# Fine-Grained Lineage for Safer Notebook Interactions
[Technical Report]

Stephen Macke[1,2] Hongpu Gong[2] Doris Jung-Lin Lee[2] Andrew Head[2]
Doris Xin[2] Aditya Parameswaran[2]
[1]University of Illinois (UIUC) [2]UC Berkeley
{smacke,ruiduoray,dorislee,andrewhead,dorx,adityagp}@berkeley.edu

## ABSTRACT

Computational notebooks have emerged as the platform of choice for data science and analytical workflows, enabling rapid iteration and exploration. By keeping intermediate program state in memory and segmenting units of execution into so-called "cells", notebooks allow users to execute their workflows interactively and enjoy particularly tight feedback. However, as cells are added, removed, reordered, and rerun, this hidden intermediate state accumulates in a way that is not necessarily correlated with the notebook's visible code, making execution behavior difficult to reason about, and leading to errors and lack of reproducibility. We present NBSAFETY, a custom Jupyter kernel that uses runtime tracing and static analysis to automatically manage lineage associated with cell execution and global notebook state. NBSAFETY detects and prevents errors that users make during unaided notebook interactions, all while preserving the flexibility of existing notebook semantics. We evaluate NBSAFETY's ability to prevent erroneous interactions by replaying and analyzing 666 real notebook sessions. Of these, NBSAFETY identified 117 sessions with potential safety errors, and in the remaining 549 sessions, the cells that NBSAFETY identified as resolving safety issues were more than 7× more likely to be selected by users for re-execution compared to a random baseline, even though the users were not using NBSAFETY and were therefore not influenced by its suggestions.

## 1. INTRODUCTION

Computational notebooks such as Jupyter [26] provide a flexible medium for developers, scientists, and engineers to complete programming tasks interactively. Notebooks, like simpler predecessor read-eval-print-loops (REPLs), do not terminate after executing, but wait for the user to give additional instructions while keeping intermediate programming state in memory. Notebooks, however, are distinguished from REPLs by three key features:

1. The atomic unit of execution in notebooks is the *cell*, composed of a sequence of one or more programming statements, rather than a single programming statement;
2. Notebooks allow users to easily refer back to previous cells to make edits and potentially re-execute; and
3. Notebooks typically allow code and documentation to be interspersed, following the *literate programming* [27] paradigm.

As a result, the IPython Notebook project [40], and its successor, Project Jupyter [26], have both grown rapidly in popularity. These projects decouple the server-side *kernel* (responsible for running user code) from a browser-accessible client side (providing the user interface). Since Jupyter uses familiar web technologies to implement the layer that facilitates communication between the UI and the kernel, it crucially allows users to run notebooks on any platform. Furthermore, Jupyter's decoupled architecture allows users to leverage powerful server computing resources from modest client-side hardware, particularly useful for coping with the ever-increasing scale of modern dataset sizes.

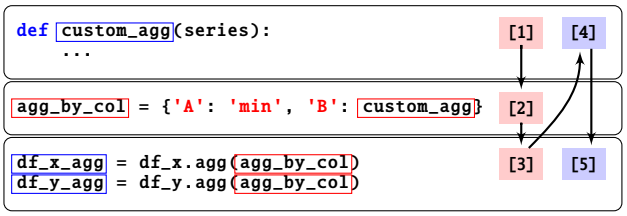



**Figure 1:** Example sequence of notebook interactions leading to a stale symbol usage. Symbols with timestamps $\leq 3$ are shown with a blue border, while symbols with timestamps $> 3$ are shown with a red border.

These key features, along with the natural ergonomics of interactive computing offered by the notebook interface, have led to an explosion in Jupyter's usage. With more than 4.7 million notebooks on GitHub as of March 2019 [44] and with hosted solutions offered by a plethora of data analytics companies, Jupyter has been called "data scientists' computational notebook of choice" [34], and was recognized by the ACM Software System award in 2018 [33]. We focus on Jupyter here due to its popularity, but we note that our ideas are applicable to computational notebooks in general.

Despite the tighter feedback enjoyed by users of computational notebooks, and, in particular, by users of Jupyter, notebooks have a number of drawbacks when used for more interactive and exploratory analysis. Compared to conventional programming environments, interactions such as *out-of-order cell execution*, *cell deletion*, and *cell editing and re-execution* can all complicate the relationship between the code visible on screen and the resident notebook state. Managing interactions with this hidden notebook state is thus a burden shouldered by users, who must remember what they have done in the past, since these past interactions cannot in general be reconstructed from what is on the screen.

**Illustration.** Consider the sequence of notebook interactions depicted in Figure 1. Each rectangular box indicates a cell, the notebook's unit of execution. The user first defines a custom aggregation function that, along with `min`, will be applied to two dataframes, `df_x` and `df_y`, and executes it as cell `[1]`. Since both aggregations will be applied to both dataframes, the user next gathers them into a function dictionary in the second cell (executed as cell `[2]`). After running the third cell, which corresponds to applying the aggregates to `df_x` and `df_y`, the user realizes an error in the logic of `custom_agg` and goes back to the first cell to fix the bug. They re-execute this cell after making their update, indicated as `[4]`. However, they forget that the old version of `custom_agg` still lingers in the `agg_by_col` dictionary and rerun the third cell (indicated as `[5]`) without rerunning the second cell. We deem this an *unsafe* execution, because the user intended for the change to `agg_by_col` to be reflected in `df_agg_x` and `df_agg_y`, but it was not. Upon inspecting the resulting dataframes `df_x_agg` and `df_y_agg`, the user may or may not realize the error. In the best case, user may identify the error and rerun the second cell. In the worst case, users may be deceived into thinking that their change had no effect, with the original error then propagating throughout the notebook.

This example underscores the inherent difficulty in manually managing notebook state, inspiring colorful criticisms such as a talk titled "I Don't Like Notebooks" presented at JupyterCon 2018 [17]. In addition to the frustration that users experience when spending valuable time debugging state-related errors, such bugs can lead to invalid research results and hinder reproducibility, inspiring the claim that one must "restart and run all or it didn't happen" [34] if presenting results via a notebook medium.

**Key Research Challenges.** The goal of this paper is to *develop techniques to automatically identify and prevent potentially unsafe cell executions*, without sacrificing existing familiar notebook semantics. We encounter a number of challenges toward this end:

*1. Automatically detecting unsafe interactions.* To detect unsafe interactions due to symbol staleness issues, the approach that immediately suggests itself is static code analysis, but upon deeper reflection, it becomes clear that static analysis on its own is not enough. A static approach must necessarily be overly conservative when gathering lineage metadata / inferring dependencies, as it must consider all branches of control flow. On the other hand, *some* amount of static analysis is necessary so that users can be warned before they execute an unsafe cell (as opposed to during cell execution, by which time the damage may already be done); finding the right balance is nontrivial.

*2. Automatically resolving unsafe behavior with suggested fixes.* In addition to detecting potentially unsafe interactions, we should ideally also identify which cells to run in order to resolve staleness issues. A simpler approach may be to automatically rerun cells when a potential staleness issue is detected (as in Dataflow notebooks [28]), but in a flexible notebook environment, there could potentially be more than one cell whose re-executions would all resolve a particular staleness issue; identifying these to present them as options to the user requires a significant amount of nontrivial static analysis.

*3. Maintaining interactive levels of performance.* We must address the aforementioned challenges without introducing unacceptable latencies or memory usage. First, we must ensure that any lineage metadata we introduce does not grow too large in size. Second, efficiently identifying cells that resolve staleness issues is also nontrivial. Suppose we are able to detect cells with staleness issues, and we have detected such issues in cell $c_s$. We can check whether prepending some cell $c_r$ (and thereby executing $c_r$ first before $c_s$) would fix the staleness issue (by, e.g., detecting whether the merged cell $c_r \oplus c_s$ has staleness issues), but we show in Section 5.2 that a direct implementation of this idea scales quadratically in the number of cells in the notebook and therefore quickly loses viability. Developing an efficient approach is thus a major challenge.

Despite previous attempts to address these challenges and to facilitate safer interactions with global notebook state [28, 47, 10], to our knowledge, NBSAFETY is the first to do so while preserving the flexibility of existing notebook semantics. For example, Dataflow notebooks [28] require users to explicitly annotate cells with their dependencies, and force the re-execution of cells whose dependencies have changed. Nodebook [47] and the Datalore kernel [10] attempt to enforce a temporal ordering of variable definitions in the order that cells appear, again forcing users to compromise on flexibility. In the design space of computational notebooks [29], Dataflow notebooks observe *reactive* execution order, while Nodebook and Datalore's kernel observe *forced in-order* execution. However, a solution that preserves *any-order* execution semantics, while simultaneously helping users avoid errors that are only made possible due to such flexibility, has heretofore evaded development.

**Contributions.** To address these challenges, we develop NBSAFETY, a custom Jupyter kernel and frontend for automatically detecting unsafe interactions and alerting users, all while maintaining interactive levels of performance and preserving existing notebook semantics. Installing NBSAFETY is easy: after running two installation commands [1], users of both JupyterLab and traditional Jupyter notebooks can opt to use the NBSAFETY kernel as a drop-in replacement for Jupyter's built-in Python 3 kernel. NBSAFETY introduces two key innovations to address the challenges outlined above:

*1. Efficient and accurate detection of staleness issues in cells via novel joint dynamic and static analysis.* The NBSAFETY kernel combines runtime tracing with static analysis in order to detect and prevent notebook interactions that are unsafe due to staleness issues of the form seen in Figure 1. The tracer (§3) instruments each program statement so that program variable definitions are annotated with parent dependencies and cell execution timestamps. This metadata is then used by a *runtime state-aware static checker* (§4) that combines said metadata with static program analysis techniques to determine whether any staleness issues are present *prior* to the start of cell execution. This allows NBSAFETY to present users with *cell highlights* (§5) that warn them about cells that are unsafe to execute due to staleness issues *before* they try executing such cells, thus preserving desirable atomicity of cell executions present in traditional notebooks.

*2. Efficient resolution of staleness issues.* Beyond simply detecting staleness issues, we also show how to detect cells whose re-execution would resolve such staleness issues — but doing so *efficiently* required us to leverage a lesser-known analysis technique called *initialized variable analysis* (§4.3) tailored to this use case. We show how initialized analysis brings staleness resolution complexity down from time quadratic in the number of cells in the notebook to linear, crucial for large notebooks.

We validate our design choices for NBSAFETY by replaying and analyzing of a corpus of 666 execution logs of real notebook sessions, scraped from GitHub (§6). In doing so, NBSAFETY identified that 117 sessions had potential safety errors, and upon sampling these for manual inspection, we found several with particularly egregious examples of confusion and wasted effort by real users that would have been saved with NBSAFETY. After analyzing the 549 remaining sessions, we found that cells suggested by NBSAFETY as resolving staleness issues were strongly favored by users for re-execution—more than 7× more likely to be selected compared to random cells, even though these user interactions were originally performed without NBSAFETY and therefore were not influenced by its suggestions. Overall, our empirical study indicates that NBSAFETY can reduce cognitive overhead associated with manual maintenance of global notebook state under any-order execution semantics, and in doing so, allows users to focus their efforts more on the already challenging task of exploratory data analysis, and less on avoiding and fixing state-related notebook bugs.

Our free and open source code is available publicly on GitHub [31].

**Organization.** Section 2 gives a high-level overview of NBSAFETY's architecture and how it integrates into a notebook workflow. The next three sections drill into each of NBSAFETY's components: Section 3 describes how the tracer maintains lineage metadata, Section 4 describes the static analyses employed by the checker, and Section 5 describes how these two components feed into the frontend in order help users avoid and resolve safety issues. We empirically validate NBSAFETY's ability to highlight (i) cells that should likely be avoided and (ii) cells that should likely be re-executed in Section 6 before surveying related work and concluding (§7, §8).

## 2. ARCHITECTURE OVERVIEW

In this section, we give an overview of NBSAFETY's components and how they integrate into the notebook workflow.

We now describe the operation of each component in more detail.

**Overview.** NBSAFETY integrates into a notebook workflow according to Figure 2. As depicted, all components of NBSAFETY are invoked upon each and every cell execution. When the user submits a request to run a cell, the tracer (❶, §3) instruments the executed cell, updating lineage metadata associated with each variable as each line executes. Once the cell finishes execution, the checker (❷, §4) performs *liveness analysis* [2] (a standard technique for finding non-redefined symbols in

[1]
```
pip install nbsafety                                #1
jupyter labextension install jupyterlab-nbsafety    #2
```

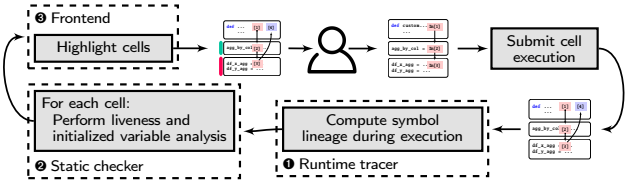



**Figure 2:** NBSAFETY workflow with architectural components.

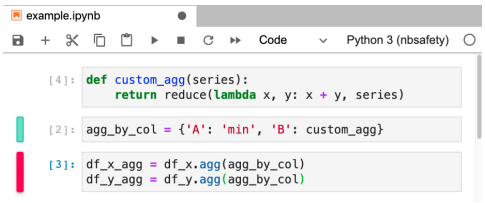



**Figure 3:** NBSAFETY highlights cells that are unsafe to execute with **staleness warnings** and cells that resolve such issues with **cleanup suggestions**.

a program) and *initialized variable analysis* [32] (discussed in §4.3) for every cell in the notebook. By combining the results of these analyses with the lineage metadata computed by the tracer, the frontend (❸, §5) is able to highlight cells that are unsafe due to staleness issues of the form seen in Figure 1, as well as cells that resolve such staleness issues.

**❶ Tracer.** The NBSAFETY tracer maintains dataflow dependencies for each symbol that appears in the notebook in the form of lineage metadata. It leverages Python's built-in tracing capabilities [42],which allows it to run custom code upon four different kinds of events: (i) `line` events, when a line starts to execute; (ii) `call` events, when a function is called, (iii) `return` events, when a function returns, and (iv) `exception` events, when an exception occurs.

To illustrate its operation, consider that, the first time $c_3$ in Figure 1 is executed, symbols `df_agg_x` and `df_agg_y` are undefined. Before the first line runs, a `line` event occurs, thereby trapping into the tracer. The tracer has access to the line of code that triggered the `line` event and parses it as an `Assign` statement in Python's grammar, followed by a quick static analysis to determine that the symbols `df_x` and `agg_by_col` appear on the right hand side of the assignment (i.e., these symbols appear in USE[R.H.S. of the `Assign`]). Thus, these two will be the dependencies for symbol `df_agg_x`. Since $c_3$ is the third cell executed, the tracer furthermore gives `df_agg_x` a timestamp of 3. Similar statements hold for `df_agg_y` once the second line executes.

**❷ Checker.** The NBSAFETY static checker performs two kinds of program analysis: (i) *liveness analysis*, and (ii) *initialized variable analysis*. The NBSAFETY liveness checker helps to detect safety issues by determining which cells have live references to stale symbols. For example, in Figure 1, `agg_by_col`, which is stale, is live in $c_3$—this information can be used to warn the user before they execute $c_3$. Furthermore, the initialized checker serves as a key component for efficiently computing resolutions to staleness issues, as we later show in Sections 4 and 5.

**❸ Frontend.** The NBSAFETY frontend uses the results of the static checker to highlight cells of interest. For example, in Figure 3, which depicts the original example from Figure 1 (but before the user submits $c_3$ for re-execution), $c_3$ is given a **staleness warning** highlight to warn the user that re-execution could have incorrect behavior due to staleness issues. At the same time, $c_2$ is given a **cleanup suggestion** highlight, because rerunning it would resolve the staleness in $c_3$. The user can then leverage the extra visual cues to make a more informed decision about which cell to next execute, possibly preferring to execute cells that resolve staleness issues before cells with staleness issues.

**Design Philosophy.** In contrast with systems that automatically resolve staleness, such as Datalore [10] or Nodebook [47], NBSAFETY is designed to be as non-intrusive as possible, while providing useful information. NBSAFETY thus only attempts to make a "passive observer" guarantee to ensure that existing notebook semantics are preserved, via tracing and static analyses that monitor, and do not alter, notebook behavior. We make this decision partially because program analysis in a dynamic language like Python is notoriously difficult; as such, NBSAFETY's analysis components are unable to make any formal guarantees regarding soundness or completeness, instead opting for a "best-effort" attempt to provide the user with useful information.

Although the guarantee NBSAFETY offers regarding preservation of semantics may seem weak at first glance, we show later (§6.5) that both Datalore and Nodebook crash on reasonable programs due to their intrusiveness. Furthermore, we also show how existing semantics lend themselves to multiverse analyses [41], and point to specific examples of such analyses in our experiments. Finally, NBSAFETY is not necessarily intended to be a substitute for systems like Dataflow notebooks [28] that allow users to give names to cell outputs and explicitly reference them; the two techniques can complement each other, as the issues faced by notebook users do not automatically disappear by using Dataflow notebooks.

Overall, each of NBSAFETY's three key components play crucial roles in helping users avoid and resolve unsafe interactions due to staleness issues without compromising existing notebook program semantics. We describe each component in detail in the following sections.

# 3. LINEAGE TRACKING

In this section, we describe how NBSAFETY traces cell execution in order to maintain symbol lineage metadata, and how such metadata aids in the detection and resolution of staleness issues. We begin by introducing helpful terminology and formalizing our notion of staleness beyond the intuition we gave in Figure 1 of Section 1.

## 3.1 Preliminaries

We begin defining our use of the term *symbol*.

**Definition 1** [Symbol]**.** *A* symbol *is any piece of data in notebook scope that can be referenced by a (possibly qualified) name.*

For example, if `lst` is a list with 10 entries, then `lst`, `lst[0]`, and `lst[8]` are all symbols. Similarly, if `df` is a dataframe with a column named "col", then `df` and `df.col` are both symbols. Symbols can be thought of as a generalized notion of variables that allow us treat different nameable objects in Python's data model in a unified manner.

NBSAFETY augments each symbol with additional lineage metadata in the form of *timestamps* and *dependencies*.

**Definition 2** [Timestamp]**.** *A symbol's* timestamp *is the execution counter of the cell that most recently modified that symbol. Likewise, a cell's timestamp is the execution counter corresponding to the most recent time that cell was executed.*

For a symbol `s` or a cell $c$, we denote its timestamp as $\texttt{ts}(\texttt{s})$ or $\texttt{ts}(c)$, respectively. For example, letting $c_1$, $c_2$, and $c_3$ denote the three cells in Figure 1, we have that $\texttt{ts}(\texttt{custom\_agg}) = \texttt{ts}(c_1) = 4$, since `custom_agg` is last updated in $c_1$, which was executed at time 4.

**Definition 3** [Dependencies]**.** *The* dependencies *of symbol* `s` *are the set of symbols that contributed to the computation of* `s` *via direct dataflow.*

In Figure 1, `agg_by_col` depends on `custom_agg`, while `df_x_agg` depends on `df_x` and `custom_agg`. We write $\texttt{Par}(\texttt{s})$ to denote the dependencies of `s`. Similarly, if $\texttt{t} \in \texttt{Par}(\texttt{s})$, then $\texttt{s} \in \texttt{Chd}(\texttt{t})$.

A major contribution of NBSAFETY is to highlight cells with unsafe usages of *stale symbols*, which we define recursively as follows:

**Definition 4** [Stale symbols]**.** *A symbol* `s` *is called* stale *if there exists some* $\texttt{s}' \in \texttt{Par}(\texttt{s})$ *such that* $\texttt{ts}(\texttt{s}') > \texttt{ts}(\texttt{s})$*, or* $\texttt{s}'$ *is itself stale; that is,* `s` *has a parent that is either itself stale or more up-to-date than* `s`.

In Figure 1, symbol `agg_by_col` is stale, because $\texttt{ts}(\texttt{agg\_by\_col}) = 2$, but $\texttt{ts}(\texttt{custom\_agg}) = 4$. Staleness gives us a rigorous conceptual framework upon which to study the intuitive notion that, because `custom_agg` was updated, we should also update its child `agg_by_col` to prevent counterintuitive behavior.

| AST Node | Example | Rule |
|---|---|---|
| Assign | `a = e` | Par(a) = USE[e] |
| (target in RHS) | `a = a + e` | Par(a) = Par(a) ∪ USE[e] |
| Assign with Call | `a = f(e)` | Par(a) = USE[e] ∪ RET[f] |
| AugAssign | `a += e` | Par(a) = Par(a) ∪ USE[e] |
| For | `for a in e:` | Par(a) = USE[e] |
| FunctionDef | `def f(a=e):` | Par(f) = USE[e] |
| ClassDef | `class c(e):` | Par(c) = USE[e] |

**Table 1:** Subset of lineage rules used by the NBSAFETY tracer.

We now draw on these definitions as we describe how NBSAFETY maintains lineage metadata for symbols while tracing cell execution.

## 3.2 Lineage Update Rules

NBSAFETY attempts to be non-intrusive when maintaining lineage with respect to the Python objects that comprise the notebook's state. To do so, we avoid modifying the Python objects created by the user, instead creating "shadow" references to each symbol. NBSAFETY then takes a hybrid dynamic / static approach to updating each symbol's lineage. After each Python statement has finished executing, the tracer inspects the AST node for the executed statement and performs a lineage update according to the rules shown in Table 1.

**Example.** Suppose the statement

```
gen = map(lambda x:  f(x), foo + [bar])
```

has just finished executing. Using rule 1 of Table 1, the tracer will then statically analyze the right hand side in order to determine

```
USE[map(lambda x:  f(x), foo + [bar])]
```

which is the set of used symbols that appear in the RHS. In this case, the aforementioned set is {`f, foo, bar`} — everything else is either a Python built-in (`map`, `lambda`), or an unbound symbol (i.e. in the case of the lambda argument `x`). The tracer will thus set

$$\text{Par}(\texttt{gen}) = \{\texttt{f},\texttt{foo},\texttt{bar}\}$$

and will also set $\text{ts}(\texttt{gen})$ to the current cell's execution counter.

**Handling Function Calls and Returns.** Recall that, in Python, a function may "capture" symbols defined in external scope by referencing them. In this case, the lineage update rule for a function call needs to be aware of the symbols referenced by the function's return statement. As such, the NBSAFETY tracer saves these symbols when it encounters a return statement, and loads them upon encountering a `return` event, so that they are available for use with lineage update rules, e.g., the third entry of Table 1.

**Rationale for Tracing.** Given that the tracer is already performing some static analysis as each Python statement executes, a natural question is: why should we trace cell execution at all, instead of performing static analysis on an entire cell in order to make lineage updates? The answer is that, when a cell executes, a relatively small number of control flow paths may be taken, and a purely static approach must consider them all in order to be conservative. This may cause each $\text{Par}(\texttt{s})$ to be much larger than necessary, or to be unnecessarily overwritten if, e.g., `s` is assigned in some unexecuted control flow path. This can happen due to, e.g., untaken branches, or if an exception is thrown mid-cell. Indeed, we attempted to infer lineage updates statically in an earlier version of NBSAFETY, but found this to be too coarse-grained in order to derive real benefits.

**Staleness Propagation.** We already saw that the tracer annotates each symbol's shadow reference with timestamp and lineage metadata. Additionally, it tracks whether each symbol is stale, as this cannot be inferred solely from timestamp and lineage metadata. To see why, recall the definition of staleness: a symbol `s` is stale if it has a more up-to-date parent (i.e., an $\texttt{s}' \in \text{Par}(\texttt{s})$ with $\text{ts}(\texttt{s}') > \text{ts}(\texttt{s})$), *or if it has a stale parent*, precluding the ability to determine staleness locally. Thus, when `s` is updated, we perform a depth first search starting from each child $\texttt{c} \in \text{Chd}(\texttt{s})$ in order to propagate the "staleness" to all descendants.

**Fine-Grained Lineage for Attributes and Subscripts.** Recall that the NBSAFETY tracer attempts to infer parent symbols statically when making lineage updates. While this is possible in many cases, there are limits to a purely static approach. For example, the statement "`s = a.b().c`" is valid Python code, but, in general, it is impossible to statically determine what `a.b()` returns. As such, we must again rely on tracing to do so. Unfortunately, Python's built-in tracing abilities operate at the granularity of code lines, so that a `return` event does not tell us the point within a line to which control returns. Thus, the tracer rewrites all `Attribute` nodes in the statement's AST, so that the earlier example will actually run as follows:

```
s = trace(trace(a, 'b').b(), 'c').c
```

The `trace` function will first determine that the symbol name `b` is referenced within symbol `a`'s namespace, setting the current RHS parent symbol to `a.b`. However, after the call event, the tracer will update this to the `c` symbol in the namespace of the return value of `a.b()`, so that the true parent symbol is pinpointed.

Subscripts are handled analogously to attributes.

**Handling External Libraries.** NBSAFETY assumes that imported libraries do not have access to notebook state. Thus, when the tracer observes a call into library code, it simply halts tracing, resuming once control returns to the notebook.

**Bounding Lineage Overhead.** Consider the following cell:

```
x = 0
for i in random.sample(range(10**7), 10**5) + [42]:
    x += lst[i]
```

In order to maintain lineage metadata for symbol `x` to 100% correctness, we would need to somehow indicate that $\text{Par}(\texttt{x})$ contains `lst[i]` for all $10^5$ random indices `i`(as well as for `lst[42]`). It is impossible to maintain acceptable performance in general under these circumstances. Potential workarounds include *conservative* approximations, as well as *lossy* approximations. For example, as a conservative approximation, we could instead specify that `x` depends on `lst`, with the implication that it also depends on everything in `lst`'s namespace. However, this will cause `x` to be incorrectly classified as stale whenever `lst` is mutated, e.g., if a new entry is appended.

Thus, NBSAFETY makes a compromise and sacrifices some correctness by *only instrumenting each AST statement the first time it executes*. In this case, after the cell executes, $\text{Par}(\texttt{x})$ will have a single entry: `lst[0]`. This helps to ensure that lineage metadata only grows in proportion to the amount of text in the user's notebook.

Note that this approach can lead to some false negatives when determining which symbols are stale. In the example above, if the user makes a point update to, e.g., `lst[42]`, `x` should technically be considered stale, but will fail to be updated as so since $\texttt{lst[42]} \notin \text{Par}(\texttt{x})$. We consider this tradeoff worthwhile, as we found that performance suffered greatly without it.

**Garbage Collection.** Each symbol's shadow metadata maintains a weak reference to the symbol. When a Python object is deleted, e.g., because the user executed a `del` statement, or because it was garbage collected by Python's built-in garbage collector, a callback associated with the weak reference executes. This callback tells the metadata to delete itself (including from any $\text{Par}(\cdot)$ or $\text{Chd}(\cdot)$ sets), thereby ensuring that lineage metadata does not accumulate without bound over long notebook sessions.

**Handling Aliases.** Since multiple symbols can reference the same object in Python, we need a mechanism to propagate mutations to some object to all symbols that reference it. For example, if both x and y refer to the same list, and then this list is appended to, both x and y should be have their timestamps bumped. NBSAFETY accomplishes this by creating a registry that maps objects to all symbols referencing them.

**Algorithm 1:** Liveness checker

**Input:** Cell $c$, CFG $G$ that stores successors succ[$s$] for each `stmt` $s \in c$
**Output:** LIVE($c$)

```
foreach stmt s ∈ c do
    LIVE_out[s] ← ∅;
    LIVE_in[s] ← ∅;
    USE[s] ← {referenced symbols in s};
    DEF[s] ← {assigned symbols or defined functions in s};
end
LIVE_in[G_exit] ← ∅;
repeat
    foreach stmt s ∈ c do
        LIVE'_out[s] ← LIVE_out[s];
        LIVE'_in[s] ← LIVE_in[s];
        LIVE_out[s] ← ⋃_{s' ∈ succ[s]} LIVE_in[s'];
        LIVE_in[s] ← USE[s] ∪ (LIVE_out[s] − DEF[s]);
    end
until LIVE_*[s] = LIVE'_*[s] for both * ∈ {in, out}, ∀s ∈ c;
return LIVE_in[G_entry];
```

**Algorithm 2:** Initialized variable checker

**Input:** Cell $c$, CFG $G$ that stores predecessors pred[$s$] for each `stmt` $s \in c$
**Output:** DEAD($c$)

```
foreach stmt s ∈ c do
    DEAD_in[s] ← ∅;
    DEAD_out[s] ← 𝒰;
    USE[s] ← {referenced symbols in s};
    DEF[s] ← {assigned symbols or defined functions in s};
end
DEAD_out[G_entry] ← ∅;
repeat
    foreach stmt s ∈ c do
        DEAD'_in[s] ← DEAD_in[s];
        DEAD'_out[s] ← DEAD_out[s];
        DEAD_in[s] ← ⋂_{s' ∈ pred[s]} DEAD_out[s'];
        DEAD_out[s] ← (DEF[s] − USE[s]) ∪ DEAD_in[s];
    end
until DEAD_*[s] = DEAD'_*[s] for both * ∈ {in, out}, ∀s ∈ c;
return DEAD_out[G_exit];
```

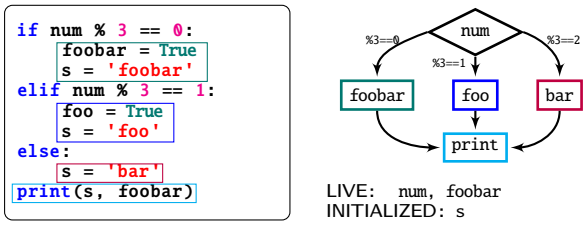



**Figure 4:** Example liveness and initialized variable analysis.

# 4. LIVENESS AND INITIALIZED ANALYSES

In this section, we describe the program analysis component of NBSAFETY's backend. The *checker* performs liveness analysis [2], and a lesser-known program analysis technique called initialized variable analysis, or definite assignment analysis [32]. These techniques are crucial for efficiently identifying which cells are unsafe to execute due to stale references, as well as which cells help resolve staleness issues. We begin with background before discussing the connection between these techniques and staleness detection and resolution.

## 4.1 Background

*Liveness analysis* [2] is a program analysis technique for determining whether the value of a variable at some point is used later in the program. Although traditionally used by compilers to, for example, determine how many registers need to be allocated at some point during program execution, we use it to determine whether a cell has references to stale symbols. We also show (§5) how initialized variable analysis analysis [32], a technique traditionally used by IDEs and linters to detect potentially uninitialized variables, can be used to efficiently determine which cells to run in order to resolve staleness issues.

**Example.** In Figure 4, symbols `num` and `foobar` are live at the top of the cell, since the value for each at the top of the cell can be used in some path of the control flow graph (CFG). In the case of `num`, the (unmodified) value is used in the conditional. In the case of `foobar`, while one path of the CFG modifies it, the other two paths leave it unchanged by the time it is used in the `print` statement; hence, it is also live at the top of the cell. The symbol that is not live at cell start is `foo`, since it is only ever assigned and never used, and `s`, since every path in the CFG assigns to `s`. We call symbols such as `s` that are assigned in every path of the CFG *dead* once they reach the end of the cell.

## 4.2 Cell Oriented Analysis

We now describe how we relate liveness, which is traditionally applied in the context of a single program, to a notebook environment. In brief, we treat each cell as if it is an individual program when performing various program analyses. We formalize these notions below.

**Definition 5** [Live symbols]. *Given a cell $c$ and some symbol* `s`*, we say that* `s` *is* live *in $c$ if there exists some execution path in $c$ in which the value of* `s` *at the start of $c$'s execution is used later in $c$.*

In other words, `s` is live in $c$ if, treating $c$ as a standalone program, `s` is live in the traditional sense at the start of $c$. We already saw in Figure 4 that the live symbols in the example cell are `num`, `fiz`, and `buz`. For a given cell $c$, we use LIVE($c$) to denote the set of all live symbols in $c$.

We are also interested in *dead symbols* that are (re)defined in every branch by the time execution reaches the end of a given cell $c$.

**Definition 6** [Dead symbols]. *Given a cell $c$ and some symbol* `s`*, we say that* `s` *is* dead *in $c$ if, by the time control reaches the end of $c$, every possible path of execution in $c$ overwrites* `s` *in a manner independent of the current value of* `s`.

Denoting such symbols as DEAD($c$), we will see in Section 5 the role they play in assisting in the resolution of staleness issues.

**Staleness and Freshness of Live Symbols in Cells.** Recall that symbols are augmented with additional lineage and timestamp metadata computed by the tracer (§3). We can thus additionally refer to the set STALE($c$) ⊆ LIVE($c$), the set of stale symbols that are live in $c$. When this set is nonempty, we say that cell $c$ itself is stale:

**Definition 7** [Stale cells]. *A cell $c$ is called* stale *if there exists some* `s` ∈ LIVE($c$) *such that* `s` *is stale; i.e., $c$ has a live reference to some stale symbol.*

A major contribution of NBSAFETY is to identify cells that are stale and preemptively warn the user about them.

Note that a symbol can be stale regardless of whether it is live in some cell. Given a particular cell $c$, we can also categorize symbols according to their lineage and timestamp metadata as they relate to $c$. For example, when a non-stale symbol `s` that is live in $c$ is more "up-to-date" than $c$, then we say that it is *fresh* with respect to $c$:

**Definition 8** [Fresh symbols]. *Given a cell $c$ and some symbol* `s`*, we say that* `s` *is* fresh w.r.t. *$c$ if (i)* `s` *is not stale, and (ii)* ts(`s`) > ts($c$).

We can extend the notion of fresh symbols to cells just as we did for stale symbols and stale cells:

**Definition 9** [Fresh cells]. *A cell $c$ is called* fresh *if it (i) it is not stale, and (ii) it contains a live reference to one or more fresh symbols; that is,* ∃`s` ∈ LIVE($c$) *such that* `s` *is fresh with respect to $c$.*

**Example.** Consider a notebook with three cells run in sequence, with code `a=4`, `b=a`, and `c=a+b`, respectively, and suppose the first cell is updated to be `a=5` and rerun. The third cell contains references to `a` and `b`, and although `a` is fresh, `b` is stale, so the third cell is not fresh, but stale. On the other hand, the second cell contains a live reference to `a` but no live references to `b`, and is thus fresh.

As we see in our experiments (§6), fresh cells are oftentimes cells that users wish to re-execute; another major contribution of NBSAFETY

is therefore to automatically identify such cells. In fact, in the above example, rerunning the second cell resolves the staleness issue present in the first cell. That said, running any other cell that assigns to `b` would also resolve the staleness issue, so staleness-resolving cells need not necessarily be fresh. Instead, fresh cells can be thought of as resolving staleness in cell output, as opposed to resolving staleness in some symbol. We study such staleness-resolving cells next.

**Cells that Resolve Staleness.** We have already seen how liveness checking can help users to identify stale cells. Ideally, we should also identify cells whose execution would "freshen" the stale variables that are live in some cell $c$, thereby allowing $c$ to be executed without potential errors due to staleness. We thus define *refresher cells* as follows:

**Definition 10** [Refresher cells]. *A non-stale cell $c_r$ is called* refresher *if there exists some other stale cell $c_s$ such that*

$$\mathsf{STALE}(c_s) - \mathsf{STALE}(c_r \oplus c_s) \neq \emptyset$$

*where $c_r \oplus c_s$ denotes the concatenation of cells $c_r$ and $c_s$. That is, the result of merging $c_r$ and $c_s$ together has fewer live stale symbol references than does $c_s$ alone.*

Intuitively, if we were to submit a refresher cell for execution, we would reduce the number of stale symbols live in some other cell (possibly to 0). Note that a refresher cell may or may not be fresh.

In addition to identifying stale and fresh cells, a final major contribution of NBSAFETY is the *efficient* identification of refresher cells. We will see in Section 5 that scalable computation of such cells requires initialized analysis to computing dead symbols, which we describe in detail next.

## 4.3 Initialized Variable Analysis

Recall that we use liveness analysis to find "live" symbols whose values at the start of each cell contribute to the computation performed in the cell, and we use inverse liveness to find "dead" symbols whose values at the end of the each will have definitely been overwritten by the time control reaches the end of the cell. A working knowledge of traditional liveness analysis is a prerequisite for for understanding our inverse liveness technique; we refer the reader to, e.g., Aho et al. [2] for any review necessary.

**Dataflow Equations.** Inverse liveness is a fixed-point method for solving the following set of dataflow equations:

$$\mathsf{DEAD}_{out}[s] = (\mathsf{DEF}[s] - \mathsf{USE}[s]) \cup \mathsf{DEAD}_{in}[s]$$
$$\mathsf{DEAD}_{in}[s] = \bigcap_{s' \in \text{predecessors of } s} \mathsf{DEAD}_{out}[s']$$

That is, a symbol is dead in statement $s$ (i) if it is defined as a function, (ii) if it appears on the left hand side of an assignment (but not the right hand side), or (iii) if it is dead in all predecessor statements in the control flow graph.

**Intuition.** Traditional liveness analysis initializes each statement at the minimum point of a lattice (i.e., the empty set). Each statement's set of used symbols ($\mathsf{USE}[s]$) are then iteratively propagated in the reverse direction of control until a fixed point is reached, thereby solving the following set of dataflow equations:

$$\mathsf{LIVE}_{in}[s] = \mathsf{USE}[s] \cup (\mathsf{LIVE}_{out}[s] - \mathsf{DEF}[s])$$
$$\mathsf{LIVE}_{out}[s] = \bigcup_{s' \in \text{successors of } s} \mathsf{LIVE}_{in}[s']$$

If the live variables propagated during liveness checking can be thought of as electrons, then the dead variables propagated during inverse liveness can be thought of as "holes", using a metaphor from electronics. Nearly every decision made over the course of inverse liveness analysis is the "inverse" of decisions made during liveness analysis. For example, in inverse liveness, each statement's set of dead symbols is initialized to *everything* (i.e., at lattice maximum), and inverse liveness uses set intersection ($\cap$) as the lattice meet operator instead of set union ($\cup$) when propagating dead variables between statements, since symbols can be dead at cell bottom only if they are overwritten in *every* branch of control.

**Comparing Liveness and Inverse Liveness.** A pseudocode description of our inverse liveness checker is given in Algorithm 2. We also give a description of a textbook liveness checker to the left in Algorithm 1 for contrast. In particular, we note that the two algorithms are nearly identical excepting a few key differences:

- On line 3, $\mathsf{DEAD}_{out}$ is initialized to *every* symbol, instead of $\emptyset$;
- On line 12 of Algorithm 2, we take the intersection of outgoing dead symbols in predecessor nodes, instead of the union of incoming live symbols in successor nodes as in Algorithm 1;
- On line 13 of Algorithm 2, we compute outgoing dead symbols as the union of symbols killed in the current node with incoming dead symbols, whereas Algorithm 1 computes incoming live symbols as the union of symbols used in the current node with non-killed outgoing live symbols;
- The dataflows is from top to bottom in Algorithm 2, while it is from bottom to top in Algorithm 1.

These differences underscore the role Algorithm 2 plays as an inversion of liveness analysis.

## 4.4 Resolving Live Symbols

In many cases, it is possible to determine the set of live symbols in a cell with high precision purely via static analysis. In some cases, however, it is difficult to do so without awareness of additional runtime data. To illustrate, consider the example below:

```
x = 0                                          [1]
def f(y):
    return x + y
lst = [f, lambda t: t + 1]
```

```
print(lst[1](2))                               [2]
```

Whether or not symbol `x` should be considered live at the top of the second cell depends on whether the call to `lst[1](2)` refers to the list entry containing the lambda, or the entry containing function `f`. In this case, a static analyzer might be able to infer that `lst[1]` does not reference `f` and that `x` should therefore not be considered live at the top of cell 2 (since there is no call to function `f`, in whose body `x` is live), but doing so in general is challenging due to Rice's theorem. Instead of doing so purely statically, NBSAFETY performs an extra resolution step, since it can actually examine the runtime value of `lst[1]` in memory. This allows NBSAFETY to be more precise about live symbols than a conservative approach would be, which would be forced to consider `x` as live even though `f` is not referenced by `lst[1]`.

# 5. CELL HIGHLIGHTS

In this section, we describe how to combine the lineage metadata from Section 3 with the output of NBSAFETY's static checker to highlight cells of interest.

## 5.1 Highlight Abstraction

We begin by defining the notion of *cell highlights* in the abstract before discussing concrete examples, how they are presented, and how they are computed.

**Definition 11** [Cell highlights]. *Given a notebook $N$ abstractly defined as an ordered set of cells $\{c_i\}$, a set of* cell highlights *$\mathcal{H}$ is a subset of $N$ comprised of cells that are semantically related in some way at a particular point in time.*

More concretely, we will consider the following cell highlights:

- $\mathcal{H}_s$, the set of stale cells in a notebook;

**Algorithm 3:** Computing refresher cells naïvely

```
Input:  Notebook N, stale cells H_s ⊆ N
Output: Refresher cells H_r
H_r ← ∅;
foreach c_s ∈ H_s do
    STALE(c_s) ← live, stale symbols in c_s;
    foreach c_r ∈ N − H_s do
        STALE(c_r ⊕ c_s) ← live, stale symbols in c_r ⊕ c_s;
        if STALE(c_s) − STALE(c_r ⊕ c_s) ≠ ∅ then
            H_r ← H_r ∪ {c_r};
        end
    end
end
return H_r;
```

- $\mathcal{H}_f$, the set of fresh cells in a notebook; and
- $\mathcal{H}_r$, the set of refresher cells in a notebook.

Note that these sets of cell highlights are all implicitly indexed by their containing notebook's execution counter. When not clear from context we write $\mathcal{H}_s^{(t)}$, $\mathcal{H}_f^{(t)}$, and $\mathcal{H}_r^{(t)}$ (respectively) to make the time dependency explicit. Along these lines, we are also interested in the following "delta" cell highlights:

- $\Delta\mathcal{H}_f^{(t)} = \mathcal{H}_f^{(t)} - \mathcal{H}_f^{(t-1)}$ (new fresh cells); and
- $\Delta\mathcal{H}_r^{(t)} = \mathcal{H}_r^{(t)} - \mathcal{H}_r^{(t-1)}$ (new refresher cells)

again omitting superscripts when clear from context.

**Interface.** We have already seen from the example in Figure 3 that stale cells are given **staleness warnings** to the left of the cell, and refresher cells are given **cleanup suggestions** to the left of the cell. The current version of NBSAFETY as of this writing (0.0.49) also augments fresh cells with **cleanup suggestions** of the same color as that used for refresher cells. Overall, the fresh and refresher highlights are intended to steer users toward cells that they may wish to re-execute, and the stale highlights are intended to steer users away from cells that they may wish to avoid, intuitions that we validate in our empirical study (§6). Experimenting with presentation techniques for the various sets $\mathcal{H}_*$ is an interesting avenue that we leave to future work.

**Computation.** Computing $\mathcal{H}_s$ and $\mathcal{H}_f$ is straightforward: for each cell $c$, we simply run a liveness checker (Algorithm 1) to determine $\mathsf{LIVE}(c)$, and then perform a metadata lookup for each symbol $\mathtt{s} \in \mathsf{LIVE}(c)$ to determine whether $\mathtt{s}$ is fresh w.r.t. $c$ or stale. The manner in which NBSAFETY computes refresher cells deserves a more thorough treatment that we consider next.

## 5.2 Computing Refresher Cells Efficiently

Before we discuss how NBSAFETY uses the initialized variable checker from Section 4 to efficiently compute refresher cells, consider how one might design an algorithm to compute refresher cells directly from Definition 10. The straightforward way is to loop over all non-stale cells $c_r \in N - \mathcal{H}_s$ and compare whether $\mathsf{STALE}(c_r \oplus c_s)$ is smaller than $\mathsf{STALE}(c_s)$. In the case that $\mathcal{H}_s$ and $N - \mathcal{H}_s$ are similar in size, this requires performing $O(|N|^2)$ liveness analyses, which would create unacceptable latency in the case of large notebooks. This inefficient approach is depicted in Algorithm 3.

By leveraging an initialized variable checker, it turns out that we can check whether $\mathsf{STALE}(c_s)$ and $\mathsf{DEAD}(c_r)$ have any overlap instead of performing liveness analysis over $c_r \oplus c_s$ and checking whether $\mathsf{STALE}(c_r \oplus c_s)$ shrinks. We state this formally as follows:

**Theorem 1.** *Let $N$ be a notebook, and let $c_s \in \mathcal{H}_s \subseteq N$. For any other $c_r \in N - \mathcal{H}_s$, the following equality holds:*

$$\mathsf{STALE}(c_s) - \mathsf{STALE}(c_r \oplus c_s) = \mathsf{DEAD}(c_r) \cap \mathsf{STALE}(c_s)$$

*Proof.* We show each side of the equality is a subset of the other side. First, suppose some stale symbol $\mathtt{x}$ is live in $c_s$ but not in $c_r \oplus c_s$. Then, at the point where control transfers from $c_r$ to $c_s$ in the outermost scope, every path of execution will definitely have redefined $\mathtt{x}$.[2] Otherwise, there would exist a path in $c_r$ wherein $\mathtt{x}$ is not redefined, and because $\mathtt{x}$ is live at the top of $c_s$, it would also be live at the top of $c_r \oplus c_s$. As such, $\mathtt{x} \in \mathsf{DEAD}(c_r)$ by definition. Furthermore, $\mathtt{x} \in \mathsf{STALE}(c_s)$ by our initial assumption, so $\mathtt{x} \in \mathsf{DEAD}(c_r) \cap \mathsf{STALE}(c_s)$.

Conversely, suppose some stale symbol $\mathtt{x}$ is live in $c_s$ but dead in $c_r$. By definition, every path of execution in $c_r$ redefines $\mathtt{x}$. We would like to say that $\mathtt{x}$ is not live in $c_r \oplus c_s$, but deadness in $c_r$ does not preclude liveness in $c_r$ (if, e.g., $\mathtt{x}$ is used in some path of $c_r$ before it is redefined later). Thus, it is only true that $\mathtt{x}$ is not live if $c_r \oplus c_s$ if it is also not live in $c_r$. In fact, $\mathtt{x}$ is not live in $c_r$ because $c_r \notin \mathcal{H}_s$; i.e., $c_r$ has no live stale symbols by assumption, and $\mathtt{x}$ is stale; thus $\mathtt{x}$ is both live in $c_s$ and not live in $c_r \oplus c_s$; i.e., $\mathtt{x} \in \mathsf{STALE}(c_s) - \mathsf{STALE}(c_r \oplus c_s)$, which is what we needed to show to complete the proof. □

**Algorithm 4:** Computing refresher cells efficiently

```
Input:  Notebook N, stale cells H_s ⊆ N,
        stale and live symbols STALE(c_s), ∀c_s ∈ H_s,
        dead symbols DEAD(c_r), ∀c_r ∈ N − H_s
Output: Refresher cells H_r ⊆ N
DEAD^{-1}[s] ← ∅, ∀s ∈ DEAD(c_r), ∀c_r ∈ N − H_s;
foreach c_r ∈ N − H_s do
    foreach s ∈ DEAD(c_r) do
        DEAD^{-1}[s] ← DEAD^{-1}[s] ∪ {c_r};
    end
end
H_r ← ∅;
foreach c_s ∈ H_s do
    foreach s ∈ STALE(c_s) do
        H_r ← H_r ∪ DEAD^{-1}[s];
    end
end
return H_r;
```

Theorem 1 relies crucially on the fact that the CFG of the concatenation of two cells $c_r$ and $c_s$ into $c_r \oplus c_s$ will have a "choke point" at the position where control transfers from $c_r$ into $c_s$, so that any symbols in $\mathsf{DEAD}(c_r)$ cannot be "revived" in $c_r \oplus c_s$.

**Computing $\mathcal{H}_r$ Efficiently.** Contrasted with taking $O(|N|^2)$ pairs $c_s \in \mathcal{H}_s$, $c_r \in N - \mathcal{H}_s$ and checking liveness on each concatenation $c_r \oplus c_s$, Theorem 1 instead allows us compute the set $\mathcal{H}_r$ as

$$\bigcup_{c_s \in \mathcal{H}_s} \bigcup_{\mathtt{s} \in \mathsf{STALE}(c_s)} \{c_r \in N - \mathcal{H}_s : \mathtt{s} \in \mathsf{DEAD}(c_r)\} \tag{1}$$

Equation 1 can be computed efficiently according to Algorithm 4, which creates an inverted index that maps dead symbols to their containing cells ($\mathsf{DEAD}^{-1}$) in order to efficiently compute the inner set union. Furthermore, Algorithm 4 only requires $O(|N|)$ liveness analyses and $O(|N|)$ initialized variable analyses as preprocessing, translating to significant latency reductions in our benchmarks (§6.4).

# 6. EMPIRICAL STUDY

We now evaluate NBSAFETY's ability to highlight unsafe cells, as well as cells that resolve safety issues (refresher cells). We do so by replaying 666 real notebook sessions and measuring how the cells highlighted by NBSAFETY correlate with real user actions. After describing data collection (§6.1) and our evaluation metrics (§6.2), we present our quantitative results (§6.3 and §6.4), followed by a qualitative comparison with other systems informed by real examples from our data (§6.5 and §6.6).

## 6.1 Notebook Session Replay Data

We now describe our data collection and session replay efforts.

**Data Scraping.** The `.ipynb` json format contains a static snapshot of the code present in a computational notebook and lacks explicit interaction data, such as how the code present in a cell evolves, which cells

[2] Note that there is no way for control to transfer from $c_r$ to $c_s$ in the outermost scope except at the point where $c_r$ and $c_s$ meet lexically (technically, $c_r$ could call a function defined in $c_s$, but if this were to occur, control would not be at the outermost scope).

are re-executed, and the order in which cells were executed.[3] Fortunately, Jupyter's IPython kernel implements a history mechanism that includes information about individual cell executions in each session, including the source code and execution counter for every cell execution. We thus scraped `history.sqlite` files from 712 repositories files using GitHub's API [15], from which we successfully extracted 657 such files. In total, these history files contained execution logs for ≈ 51000 notebook sessions, out of which we were able to collect metrics for 666 after conducting the filter and repair steps described next.

**Notebook Session Repair.** Many of the notebook sessions were impossible to replay with perfect replication of the session's original behavior (due to, e.g., missing files). To cope, we adapted ideas from Yan et al. [44] to repair sessions wherever possible. Specifically, we took the following measures:

- Since NBSAFETY runs on Python 3, we used the `2to3` tool [1] whenever we encountered Python 2 code.
- To deal with differing APIs used by different versions of the same library (e.g., `scikit-learn`), we first gathered all the import statements for each library and tried to execute them under different versions of the aforementioned library, using the version that minimized import errors to finally replay the session.
- We normalized all path-like strings to point to the same directory, to prevent invalid accesses to nonexistent directories.
- We used the Kaggle API to search for and attempt to download any `csv` files referenced by each session.
- We removed any lines or cells that attempted to run system commands through Jupyter's line (resp. cell) magic functionality.
- We executed the line magic `%matplotlib inline` before replaying a session to avoid rendering matplotlib charts with `Qt`.

**Session Filtering.** Despite these efforts, we were unable to reconstruct some sessions to their original fidelity due to various environment discrepancies. Furthermore, certain sessions had few cell executions and appeared to be random tinkering. We therefore filtered out sessions fitting any of the following criteria:

- Sessions with fewer than 50 cell executions;
- Sessions that attempted to run shell commands;
- Sessions that solicited user input via `readline` or other means;
- Sessions that attempted to connect to external services (e.g. AWS, Spark, Postgres, MySQL, etc.);
- Sessions that attempted to read nonexistent files (or those that could not be found using the Kaggle API).

After these steps, we were left with 2566 replayable sessions. However, we were unable to gather meaningful metrics on more than half of the sessions we replayed because of exceptions thrown upon many cell executions. We filtered these in post-processing by removing data for any session with more than 50% of cell executions resulting in exceptions.

After the repair and filtration steps, we extracted metrics from a total of 666 sessions. Our repair, filtering, and replay scripts are available on GitHub [30].

**Environment.** All experiments were conducted on a 2019 MacBook Pro with 32 GiB RAM and a Core i9 processor running macOS 10.14.5, Python 3.7, and NBSAFETY 0.0.49. We replayed notebook sessions in a container instance to ensure our local files would not be compromised in the event of intentionally or unintentionally malicious code present in the sessions we scraped.

## 6.2 Metrics

Besides conducting benchmark experiments to measure overhead associated with NBSAFETY (§6.4), the primary goal of our empirical study is to evaluate our system and interface design choices from the previous sections by testing two hypotheses. Our first hypothesis (i) is that *cells with staleness issues highlighted by NBSAFETY are likely to be avoided by real users*, suggesting that these cells are indeed unsafe to execute. Our second hypothesis (ii) is that *fresh and refresher cells highlighted by NBSAFETY are more likely to be selected for re-execution*, indicating that these suggestions can help reduce cognitive overhead for users trying to choose which cells to re-execute. To test these hypotheses, we introduce the notion of *predictive power* for cell highlights.

[3]The cell counter in a `.ipynb` file only contains the latest executed cell version for each cell, and says nothing about how executions of earlier iterations of the cell are ordered w.r.t. others.

| Quantity | $\mathcal{H}_n$ | $\mathcal{H}_{\text{rnd}}$ | $\mathcal{H}_s$ | $\mathcal{H}_f$ | $\mathcal{H}_r$ | $\Delta\mathcal{H}_f$ | $\Delta\mathcal{H}_r$ |
|---|---|---|---|---|---|---|---|
| AVG($\mathcal{P}(\mathcal{H}_*)$) | 2.64 | 1.02 | **0.30** | 2.83 | 3.90 | **9.17** | 6.20 |
| AVG($\lvert\mathcal{H}_*\rvert$) | 1.00 | 1.00 | 2.71 | 3.73 | 2.31 | 1.73 | 1.81 |

**Table 2:** Summary of measurements taken for various highlight sets.

**Definition 12** [Predictive Power]. *Given a notebook $N$ with a total of $|N|$ cells, the id of the next cell executed $c$, and a non-empty set of cell highlights $\mathcal{H}$ (chosen before $c$ is known), the* predictive power *of $\mathcal{H}$ is defined as $\mathcal{P}(\mathcal{H}) = \mathbb{I}\{c \in \mathcal{H}\} \cdot |N|/|\mathcal{H}|$.*

Averaged over many measurements, predictive power assesses how many more times more likely a cell from some set of highlights $\mathcal{H}$ is to be picked for re-execution, compared to random cells.

**Intuition.** To understand predictive power, consider a set of highlights $\mathcal{H}$ chosen uniformly randomly without replacement from the entire set of available cells. In this case,

$$\mathbb{E}[\mathbb{I}\{c \in \mathcal{H}\}] = \mathbb{P}(c \in \mathcal{H}) = |\mathcal{H}|/|N|$$

so that the predictive power of $\mathcal{H}$ is $(|\mathcal{H}|/|N|) \cdot (|N|/|\mathcal{H}|) = 1$. This holds for any number of cells in the set of highlights $\mathcal{H}$, even when $|\mathcal{H}| = |N|$. Increasing the size of $\mathcal{H}$ increases the chance for a nonzero predictive power, but it also decreases the "payout" when $c \in \mathcal{H}$. For a fixed notebook $N$, the maximum possible predictive power for $\mathcal{H}$ occurs when $\mathcal{H} = \{c\}$, in which case $\mathcal{P}(\mathcal{H}) = |N|$.

**Rationale.** Our goal in introducing predictive power is not to give a metric that we then attempt to optimize; rather, we merely want to see how different sets of cell highlights correlate with real user behavior. In some sense, any $\mathcal{P}(\mathcal{H}) \neq 1$ is interesting: $\mathcal{P}(\mathcal{H}) < 1$ indicates that users tend to avoid $\mathcal{H}$, and $\mathcal{P}(\mathcal{H}) > 1$ indicates that users tend to prefer $\mathcal{H}$. For the different sets of cell highlights $\{\mathcal{H}_*\}$ introduced in Section 5, each $\mathcal{P}(\mathcal{H}_*)$ helps us to make this determination.

**Gathering measurements.** The session interaction data available in the scraped history files only contains the submitted cell contents for each cell execution, and unfortunately lacks cell identifiers. Therefore, we attempted to infer the cell identifier as follows: for each cell execution, if the cell contents were ≥80% similar to a previously submitted cell (by Levenshtein similarity), we assigned the identifier of that cell; otherwise, we assigned a new identifier. Whenever we inferred that an existing cell was potentially edited and re-executed, we measured predictive power for various highlights $\mathcal{H}_*$ when such highlights were non-empty. Across the various highlights, we computed the average of such predictive powers for each sessions, and the averaged the average predictive powers across all sessions, reporting the result as AVG($\mathcal{P}(\mathcal{H}_*)$) for each $\mathcal{H}_*$ (§6.3).

**Highlights of Interest.** We gathered metrics for $\mathcal{H}_s$, $\mathcal{H}_f$, $\Delta\mathcal{H}_f$, $\mathcal{H}_r$, and $\Delta\mathcal{H}_r$, which we described earlier in Section 5. Additionally, we also gathered metrics for the following "baseline highlights":

- $\mathcal{H}_n$, or the *next cell highlight*, which contains only the $k+1$ cell (when applicable) if cell $k$ was the previous cell executed; and
- $\mathcal{H}_{\text{rnd}}$, or the *random cell highlight*, which simply picks a random cell from the list of existing cells.

We take measurements for $\mathcal{H}_n$ because picking the next cell in a notebook is a common choice, and it is interesting to see how its predictive power compares with cells highlighted by the NBSAFETY frontend such as $\mathcal{H}_f$ and $\mathcal{H}_r$. We also take measurements for $\mathcal{H}_{\text{rnd}}$ to validate via Monte Carlo simulation the claim that random cells $\mathcal{H}_{\text{rnd}}$ should satisfy $\mathcal{P}(\mathcal{H}_{\text{rnd}}) = 1$ in expectation.

## 6.3 Predictive Power Results

In this section, we present the results of our empirical evaluation. *Overall, NBSAFETY discovered that* 117 *sessions out of the* 666 *encountered*

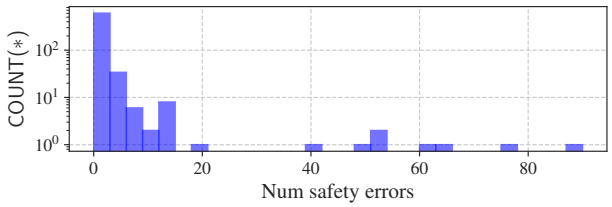



**Figure 5:** Histogram showing distribution of safety errors across sessions.

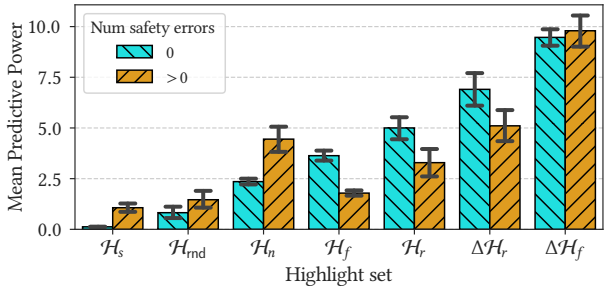



**Figure 6:** Comparing $\mathsf{AVG}(\mathcal{P}(\mathcal{H}_*))$ for sessions with/without safety issues.

*staleness issues at some point*, underscoring the very real need for a tool to prevent such errors. Furthermore, we found that the "positive" highlights like $\mathcal{H}_f$ and $\mathcal{H}_r$ correlated strongly with user choices.

**Predictive Power for Various Highlights.** We now discuss average $\mathcal{P}(\mathcal{H}_*)$ for the various $\mathcal{H}_*$ we consider, summarized in Table 2.

> ***Summary.*** Out of the highlights $\mathcal{H}_*$ with $\mathcal{P}(\mathcal{H}_*) > 1$, new fresh cells, $\Delta\mathcal{H}_f$, had the highest predictive power, while $\mathcal{H}_n$ had the lowest (excepting $\mathcal{H}_{\text{rnd}}$, which had $\mathcal{P}(\mathcal{H}_{\text{rnd}}) \approx 1$ as expected). $\mathcal{H}_s$ had the lowest predictive power coming in at $\mathcal{P}(\mathcal{H}_s) \approx 0.30$, suggesting that users do, in fact, avoid stale cells.

We measured the average value of $\mathcal{P}(\mathcal{H}_s)$ at roughly 0.30, which is the lowest mean predictive power measured out of any highlights. One way to interpret this is that users were more then 3× *less* likely to re-execute stale cells than they are to re-execute randomly selected highlights of the same size as $\mathcal{H}_s$ — strongly supporting the hypothesis that users tend to avoid stale cells.

On the other hand, all of the highlights $\mathcal{H}_n, \mathcal{H}_f, \mathcal{H}_r, \Delta\mathcal{H}_f$, and $\Delta\mathcal{H}_r$ satisfied $\mathcal{P}(\mathcal{H}_*) > 1$ on average, with $\mathcal{P}(\Delta\mathcal{H}_f)$ larger than the others at 9.17, suggesting that users are *more than* 9× *more likely* to select newly fresh cells to re-execute than they are to re-execute randomly selected highlights of the same size as $\Delta\mathcal{H}_f$. In fact, $\mathcal{H}_n$ was the lowest non-random set of highlights with mean predictive power > 1, strongly supporting our design decision of specifically guiding users to all the cells from $\mathcal{H}_f$ and $\mathcal{H}_r$ (and therefore to $\Delta\mathcal{H}_f$ and $\Delta\mathcal{H}_r$ as well) with our aforementioned **visual cues**. Furthermore, we found that no $|\mathcal{H}_*|$ was larger than 4 on average, suggesting that these cues are useful, and not overwhelming.

Finally, given the larger predictive powers of $\Delta\mathcal{H}_f$ and $\Delta\mathcal{H}_r$, we plan to study interfaces that present these highlights separately from $\mathcal{H}_f$ and $\mathcal{H}_r$ in future work.

**Effect of Safety Issues on Predictive Power.** Of the 666 sessions we replayed, we detected 1 or more safety issues (due to the user executing a stale cell) in 117, while the majority (549) did not have safety issues. A histogram depicting the distribution of "# safety issues" is given in Figure 5. We reveal interesting behavior by computing $\mathsf{AVG}(\mathcal{P}(\mathcal{H}_*))$ when restricted to (a) sessions without safety errors, and (b) sessions with 1 or more safety errors, depicted in Figure 6.

> ***Summary.*** For sessions with safety errors, users were more likely to select the next cell ($\mathcal{H}_n$), and less likely to select fresh or refresher cells ($\mathcal{H}_f$ and $\mathcal{H}_r$, respectively).

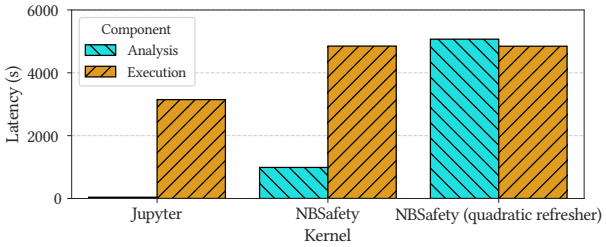



**Figure 7:** Comparing latencies of execution and (if applicable) analysis components for different kernels.

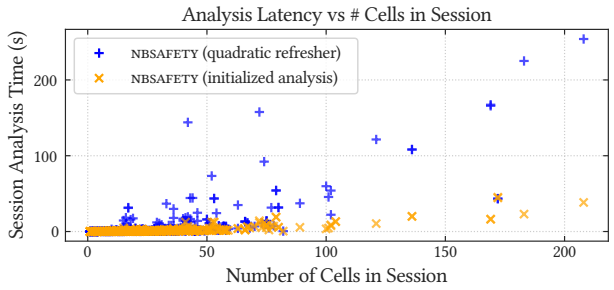



**Figure 8:** Measuring the impact of cell count on analysis latency for NBSAFETY with and without efficient refresher computation.

| Approach | Jupyter | NBSAFETY | NBSAFETY (quadratic refresher) |
|---|---|---|---|
| Total Time (s) | 3150 | 5840 | 9920 |
| Median Slowdown | 1× | 1.44× | 1.58× |

**Table 3:** Summary of latency measurements. Median slowdown measured on sessions that took longer than 5 seconds to execute in vanilla Jupyter.

Figure 6 plots $\mathsf{AVG}(\mathcal{P}(\mathcal{H}_*))$ for various highlight sets after faceting on sessions that did and did not have safety errors. By definition, $\mathsf{AVG}(\mathcal{P}(\mathcal{H}_s)) = 0$ for sessions without safety errors (otherwise, users would have attempted to execute one or more stale cells), but even for sessions with safety errors, we still found $\mathcal{P}(\mathcal{H}_s) < 1$ on average, though not enough to rule out random chance.

Interestingly, we found that $\mathsf{AVG}(\mathcal{P}(\mathcal{H}_n))$ was significantly higher for sessions with safety issues, suggesting that users were more likely to "blindly" execute the next cell without thought.

Finally, we found that users were significantly *less* likely to choose cells from $\mathcal{H}_f, \mathcal{H}_r$, or $\Delta\mathcal{H}_r$ for sessions with safety errors. In fact, users favored $\mathcal{H}_n$ over $\mathcal{H}_r$ or $\mathcal{H}_f$ in this case. Regardless of whether sessions had safety issues, however, $\Delta\mathcal{H}_f$ and $\Delta\mathcal{H}_r$ still had the highest mean predictive powers out of any of the highlights, with $\mathsf{AVG}(\mathcal{P}(\Delta\mathcal{H}_f))$ relatively unaffected by safety issues.

## 6.4 Benchmark Results

Our benchmarks are designed to assess the additional overhead incurred by our tracer and checker by measuring the end-to-end execution latency for the aforementioned 666 sessions, with and without NBSAFETY. Furthermore, we assess the impact of our initialized analysis approach to computing refresher cells (Algorithm 4) by comparing it with the naïve quadratic baseline (Algorithm 3).

**Overall Execution Time.** We summarize the time needed for various methods to replay the 666 sessions in our execution logs in Table 3, and furthermore faceted on the static analysis and tracing / execution components in Figure 7. We measured latencies for both vanilla Jupyter and NBSAFETY, as well as for an ablation that replaces the efficient refresher computation algorithm with the quadratic variant (Algorithm 3).

| System | Datalore | Nodebook | Dataflow | NBSAFETY |
|---|---|---|---|---|
| Auto infers symbol lineage | ✓ | ✓ | ✗ | ✓ |
| Composes with NBSAFETY | ✗ | ✗ | ✓ | N/A |
| Auto resolves staleness | ✓ | ✓ | ✓* | optionally |
| ↬always does so correctly | ✓ | ✗ | ✓* | N/A‡ |
| Preserves Jupyter semantics | ✗ | ✗ | ✓† | ✓ |
| No crashing on valid Python | ✗ | ✗ | ✓ | ✓ |
| No other performance penalty | ✗ | ✗ | ✓ | ✗ (minor) |

**Table 4:** Summary of key distinguishing properties of notebook systems that help prevent stale executions.
∗ Only for manually specified dependencies.
† Except for manually specified dependencies.
‡ NBSAFETY can display the run-plan and allow manual corrections if needed.

***Summary.*** The additional overhead introduced by NBSAFETY is within the same order-of-magnitude as vanilla Jupyter, taking less than 2× longer to replay all 666 sessions, with typical slowdowns less than 1.5×. Without initialized analysis for refresher computation, however, total reply time increased to more than 3× the time taken by the vanilla Jupyter kernel.

Furthermore, we see from Figure 7 that refresher computation begins to dominate with the quadratic variant, while it remains relatively minor for the linear variant based on initialized analysis.

Although NBSAFETY's tracer introduces some additional overhead compared to the vanilla Jupyter kernel, we note that this overhead is relatively minor (less than 1.5×), and that a less-optimized tracing implementation would have performed far worse. For example, suppose `lst` contains one million elements, and we materialize the output of a `map` operation, e.g. `lst = list(lst.map(f))`. Using Python's tracing mechanism directly, this would produce at least one million `call` and `return` events, leading to overhead in excess of 10×. NBSAFETY is smart enough to disable tracing if the same program statement is encountered twice during a given execution, so that this statement executes just as in vanilla Jupyter, while still detecting that symbol `lst` should be given `f` as a dependency.

**Impact of Number of Cells on Analysis Latency.** To better illustrate the benefit of using initialized analysis for efficient computation of refresher cells, we measured the latency of just NBSAFETY's analysis component, and for each session, we plotted this time versus the total number of cells created in the session, in Figure 8.

***Summary.*** While quadratic refresher computation is acceptable for sessions with relatively few cells, we observe unacceptable per-cell latencies for larger notebooks with more than 50 or so cells. The linear variant that leverages initialized analysis, however, scales gracefully even for the largest notebooks in our execution logs.

The variance in Figure 8 for notebooks of the same size can be attributed to cells with different amounts of code, as well as different numbers of cell executions (since the size of the notebook is merely a lower bound for the aforementioned according to our replay strategy).

## 6.5 Comparison with Other Systems

We now give a qualitative comparison of NBSAFETY with other systems that attempt to resolve staleness issues, viewed through the lens of the data we collected in our empirical study. After surveying relevant literature and open source software repositories, we are aware of three such systems: Dataflow notebooks [28], Nodebook [47], and the Datalore kernel from JetBrains [10].

The most salient distinctions for each approach are summarized in Table 4. We now provide a summary for each system.

*Dataflow Notebooks.* While Dataflow notebooks have many desirable properties, they require the user to specify dependencies manually in order to leverage any staleness-resolving functionality. Dataflow notebooks can be used in conjunction with NBSAFETY if reactive cell execution via manually-specified dependencies is desired.

*Datalore and Nodebook.* Both the Datalore kernel and Nodebook seem to take take a hybrid analysis / memoization approach toward automatic staleness resolution: each cell serializes the variables that it assigns, and if a cell $c$ is rerun, a liveness checker determines what symbols need to be deserialized (using versions computed by cells that appear in $c$ spatially) and used as "inputs" to $c$, possibly rerunning cells prior to $c$ if they were edited or if they depend on an edited cell. For example, in Figure 2, the second cell would be automatically rerun if the user attempts to rerun the third cell after rerunning the first.

These approaches allow notebooks to emulate script-like top-to-bottom behavior, but serialization can come at significant cost for objects like large dataframes, and furthermore, not all objects are serializable, thereby rendering these approaches viable only on a much smaller set of programs, as we will see. Finally, because liveness gives a conservative overestimation of symbols used, these approaches may perform more work than necessary to rerun prior edited cells, or to deserialize possibly-needed symbols.

**Ability to run valid Python.** Perhaps the most serious shortcoming of memoization-based approaches stems from their failure to execute valid Python code. Consider the following example:

```
y = (i + 2 for i in range(10))                    [1]
```

```
print(list(y))                                    [3]
```

If the user edits the first cell and then attempts to run the second cell twice using either Nodebook or the Datalore kernel, they will observe an error when these approaches try (and fail) to load the non-serializable object `y` from storage.

**Ability to conduct multiverse analyses.** To facilitate the below discussion, we define the "rerun all cells" approach adopted by Nodebook, Datalore, and Dataflow notebooks (for manual dependencies) to be a "forcible cascade" approach, the selective rerun approach adopted by NBSAFETY to be a "supervised permissive cascade" approach, and that of Jupyter to be a "manual cascade" approach.

In exploratory programming and data analysis, users do not usually have a clear indication of which approach might work well upfront [24]. So, they typically try various alternative approaches to achieve their end-goal, while also recording snippets of what they had tried previously, for reuse, and for returning to old alternatives [23]. The forcible cascade approach in this case has the unintended effect of having all of their downstream alternatives being executed, when the user wanted to execute precisely one.

These sorts of multiverse analyeses are hindered by the forcible cascade approaches. In fact, we found several instances in our execution logs wherein users explicitly saved off variables to be returned to later, and where forcible cascades would have overwritten these variables' saved values. We now give a typical example, depicted below:

```
df = pd.read_csv('universe1.csv')                 [1]
```

```
df = df.dropna()[['col1', 'col2', 'col3', ...]]   [2]
```

```
df = df.rename({...})                             [3]
```

```
df = df.merge(                                    [4]
        df.grouby(...).sum().reset_index(),
        on='col', how='left')
```

```
# more cells ...                                  [...]
```

```
df_saved = df.copy()                              [n]
```

After reading the file `universe1.csv` into a dataframe and performing some transformations, the user would then save a copy of the transformed dataframe. The user would then repeat the same transformations by changing cell 1 to read in `universe2.csv` and manually running the cells below, but stopping before creating the copy.

The user then would perform some comparison between the transformed `universe1.csv` data and the transformed `universe2.csv` data. Note that a forcible cascade would have overwritten the variable `df_saved`, preventing this comparison.

## 6.6 Staleness Case Study

We now discuss a particularly egregious example of unsafe behavior in one of the 666 replayed sessions that would have been caught by NBSAFETY. In this session, the user was attempting to visualize a Wiener process defined by the following function:

```
[1]
def wiener(tmax, n):
    # Return one realization of a Wiener process
    # with n steps and a max time of tmax.
    times = np.linspace(0, tmax, n)
    difference = np.diff(times)
    process = np.random.normal(0, difference**.5)
    process = np.cumsum(process)
    return times, process
```

The user then initially called this function and saved the output in variables `t` and `w`:

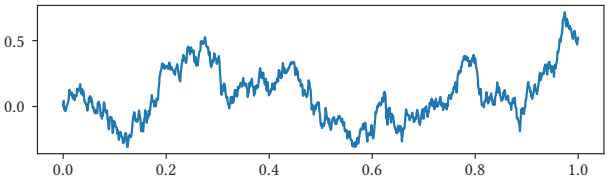
```
[2]
t, w = wiener(1.0, 1000)
```

After inspecting a few values in the array `w`, the user then decided to rename it from lowercase `w` to uppercase `W`:

```
[3]
t, W = wiener(1.0, 1000)
```

Next, the user used the popular visualization library Altair [43] to plot the output of the `wiener` function, using the following code (and producing output similar to the figure below the cell):

```
[4]
data = pd.DataFrame({'time': t, 'W': w})
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
```




However, note that in cell 4, the dataframe created by the user, `data`, refers to the old lowercase `w`, and not the new uppercase `W` most recently created. Thus, when the user reran cells 3 and 4 in succession, the exact same figure as that from the original cell 4 was generated.

Confused, the user then reran cells 1, 3, and 4 in succession, each time generating a plot identical to that from the original cell 4. This process repeated itself around 20 times, before the user finally noticed the problem and changed cell 4 to the following:

```
[4]
data = pd.DataFrame({'time': t, 'W': W})
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
```

This at last generated a different plot from the output of the original cell 4, but the entire process resulted in a large amount of wasted effort that would have been saved had the user noticed the error earlier.

**How NBSAFETY Helps.** To see how NBSAFETY would have helped, let us examine the highlights that NBSAFETY would have presented after the user reran 3 and 4, got confused, and then reran cell 1. The state of the notebook would have then appeared similar to the following:

```
[7]
def wiener(tmax, n):
    # Return one realization of a Wiener process
    # with n steps and a max time of tmax.
    times = np.linspace(0, tmax, n)
    difference = np.diff(times)
    process = np.random.normal(0, difference**.5)
    process = np.cumsum(process)
    return (times, np.insert(process, 0, 0))
```

```
[5]
t, W = wiener(1.0, 1000)
```

```
[6]
data = pd.DataFrame({'time': t, 'W': w})
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
```

That is, cell 3 would be given a **fresh** cleanup suggestion highlight, because the `wiener` symbol was recently updated when the user reran the first cell (now labeled as 7). Likewise, cell 6 is given a **stale** highlight because lowercase `w` depends on the old version of `wiener`.

Next, when the user reruns the cell labeled as 5 in the above notebook, they would expect the unsafe highlight over cell 6 to be replaced with a fresh highlight, because they refreshed the symbol `W`. However, this does not occur, and the unsafe highlight remains. The user could have then query NBSAFETY's API to determine why, and would have been presented with the following:

```
[7]
t, W = wiener(1.0, 1000)
```

```
[6]
data = pd.DataFrame({'time': t, 'W': w})
alt.Chart(data).mark_line().encode(
    x = 'time',
    y = 'W:Q'
)
# WARNING: `w` (latest update in cell 2) may depend
#          on old version of symbol(s) [`wiener`].
```

At this point, the user likely would have noticed that cell 6 does not refer to symbol `W`, which was updated when the user ran cell 7, but on symbol `w`, which lingers in notebook state from when the user originally ran cell 2.

# 7. RELATED WORK

Our work has connections to notebook systems, fine-grained data versioning and provenance, and data-centric applications of program analysis. Our notion of staleness and cell execution orders is reminiscent of the notion of serializability.We survey each area below.

**Notebook Systems.** Error-prone interactions with global notebook state are well-documented in industry and academic communities [8, 17, 20, 25, 28, 29, 34, 35, 39, 47]. The idea of treating a notebook as a dataflow computation graph with interdependent cells has been studied previously [6, 28, 47]; however, NBSAFETY is the first such system to our knowledge that preserves existing any-order execution semantics. We already surveyed Dataflow notebooks [28], Nodebook [47], and Datalore's kernel in Sections 1 and 6.5. NBGATHER [20] takes a purely static and non-dataflow approach to automatically organize notebooks using program slicing techniques, and thereby reducing non-reproducibility and errors due to messy notebooks. However, the functionality provided by NBGATHER is orthogonal to NBSAFETY and could be used in conjunction; for example, it does not help prevent state-related errors made before NBGATHER-induced reorganization.

**Versioning and Provenance.** The work on data versioning and provenance has enjoyed a long history in the database community. Provenance capture can be either *coarse-grained*, typically employed by

scientific workflow systems, e.g. [3, 5, 11, 12, 7], or *fine-grained* provenance as in database systems [16, 9, 21], typically at the level of individual rows. Within systems targeted toward individual scientists, Burrito [18] tracks file and script-level coarse-grained provenance, whereas our work falls under the fine-grained umbrella. Recent work has examined challenges related to version compaction [4, 22], and fine-grained lineage for scalable interactive visualization [36]; our focus is on enabling safer notebook interactions. Vizier [6] attempts to combine cell versioning and data provenance into a cohesive notebook system with an intuitive interface, while warning users of *caveats* (i.e., possibly brittle assumptions that the analyst made about the data). Like Vizie, we leverage lineage to propagate information about potential errors. However, data dependencies still need to be specified using their dataset API, while NBSAFETY infers them automatically using its tracer. Furthermore, in our case, the semantics of the error stem directly from the ability to execute cells out-of-order, while in Vizier, they stem from their so-called caveats. That said, NBSAFETY could, in principle, also propagate caveats; incorporating an API for specifying such caveats is an interesting avenue for future work.

**Data-centric Program Checking.** The database community has traditionally leveraged program analysis to *optimize database-backed applications* [14, 19, 37, 45], while we focus on catching bugs in an interactive notebook environment. One exception is SQLCheck [13], which employs a data-aware static analyzer to detect and fix so-called antipatterns that occur during schema and query design. Our goal with NBSAFETY is similar in spirit, though we focus on detecting and rectifying potential errors that occur over the course of interactive notebook sessions. Within the notebook space, Vizier [6] also uses static analysis to determine whether particular queries are affected by brittle assumptions / caveats. This use case is orthogonal to our goal, which is to preserve traditional notebook semantics while reducing error-proneness of such interactions; we could incorporate caveat-checking into NBSAFETY's static analysis in the future (by, e.g., detecting liveness of symbols with attached caveats in cells).

**Parallels with Transaction Processing.** There are some parallels between the notion of transactional serializability and safety. For example, if we view a cell as a transaction, conflicts between two transactions would correspond to dependencies between cells (either in a R/W, W/R or W/W fashion). Moreover, our goal in identifying stale and refresher cells is akin to *conservatively* identifying whether the execution order corresponds to a safe / desirable schedule in terms of the read/writes (e.g., whether the schedule is view equivalent to a "run from top-to-bottom schedule"). Such conservative mechanisms of identifying schedules that adhere to various consistency have been proposed in prior work [38, 46]. However, the similarities largely end there:

1. If we view a cell as a transactional boundary, reads and writes within one cell cannot be interleaved with reads and writes within another cell. Thus, the notion of serializability is itself too weak in that it allows for interleavings between transactions.
2. Say we abandon the notion of serializability, but instead consider the notion of view equivalence of two different cell execution schedules. Here, we note that there are often multiple ways to "refresh" a stale cell, typical in multiverse analyses—corresponding to different execution paths in a DAG of cell dependencies (or lineage). Any one of these would be permissible from our viewpoint. On the other hand, view equivalence is a strict linear definition, unlike our DAG-based permissive definition.
3. Finally, even in the case that there is a single path in our DAG, view equivalence ends up being overly conservative, dismissing certain valid cell execution schedules as non-equivalent, when they are indeed equivalent from an end result standpoint.

To illustrate this last point, consider the following three cells:

```
x = 0                                    [1]
```

```
y = 5                                    [2]
print(x)
```

```
print(y)                                 [3]
```

In the above example, $c_1$ writes $x$, $c_2$ reads $x$ and writes $y$, and cell $c_3$ reads $y$. Suppose the user changes the assignment in $c_1$ to `x = 42`. If enforcing view serializability, we would highlight cell $c_3$ as unsafe to execute, because $c_2$ would need to execute before it under view serializability: $c_2$ reads $c_1$'s write of $x$, and $c_3$ in turn reads $c_2$'s write of $y$. However, it is easy to see in the above example that the order of re-execution between $c_2$ and $c_3$ does not matter. If, on the other hand, $c_2$ had set `y = x + 1`, then it is clear that $c_2$ should be rerun before $c_3$ can be safely rerun. It is for this reason that we adopted a lineage-centric framework, wherein cells are used primarily to associate timestamps with symbols and blocks of code via their execution counters.

## 8. CONCLUSION

We presented NBSAFETY, a kernel and frontend for Jupyter that attempts to detect and correct potentially unsafe interactions in notebooks, all while preserving the flexibility of familiar any-order notebook semantics. We described the implementation of NBSAFETY's tracer, checker, and frontend, and how they integrate into existing notebook workflows to *efficiently* reduce error-proneness in notebooks. We showed how cells that NBSAFETY would have warned as unsafe were actively avoided, and cells that would have been suggested for re-execution were prioritized by real users on a corpus of 666 real notebook sessions. While we focused on unsafe interactions due to staleness in this paper, extending our approach to other types of unsafe interactions is a promising direction for future research.

## REFERENCES


[1] *2to3: Automated Python 2 to 3 code translation*. https://docs.python.org/3/library/2to3.html. Date accessed: 2020-07-29. 2020.

[2] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. "Compilers, principles, techniques". In: *Addison wesley* 7.8 (1986), p. 9.

[3] Manish Kumar Anand et al. "Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs". In: *Scientific and Statistical Database Management*. Springer. 2009, pp. 237–254.

[4] Souvik Bhattacherjee et al. "Principles of dataset versioning: Exploring the recreation/storage tradeoff". In: *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*. Vol. 8. 12. NIH Public Access. 2015, p. 1346.

[5] Shawn Bowers. "Scientific workflow, provenance, and data modeling challenges and approaches". In: *Journal on Data Semantics* 1.1 (2012), pp. 19–30.

[6] Mike Brachmann et al. "Your notebook is not crumby enough, REPLace it." In: *CIDR*. 2020.

[7] Peter Buneman et al. "Archiving scientific data". In: *ACM Transactions on Database Systems (TODS)* 29.1 (2004), pp. 2–42.

[8] Souti Chattopadhyay et al. "What's Wrong with Computational Notebooks? Pain Points, Needs, and Design Opportunities". In: *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 2020, pp. 1–12.

[9] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. *Provenance in databases: Why, how, and where*. Now Publishers Inc, 2009.

[10] *Datalore*. https://datalore.jetbrains.com/. 2018 (accessed December 1, 2020).

[11] Susan B Davidson, Sarah Cohen Boulakia, et al. "Provenance in Scientific Workflow Systems." In: *IEEE Data Eng. Bull.* 30.4 (2007), pp. 44–50.

[12] Susan B Davidson and Juliana Freire. "Provenance and scientific workflows: challenges and opportunities". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. ACM. 2008, pp. 1345–1350.

[13] Prashanth Dintyala, Arpit Narechania, and Joy Arulraj. "SQLCheck: Automated Detection and Diagnosis of SQL Anti-Patterns". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 2331–2345.

[14] K Venkatesh Emani et al. "Dbridge: Translating imperative code to sql". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1663–1666.

[15] *GitHub Search API*. https://developer.github.com/v3/search/. Date accessed: 2020-07-29. 2020.

[16] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. "Provenance Semirings". In: *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '07. Beijing, China: ACM, 2007, pp. 31–40. ISBN: 978-1-59593-685-1. DOI: 10.1145/1265530.1265535. URL: http://doi.acm.org/10.1145/1265530.1265535.

[17] Joel Grus. *I Don't Like Notebooks (JupyterCon 2018 Talk)*. https://t.ly/Wt3S. 2018 (accessed June 26, 2020).

[18] Philip J Guo and Margo I Seltzer. "Burrito: Wrapping your lab notebook in computational infrastructure". In: (2012).

[19] Surabhi Gupta, Sanket Purandare, and Karthik Ramachandra. "Aggify: Lifting the Curse of Cursor Loops using Custom Aggregates". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 559–573.

[20] Andrew Head et al. "Managing messes in computational notebooks". In: *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 2019, pp. 1–12.

[21] Melanie Herschel et al. "A survey on provenance: What for? What form? What from?" In: *The VLDB Journal* 26.6 (2017), pp. 881–906.

[22] Silu Huang et al. "OrpheusDB: bolt-on versioning for relational databases (extended version)". In: *The VLDB Journal* 29.1 (2020), pp. 509–538.

[23] Mary Beth Kery, Amber Horvath, and Brad A Myers. "Variolite: Supporting Exploratory Programming by Data Scientists." In: *CHI*. Vol. 10. 2017, pp. 3025453–3025626.

[24] Mary Beth Kery and Brad A Myers. "Interactions for untangling messy history in a computational notebook". In: *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2018, pp. 147–155.

[25] Mary Beth Kery et al. "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*. 2018, pp. 1–11.

[26] Thomas Kluyver et al. "Jupyter Notebooks-a publishing format for reproducible computational workflows." In: *ELPUB*. 2016, pp. 87–90.

[27] Donald Ervin Knuth. "Literate programming". In: *The Computer Journal* 27.2 (1984), pp. 97–111.

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

[29] Sam Lau et al. "The Design Space of Computational Notebooks: An Analysis of 60 Systems in Academia and Industry". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. VL/HCC '20. Aug. 2020.

[30] Stephen Macke. *NBSafety Experiments*. https://github.com/nbsafety-project/nbsafety-experiments/. 2020 (accessed July 29, 2020).

[31] Stephen Macke and Hongpu Gong. *NBSafety*. https://github.com/nbsafety-project/nbsafety/. 2020 (accessed July 29, 2020).

[32] Anders Møller and Michael I Schwartzbach. "Static program analysis". In: *Notes. Feb* (2012).

[33] Jim Ormond. *ACM Recognizes Innovators Who Have Shaped the Digital Revolution*. https://awards.acm.org/binaries/content/assets/press-releases/2018/may/technical-awards-2017.pdf. 2018 (accessed June 26, 2020).

[34] Jeffrey M Perkel. "Why Jupyter is data scientists' computational notebook of choice". In: *Nature* 563.7732 (2018), pp. 145–147.

[35] João Felipe Pimentel et al. "A large-scale study about quality and reproducibility of jupyter notebooks". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 507–517.

[36] Fotis Psallidas and Eugene Wu. "Smoke: Fine-grained lineage at interactive speed". In: *arXiv preprint arXiv:1801.07237* (2018).

[37] Karthik Ramachandra et al. "Froid: Optimization of imperative programs in a relational database". In: *Proceedings of the VLDB Endowment* 11.4 (2017), pp. 432–444.

[38] Sudip Roy et al. "The homeostasis protocol: Avoiding transaction coordination through program analysis". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 2015, pp. 1311–1326.

[39] Adam Rule, Aurélien Tabard, and James D Hollan. "Exploration and explanation in computational notebooks". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 2018, pp. 1–12.

[40] Helen Shen. "Interactive notebooks: Sharing the code". In: *Nature* 515.7525 (2014), pp. 151–152.

[41] Sara Steegen et al. "Increasing transparency through a multiverse analysis". In: *Perspectives on Psychological Science* 11.5 (2016), pp. 702–712.

[42] *sys: System-specific parameters and functions*. https://docs.python.org/2/library/sys.html#sys.settrace. Date accessed: 2020-07-29. 2020.

[43] Jacob VanderPlas et al. "Altair: Interactive statistical visualizations for python". In: *Journal of open source software* 3.32 (2018), p. 1057.

[44] Cong Yan and Yeye He. "Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2020, pp. 1539–1554.

[45] Cong Yan et al. "Understanding database performance inefficiencies in real-world web applications". In: *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*. 2017, pp. 1299–1308.

[46] Yang Zhang et al. "Transaction chains: achieving serializability with low latency in geo-distributed storage systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 276–291.

[47] Kevin Zielnicki. *Nodebook*. https://multithreaded.stitchfix.com/blog/2017/07/26/nodebook/. 2017 (accessed July 5, 2020).