionally add random cells (w.r.t. the producer/consumer dependency) to all the skips in the execution counter.

6.2 Threats to Validity

The main threat to the validity of this work lies in the size of our dataset, though we started from 10,000 real-world Jupyter notebooks, which (unfortunately) may not be fully representative [4]. We attempt to mitigate this threat by randomly selecting the notebooks from a large set. To avoid potential biases, we only select one notebook from each Git repository that may contain multiple notebooks. Since our approach only supports Python code analysis, we further limit the selected dataset to be Python-based notebooks only, letting several notebooks involving other programming languages such as R and Julia overlooked.

Apart from the dataset, our work also involves substantial manual work. For example, the root causes of non-reproducibility are manually summarized based on our preliminary reproducibility study. The results of targeted failure debugging analysis are also characterized manually. Such manual processes are known to be error-prone. To mitigate the threat, we have cross-validated the results. We also release our tool and dataset for public access.

Furthermore, we have no evidence to show that the randomly selected notebooks evaluated in this work are designed to be reproducible. If they are not intended to be reproduced, our evaluation results might be impacted.

Nevertheless, as shown by Rule et al. [44], Jupyter notebooks have been recurrently leveraged to support reproducible researches and hence such impact should be neglected. Moreover, a large number of notebooks cannot be executed (e.g., due to incorrect dependencies or unsupported Python versions), which may impact the generality of our sample set. To mitigate this, we have manually sampled some of the executable notebooks. Our manual observation confirms that the randomly selected notebooks are not toy ones. The fact that half of the notebooks have more than 15 code cells also confirms this observation.

7 RELATED WORK

Despite its popularity, Jupyter notebooks have not yet been well studied by the software engineering (SE) community. To the best of our knowledge, there are only a few such studies available to notebook authors. Wang et al. [52] have conducted a preliminary study on the code quality of Jupyter notebooks. They have empirically found that Jupyter notebooks are inundated with poor quality code, e.g., not respecting recommended coding practices, or containing unused variables and deprecated functions. The authors argue that there is a strong need to programmatically analyze Jupyter notebooks. Our work, by providing valid program code (reordered code cell, for example), and its cell dependency graph, can be considered as the first attempt in the SE community towards systematically analyzing and testing Jupyter notebooks.

The work most closely related to ours is the one by Pimentel et al. [33]. The authors have conducted a large-scale empirical study about the quality and reproducibility of Jupyter notebooks. Unlike the work conducted by Wang et al. [52], which mainly focuses on the quality of the code presented in the notebooks, our paper is more focused on the good or bad practices used in the development of notebooks. Pimentel et al. [33] have also investigated the reproducibility of Jupyter notebooks. Through a straightforward approach, the authors empirically find that only 24.11% of their selected notebooks can be executed without errors, and only 4.03% of them can produce the same results, compared to that of the original outputs of the notebooks. The authors have also summarised the common root causes making notebooks non-executable and hence non-reproducible. As an implication of their work, the authors argue that there is an opportunity to improve the reproducibility rate in notebooks by devising approaches addressing the root causes. In this work, we take action to actually design and implement such an approach, which combines both static and dynamic analyses, to explore the possibilities of reproducing Jupyter notebooks.

Notebooks have been frequently investigated by researchers outside of Software Engineering [16, 17, 22, 38, 42, 45, 46, 48, 53]. For example, Rehman et al. [38] address the reproducibility of Jupyter notebooks by enriching the provenance-based semantics of interactive notebooks. Their ProvBook prototype aims at capturing and viewing the provenance of notebook changes and thereby providing a means to track complete paths of scientific experiments. Yaniv et al. [55] designed "SimpleITK Jupyter Notebooks" based on Jupyter Notebook, an image analysis environment for researchers with different levels of development skills, to facilitate reproducible research.

Kery et al. [16, 17], from the Human-Computer Interaction (HCI) community, have conducted several studies in understanding notebooks w.r.t. their literate programming features and untangling

messy histories of notebooks. By interviewing and surveying 66 data scientists, they have observed that notebooks are mainly used for scratchpad construction, scripts preparation and knowledge sharing. To combat the challenge of navigating messy version data to pick out the relevant information for a given task, they introduce an approach called *Verdant* for supporting lightweight interactions among many versions of code and non-code artifacts in the editor.

Similarly, Rule et al. [46] also look at computational notebooks from the human factors point of view. The authors have empirically found that, via a large-scale empirical study of computational notebooks on Github, computational notebooks may not always contain explanatory text and only a small set of them will discuss the reasoning or results of the methods described. Rule et al. [43] have also noticed the non-reproducing problem of notebooks and hence summarised the top 10 simple rules for reproducibility in Jupyter notebooks. The authors argue that notebook authors should advocate in promoting the reproducibility of notebooks. They even suggest that notebook authors should ask lab-mates or colleagues to try to run their notebooks to ensure their reproducible uses. Our work fully supports their claims (i.e., the reproducibility of notebooks is very important) and supplement their claims with an automated approach for helping notebooks authors check the reproducibility of their notebooks.

Our work focuses on the reproducibility of Jupyter notebooks, aiming at reconstructing the execution orders that reproduce the exact notebook results and hence supplementing the implementation of advanced static and dynamic analyses of Jupyter notebooks. Although the def-use has been a traditional mechanism in program analysis [31][9], the relevant research on Jupyter notebooks is still empty. The uniqueness of Jupyter notebook's cell structure requires def-use relation to be built for code cells in order to determine their dependencies.

The software engineering community has investigated the reproducibility of other software artifacts [3, 10, 14, 21, 27, 41]. For example, to ensure reproducibility, researchers have proposed various approaches to achieve record-and-reply executions [28, 34, 47], facilitate bug reproductions by analyzing execution logs or dumps [1, 20, 54], and understand and detect flaky tests or compatbility issues that make reproducibility studies difficult [5, 8, 23, 25, 30]. As another example, Ren et al. [39] have proposed an approach to locating the problematic files for unreproducible builds. The authors claim that identifying unreproducible issues remains a labor-intensive and time-consuming challenge due to the diversity of root causes that may lead to unreproducible binaries. In line with this research, the authors further present a dedicated study focusing on locating such root causes [40]. In general, reproducibility has been considered a hard yet useful endeavor in the SE community. Indeed, as advocated by Anda et al. [3], achieving more reproducibility in SE remains a great challenge for SE research, education, and industry.

8 CONCLUSION

Motivated by a low reproducible rate of Jupyter notebooks reported by the state-of-the-art and detailed in our study, we present to the SE community the first work to automatically restore reproducibility of Jupyter Notebooks. Our Osiris prototype explores all the possible execution schemes to reproduce as much of notebook results as possible. In the evaluation of Osiris, we show it is effective to restore the reproducibility of Jupyter notebooks, achieving a significant improvement over the state-of-the-art[33]. In particular, our approach enables the use of static and dynamic analyses on Jupyter Notebooks, which can be used for testing, empirical studies, automatic repair techniques, and more.

To help readers access our tool and replicate our experiments, we make our tool Osiris and scripts available at

https://github.com/Osiris-Jupyter/Osiris

REFERENCES

[1] Gautam Altekar and Ion Stoica. 2009. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 193–206.

[2] Anaconda. 2019. Anaconda Enterprise 4 Repository: Open Data Science Hub. (2019). Retrieved August 23, 2019 from https://docs.continuum.io/anaconda-repository/

[3] Bente CD Anda, Dag IK Sjøberg, and Audris Mockus. 2008. Variability and reproducibility in software engineering: A study of four companies that developed the same system. *IEEE Transactions on Software Engineering* 35, 3 (2008), 407–429.

[4] Sebastian Baltes and Paul Ralph. 2020. Sampling in Software Engineering Research: A Critical Review and Guidelines. *arXiv preprint arXiv:2002.07764* (2020).

[5] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. DeFlaker: Automatically detecting flaky tests. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 433–444.

[6] Adam Brinckman, Kyle Chard, Niall Gaffney, Mihael Hategan, Matthew B Jones, Kacper Kowalik, Sivakumar Kulasekaran, Bertram Ludäscher, Bryce D Mecum, Jarek Nabrzyski, et al. 2019. Computing environments for reproducibility: Capturing the "Whole Tale". *Future Generation Computer Systems* 94 (2019), 854–867.

[7] Robert J Brunner and Edward J Kim. 2016. Teaching data science. *Procedia Computer Science* 80 (2016), 1947–1956.

[8] Haipeng Cai, Ziyi Zhang, Li Li, and Xiaoqin Fu. 2019. A Large-Scale Study of Application Incompatibilities in Android. In *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*.

[9] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C Chu, and Baowen Xu. 2014. Dynamic slicing of Python programs. In *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE, 219–228.

[10] Jürgen Cito, Vincenzo Ferme, and Harald C Gall. 2016. Using Docker containers to improve reproducibility in software and web engineering research. In *International Conference on Web Engineering*. Springer, 609–612.

[11] Conda. 2019. Conda: Package, dependency and environment management for any language. (2019). Retrieved August 23, 2019 from https://docs.conda.io/en/latest/

[12] Michael P Fay and Michael A Proschan. 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Statistics surveys* 4 (2010), 1.

[13] freezegun. 2019. Let your Python tests travel through time. (2019). Retrieved August 23, 2019 from https://pypi.org/project/freezegun/0.1.11/

[14] Jesús M González-Barahona and Gregorio Robles. 2012. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering* 17, 1-2 (2012), 75–89.

[15] Eric Horton and Chris Parnin. 2019. DockerizeMe: automatic inference of environment dependencies for python code snippets. In *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 328–338.

[16] Mary Beth Kery and Brad A Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 147–155.

[17] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 174.

[18] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks—a publishing format for reproducible computational workflows. In *ELPUB*. 87–90.

[19] Donald Ervin Knuth. 1984. Literate programming. *Comput. J.* 27, 2 (1984), 97–111.

[20] Pingfan Kong, Li Li, Jun Gao, Tegawendé F Bissyandé, and Jacques Klein. 2019. Mining Android Crash Fixes in the Absence of Issue- and Change-Tracking Systems. In *The 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019)*.

[21] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F Bissyandé, and Jacques Klein. 2018. Automated Testing of Android Apps: A Systematic Literature Review. *IEEE Transactions on Reliability* (2018).

[22] David Koop and Jay Patel. 2017. Dataflow notebooks: encoding and tracking dependencies of cells. In *9th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2017)*.

[23] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: Automating the Detection of API-related Compatibility Issues in Android Apps. In *The ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*.

[24] The Python Standard Library. 2019. Build-in Functions. (2019). Retrieved August 23, 2019 from https://docs.python.org/3/library/functions.html

[25] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 643–653.

[26] Guillermo Macbeth, Eugenia Razumiejczyk, and Rubén Daniel Ledesma. 2011. Cliff's Delta Calculator: A non-parametric effect size program for two groups of observations. *Universitas Psychologica* 10, 2 (2011), 545–555.

[27] Audris Mockus, Bente Anda, and Dag IK Sjøberg. 2010. Experiences from replicating a case study to investigate reproducibility of software development. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research*.

[28] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. 2017. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 377–389.

[29] Keith O'Hara, Douglas Blank, and James Marshall. 2015. Computational notebooks for AI education. In *The Twenty-Eighth International Flairs Conference*.

[30] Fabio Palomba and Andy Zaidman. 2019. The smell of fear: On the relation between test smells and flaky tests. *Empirical Software Engineering* 24, 5 (2019), 2907–2946.

[31] Hemant D. Pande, William A Landi, and Barbara G. Ryder. 1994. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering* 20, 5 (1994), 385–403.

[32] Jeffrey M. Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature news* 563 (2018), 145–146.

[33] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of Jupyter notebooks. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 507–517.

[34] Ernest Pobee and WK Chan. 2019. AggrePlay: efficient record and replay of multi-threaded programs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 567–577.

[35] PythonClock. 2019. Python 2.7 Countdown. (2019). Retrieved August 23, 2019 from https://pythonclock.org

[36] Min Ragan-Kelley, F Perez, B Granger, T Kluyver, P Ivanov, J Frederic, and M Bussonnier. 2014. The Jupyter/IPython architecture: a unified view of computational research, from interactive exploration to communication and publication.. In *AGU Fall Meeting Abstracts*.

[37] Bernadette M Randles, Irene V Pasquetto, Milena S Golshan, and Christine L Borgman. 2017. Using the Jupyter notebook as a tool for open science: An empirical study. In *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*. IEEE, 1–2.

[38] Mohammed Suhail Rehman. 2019. Towards Understanding Data Analysis Workflows using a Large Notebook Corpus. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 1841–1843.

[39] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated localization for unreproducible builds. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 71–81.

[40] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. [n. d.]. Root Cause Localization for Unreproducible Builds via Causality Analysis over System Call Tracing. ([n. d.]).

[41] Gema Rodríguez-Pérez, Gregorio Robles, and Jesús M González-Barahona. 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm. *Information and Software Technology* 99 (2018), 164–176.

[42] Héctor Rodríguez-Pérez, Tamara Hernández-Beeftink, José M Lorenzo-Salazar, José L Roda-García, Carlos J Pérez-González, Marcos Colebrook, and Carlos Flores. 2019. NanoDJ: a Dockerized Jupyter notebook for interactive Oxford Nanopore MinION sequence manipulation and genome assembly. *BMC bioinformatics* 20, 1 (2019), 234.

[43] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. 2018. Ten simple rules for reproducible research in Jupyter notebooks. *arXiv preprint arXiv:1810.08055* (2018).

[44] Adam Rule, Amanda Birmingham, Cristal Zuniga, Ilkay Altintas, Shih-Cheng Huang, Rob Knight, Niema Moshiri, Mai H Nguyen, Sara Brin Rosenthal, Fernando Pérez, et al. 2019. Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks. (2019).

[45] Adam Rule, Ian Drosos, Aurélien Tabard, and James D Hollan. 2018. Aiding collaborative reuse of computational notebooks with annotated cell folding. *Proceedings of the ACM on Human-Computer Interaction* 2, CSCW (2018), 150.

[46] Adam Rule, Aurélien Tabard, and James D Hollan. 2018. Exploration and explanation in computational notebooks. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 32.

[47] Onur Sahin, Assel Aliyeva, Hariharan Mathavan, Ayse Coskun, and Manuel Egele. 2019. RANDR: Record and Replay for Android Applications via Targeted Runtime Instrumentation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 128–138.

[48] Sheeba Samuel and Birgitta König-Ries. 2018. ProvBook: Provenance-based Semantic Enrichment of Interactive Notebooks for Reproducibility.. In *International Semantic Web Conference (P&D/Industry/BlueSky)*.

[49] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature News* 515, 7525 (2014), 151.

[50] Diomidis Spinellis. 2003. The decay and failures of web references. *Commun. ACM* 46, 1 (2003), 71–77.

[51] Dan Toomey. 2017. *Jupyter for data science: Exploratory analysis, statistical modeling, machine learning, and data visualization with Jupyter*. Packt Publishing Ltd.

[52] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks. In *The 42nd International Conference on Software Engineering, NIER Track (ICSE 2020)*.

[53] Alex Watson, Scott Bateman, and Suprio Ray. 2019. PySnippet: Accelerating Exploratory Data Analysis in Jupyter Notebook through Facilitated Access to Example Code.. In *EDBT/ICDT Workshops*.

[54] Dasarath Weeratunge, Xiangyu Zhang, and Suresh Jagannathan. 2010. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proceedings of the fifteenth International Conference on Architectural support for programming languages and operating systems*. 155–166.

[55] Ziv Yaniv, Bradley C Lowekamp, Hans J Johnson, and Richard Beare. 2018. SimpleITK image-analysis notebooks: a collaborative environment for education and reproducible research. *Journal of digital imaging* 31, 3 (2018), 290–303.