# Fine-Grained Lineage for Safer Notebook Interactions

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]University of California, Berkeley
{smacke,ruiduoray,dorislee,andrewhead,dorx,adityagp}@berkeley.edu

## ABSTRACT

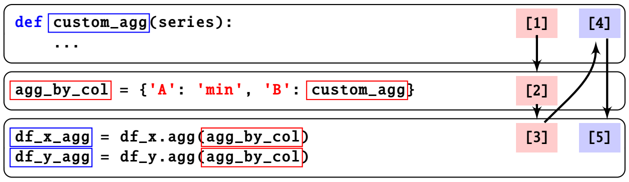
Computational notebooks have emerged as the platform of choice for data science and analytical workflows, enabling rapid iteration and exploration. By keeping intermediate program state in memory and segmenting units of execution into so-called "cells", notebooks allow users to enjoy particularly tight feedback. However, as cells are added, removed, reordered, and rerun, this hidden intermediate state accumulates, 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.

**PVLDB Reference Format:**
Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. Fine-Grained Lineage for Safer Notebook Interactions. PVLDB, 14(6): 1093-1101, 2021.
doi:10.14778/3447689.3447712

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/nbsafety-project/.

## 1 INTRODUCTION

Computational notebooks such as Jupyter [23] 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 their use of the *cell* as the atomic unit of execution, allowing users to edit and re-execute previous cells. This cell-based iterative execution modality is a particularly good fit for the exploratory, ad-hoc nature of modern data science.



```
def custom_agg(series):
    ...

agg_by_col = {'A': 'min', 'B': custom_agg}

df_x_agg = df_x.agg(agg_by_col)
df_y_agg = df_y.agg(agg_by_col)
```

**Figure 1: Example sequence of notebook interactions leading to a stale symbol usage. Symbols with timestamps $\leq 3$ (resp. $> 3$) are shown with blue (resp. red) borders.**

As a result, the IPython Notebook project [35], and its successor, Project Jupyter [23], have both grown rapidly in popularity. With more than 4.7 million notebooks on GitHub as of March 2019 [37], Jupyter has been called "data scientists' computational notebook of choice" [30]. We focus on Jupyter here due to its popularity, but 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.

**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.

This example underscores the inherent difficulty in manually managing notebook state, inspiring colorful criticisms such as a talk

Proceedings of the VLDB Endowment, Vol. 14, No. 6 ISSN 2150-8097.
doi:10.14778/3447689.3447712

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.

**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, 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 [24]), 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.

Despite previous attempts to address these challenges and to facilitate safer interactions with global notebook state [1, 24, 38], to our knowledge, NBSAFETY is the first to do so while preserving existing notebook semantics. For example, Dataflow notebooks [24] require users to explicitly annotate cells with their dependencies, and force the re-execution of cells whose dependencies have changed. Nodebook [38] and the Datalore kernel [1] 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 [25], 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. After a single installation command [27], 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:

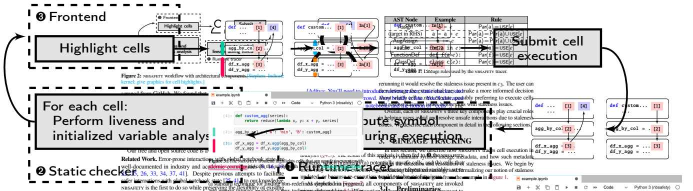


**Figure 2: NBSAFETY workflow with architectural components.**

*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.

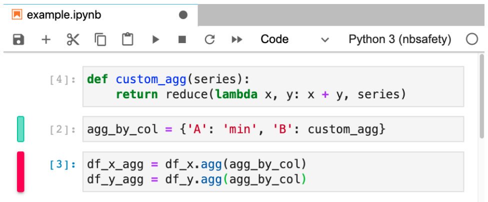
*2. Efficient resolution of staleness issues.* Beyond simply detecting staleness issues, we also show how to detect cells whose re-execution would resolve such staleness issues — but doing so *efficiently* required us to leverage a lesser-known analysis technique called *initialized variable analysis* (§4) 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 exploratory data analysis, and less on avoiding and fixing state-related notebook bugs.

Our free and open source code is available publicly on GitHub [27].

## 2 ARCHITECTURE OVERVIEW

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



```
[4]: def custom_agg(series):
         return reduce(lambda x, y: x + y, series)

[2]: agg_by_col = {'A': 'min', 'B': custom_agg}

[3]: df_x_agg = df_x.agg(agg_by_col)
     df_y_agg = df_y.agg(agg_by_col)
```

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

**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* [4] and *initialized variable analysis* [29] 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 [3],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.

| AST Node | Example | Rule |
|---|---|---|
| Assign | `a = ` $e$ | $\mathsf{Par}(\mathsf{a}) = \mathsf{USE}[e]$ |
| (target in RHS) | `a = a + ` $e$ | $\mathsf{Par}(\mathsf{a}) = \mathsf{Par}(\mathsf{a}) \cup \mathsf{USE}[e]$ |
| AugAssign | `a += ` $e$ | $\mathsf{Par}(\mathsf{a}) = \mathsf{Par}(\mathsf{a}) \cup \mathsf{USE}[e]$ |
| For | `for a in ` $e$`:` | $\mathsf{Par}(\mathsf{a}) = \mathsf{USE}[e]$ |
| FunctionDef | `def f(a=`$e$`):` | $\mathsf{Par}(\mathsf{f}) = \mathsf{USE}[e]$ |
| ClassDef | `class c(`$e$`):` | $\mathsf{Par}(\mathsf{c}) = \mathsf{USE}[e]$ |

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

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 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.

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

**Definition 3** [Dependencies]. *The* dependencies *of symbol* $\mathsf{s}$ *are those symbols that contributed to* $\mathsf{s}$*'s computation 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 denote the dependencies of $\mathsf{s}$ with $\mathsf{Par}(\mathsf{s})$.

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* $\mathsf{s}$ *is called* stale *if there exists some* $\mathsf{s}' \in \mathsf{Par}(\mathsf{s})$ *such that* $\mathsf{ts}(\mathsf{s}') > \mathsf{ts}(\mathsf{s})$*, or* $\mathsf{s}'$ *is itself stale; that is,* $\mathsf{s}$ *has a parent that is either itself stale or more up-to-date than* $\mathsf{s}$.

In Figure 1, symbol `agg_by_col` is stale, because $\mathsf{ts}(\mathsf{agg\_by\_col}) = 2$, but $\mathsf{ts}(\mathsf{custom\_agg}) = 4$. 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.

We now draw on these definitions as we describe how NBSAFETY maintains lineage metadata 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 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 Par(`gen`) = {`f`,`foo`,`bar`} and will also update ts(`gen`).

**Fine-Grained Lineage for Attributes and Subscripts.** NBSAFETY is able to track lineage at a finer granularity than simply top-level symbols. For example, NBSAFETY tracks parents and children of subscript symbols like `x[0]` and attribute symbols like `x.a` (as well as combinations thereof) as first-class citizens, in addition to those of top-level symbols such as `x`. Please see the technical report [28] for more details.

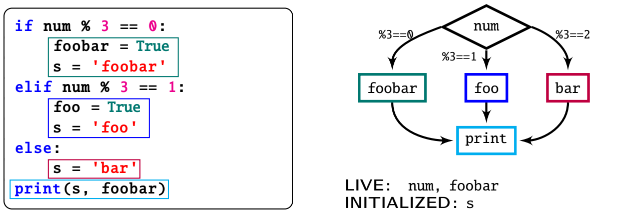
**Staleness Propagation.** We already saw that the tracer annotates each symbol’s shadow reference with timestamp and lineage metadata. Additionally, it tracks whether each symbol is stale, as this cannot be inferred solely from timestamp and lineage metadata. To see why, recall the definition of staleness: a symbol `s` is stale if it has a more up-to-date parent (i.e., an `s′` ∈ Par(`s`) with ts(`s′`) > ts(`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 `c` ∈ Chd(`s`) in order to propagate the “staleness” to all descendants.

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

```
x = 0                                                  [1]
for i in random.sample(range(10**7), 10**5):
    x += lst[i]
```

In order to maintain lineage metadata for symbol `x` to 100% correctness, we would need to somehow indicate that Par(`x`) contains `lst[i]` for all $10^5$ random indices `i`. 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. We therefore opted for a lossy approximation that we describe in our extended technical report [28].

**Handling Calls to External Libraries.** When NBSAFETY’s tracer traps due to a function call, it inspects the location of the called function. If the called function was not defined in the user’s notebook, but in some imported file, NBSAFETY disables tracing until control returns to the notebook proper, since external files typically do not have access to state defined in the notebook. If an object in notebook state is passed explicitly as, e.g., a function parameter, NBSAFETY assumes the library does not mutate it; we leave improvements to future work. By disabling tracing when control is outside the notebook, we ensure that additional tracing overhead is bounded by the size of the user’s notebook.



```
if num % 3 == 0:
    foobar = True
    s = 'foobar'
elif num % 3 == 1:
    foo = True
    s = 'foo'
else:
    s = 'bar'
print(s, foobar)
```

LIVE: num, foobar
INITIALIZED: s

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

Finally, our technical report [28] contains additional details surrounding the tracer, such as how we garbage collect shadow metadata, how we handle mutations when variables alias each other, and how we handle namespaced symbols such as attributes .

## 4 LIVENESS AND INITIALIZED ANALYSES

In this section, we describe the program analysis component of NBSAFETY’s backend. The *checker* performs liveness analysis [4], and a lesser-known program analysis technique called initialized variable analysis, or definite assignment analysis [29]. 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* [4] 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 [29], a technique traditionally used by IDEs and linters to detect potentially uninitialized variables, can be used to efficiently determine which cells to to resolve staleness.

**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.

**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 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 $\textsc{dead}(c)$, 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 $\textsc{stale}(c) \subseteq \textsc{live}(c)$, 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* $\mathtt{s} \in \textsc{live}(c)$ *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)* $\mathtt{ts}(\mathtt{s}) > \mathtt{ts}(c)$.

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, $\exists \mathtt{s} \in \textsc{live}(c)$ 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*

$$\textsc{stale}(c_s) - \textsc{stale}(c_r \oplus c_s) \neq \varnothing$$

*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 compute dead symbols.

**Initialized Variable Analysis.** Initialized variable analysis [29] can be thought of as the "inverse" of liveness analysis. While liveness analysis is a "backwards-may" technique for computing symbols whose non-overwritten values "may" be used in a cell, initialized analysis is a "forwards-must" technique that computes symbols that will be "definitely assigned" by the time control reaches the end of a cell. Please see the technical report [28] for a detailed discussion of initialized analysis; we leverage it within the context of nbsafety to determine whether a non-stale cell will overwrite any stale symbols, which turns out to be an efficient mechanism for computing refresher cells (§5.2).

### 4.3 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 with the output of the 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;

- $\mathcal{H}_f$, 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. NBSAFETY 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).

**Computation.** Computing $\mathcal{H}_s$ and $\mathcal{H}_f$ is straightforward: for each cell $c$, we simply run a liveness checker to determine $\textsc{live}(c)$, and then perform a metadata lookup for each symbol $\mathsf{s} \in \textsc{live}(c)$ to determine whether $\mathsf{s}$ is fresh w.r.t. $c$ or stale. Refresher cell computation deserves a more thorough treatment that we consider next.

## 5.2 Computing Refresher Cells Efficiently

Before we discuss how NBSAFETY uses an 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 $\textsc{stale}(c_r \oplus c_s)$ is smaller than $\textsc{stale}(c_s)$. 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.

By leveraging an initialized variable checker, it turns out that we can check whether $\textsc{stale}(c_s)$ and $\textsc{dead}(c_r)$ have any overlap instead of performing liveness analysis over $c_r \oplus c_s$ and checking whether $\textsc{stale}(c_r \oplus c_s)$ 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:*

$$\textsc{stale}(c_s) - \textsc{stale}(c_r \oplus c_s) = \textsc{dead}(c_r) \cap \textsc{stale}(c_s)$$

Please see the technical report [28] for a proof. □

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 $\textsc{dead}(c_r)$ 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

$$\bigcup_{c_s \in \mathcal{H}_s} \bigcup_{\mathsf{s} \in \textsc{stale}(c_s)} \{c_r \in N - \mathcal{H}_s : \mathsf{s} \in \textsc{dead}(c_r)\} \tag{1}$$

Equation 1 can be computed efficiently by creating inverted index that maps dead symbols to their containing cells ($\textsc{dead}^{-1}$) in order to efficiently compute the inner set union. Furthermore, this approach 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).

## 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 are re-executed, and the order in which cells were executed. 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 [2], 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. [37] to repair sessions wherever possible. Please see our technical report [28] for details for repair and filtering (below).

**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 undesirable sessions, after which 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 scripts are available on GitHub [26].

## 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.

**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 $\mathcal{P}(\mathcal{H}) = \mathbb{I}\{c \in \mathcal{H}\} \cdot |N|/|\mathcal{H}|$.*

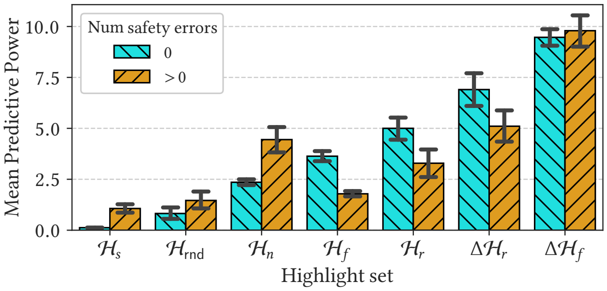


**Figure 5: $\mathsf{AVG}(\mathcal{P}(\mathcal{H}_*))$ for sessions with/without safety issues.**

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 $\mathsf{AVG}(\mathcal{P}(\mathcal{H}_*))$ 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.

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

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

## 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 staleness issues at some point*, underscoring the 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. 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 5.

> ***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 5 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.

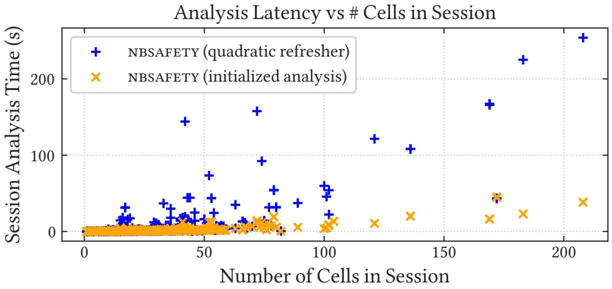


**Figure 6: Measuring the impact of cell count on analysis latency for NBSAFETY with and without efficient refresher computation.**

| **Approach** | Jupyter | NBSAFETY | NBSAFETY (quadratic refresher) |
|---|---|---|---|
| Analysis Time (s) | 0 | 990 | 5070 |
| Execution Time (s) | 3150 | 4850 | 4850 |
| 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 $>5$ seconds to execute in vanilla Jupyter.**

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.

## 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 by comparing it with the naïve quadratic baseline (both discussed in Section 5).

**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 the same table. 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.

> ***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 Table 3 that refresher computation begins to dominate with the quadratic variant, while it remains relatively minor for the linear variant based on initialized analysis.

**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 6.

> ***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 6 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 a lower bound for the aforementioned according to our replay strategy).

## 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 elaborate on this connection in our technical report [28].

**Notebook Systems.** Error-prone interactions with global notebook state are well-documented [10, 17, 19, 22, 24, 25, 30, 31, 34, 38]. The idea of treating a notebook as a dataflow computation graph has been studied previously [8, 24, 38]; however, NBSAFETY is the first such system to our knowledge that preserves existing any-order execution semantics. We already surveyed Dataflow notebooks [24], Nodebook [38], and Datalore's kernel in Section 1. NBGATHER [19] takes a purely static approach to automatically organize notebooks thereby reducing non-reproducibility. However, NBGATHER does not help prevent state-related errors made before reorganization.

**Versioning and Provenance.** Provenance capture can be either *coarse-grained*, typically employed by scientific workflow systems, e.g. [5, 7, 9, 12, 13], or *fine-grained* provenance as in database systems [11, 16, 20], typically at the level of individual rows. Recent work has examined challenges related to version compaction [6, 21], and fine-grained lineage for scalable interactive visualization [32]. Vizier [8] 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.

**Data-centric Program Checking.** The database community has traditionally leveraged program analysis to *optimize database-backed applications* [15, 18, 33, 36], while we focus on catching bugs in an interactive notebook environment. One exception is SQLCheck [14], which employs a data-aware static analyzer to detect and fix so-called antipatterns that occur during schema and query design.

## 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.

## ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We additionally thank Sarah Chasins for bringing to our attention the connection between refresher cells and initialized variable analysis. We acknowledge support from grants IIS-1940759 and IIS-1940757 awarded by the National Science Foundation, and funds from the Alfred P. Sloan Foundation, Facebook, Adobe, Toyota Research Institute, Google, the Siebel Energy Institute, and State Farm. The content is solely the responsibility of the authors and does not necessarily represent the official views of the funding agencies and organizations.

# BIBLIOGRAPHY

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

[2] 2020. GitHub Search API. https://developer.github.com/v3/search/. Date accessed: 2020-07-29.

[3] 2020. sys: System-specific parameters and functions. https://docs.python.org/2/library/sys.html#sys.settrace. Date accessed: 2020-07-29.

[4] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. Compilers, principles, techniques. *Addison wesley* 7, 8 (1986), 9.

[5] Manish Kumar Anand, Shawn Bowers, Timothy Mcphillips, and Bertram Ludäscher. 2009. Exploring scientific workflow provenance using hybrid queries over nested data and lineage graphs. In *Scientific and Statistical Database Management.* Springer, 237–254.

[6] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of dataset versioning: Exploring the recreation/storage tradeoff. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 8. NIH Public Access, 1346.

[7] Shawn Bowers. 2012. Scientific workflow, provenance, and data modeling challenges and approaches. *Journal on Data Semantics* 1, 1 (2012), 19–30.

[8] Mike Brachmann, William Spoth, Oliver Kennedy, Boris Glavic, Heiko Mueller, Sonia Castelo, Carlos Bautista, and Juliana Freire. 2020. Your notebook is not crumby enough, REPLace it.. In *CIDR*.

[9] Peter Buneman, Sanjeev Khanna, Keishi Tajima, and Wang-Chiew Tan. 2004. Archiving scientific data. *ACM Transactions on Database Systems (TODS)* 29, 1 (2004), 2–42.

[10] Souti Chattopadhyay et al. 2020. 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.* 1–12.

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

[12] Susan B Davidson, Sarah Cohen Boulakia, et al. 2007. Provenance in Scientific Workflow Systems. *IEEE Data Eng. Bull.* 30, 4 (2007), 44–50.

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

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

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

[16] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance Semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Beijing, China) *(PODS '07).* ACM, New York, NY, USA, 31–40. https://doi.org/10.1145/1265530.1265535

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

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

[19] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems.* 1–12.

[20] Melanie Herschel et al. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26, 6 (2017), 881–906.

[21] Silu Huang, Liqi Xu, Jialin Liu, Aaron J Elmore, and Aditya Parameswaran. 2020. OrpheusDB: bolt-on versioning for relational databases (extended version). *The VLDB Journal* 29, 1 (2020), 509–538.

[22] Mary Beth Kery, Marissa Radensky, Mahima Arya, Bonnie E John, and Brad A Myers. 2018. The story in the notebook: Exploratory data science using a literate programming tool. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems.* 1–11.

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

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

[25] Sam Lau, Ian Drosos, Julia M. Markel, and Philip J. Guo. 2020. 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).*

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

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

[28] Stephen Macke, Hongpu Gong, Doris Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2020. Fine-Grained Lineage for Safer Notebook Interactions. *Technical Report* (2020). https://drive.google.com/file/d/1U8NTLRMEiwevGPrXEHvPGWL_uCBWUdyZ/view Available at: https://smacke.net/papers/nbsafety.pdf.

[29] Anders Møller and Michael I Schwartzbach. 2012. Static program analysis. *Notes. Feb* (2012).

[30] Jeffrey M Perkel. 2018. Why Jupyter is data scientists' computational notebook of choice. *Nature* 563, 7732 (2018), 145–147.

[31] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2019. A large-scale study about quality and reproducibility of jupyter notebooks. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, 507–517.

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

[33] Karthik Ramachandra, Kwanghyun Park, K Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: Optimization of imperative programs in a relational database. *Proceedings of the VLDB Endowment* 11, 4 (2017), 432–444.

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

[35] Helen Shen. 2014. Interactive notebooks: Sharing the code. *Nature* 515, 7525 (2014), 151–152.

[36] Cong Yan, Alvin Cheung, Junwen Yang, and Shan Lu. 2017. Understanding database performance inefficiencies in real-world web applications. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management.* 1299–1308.

[37] Cong Yan and Yeye He. 2020. 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.* 1539–1554.

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