Bolt-on, Compact, and Rapid Program Slicing for Notebooks [Technical Report]

Shreya Shankar†1, Stephen Macke†2, Sarah Chasins1, Andrew Head3, Aditya Parameswaran1

1University of California, Berkeley {shreyashankar,schasins,adityagp}@berkeley.edu

2Unaffiliated stephen.macke@gmail.com

3University of Pennsylvania head@seas.upenn.edu

†Equal contribution (order determined by coin flip)

ABSTRACT

Computational notebooks are commonly used for iterative workflows, such as in exploratory data analysis. This process lends itself to the accumulation of old code and hidden state, making it hard for users to reason about the lineage of, e.g., plots depicting insights or trained machine learning models. One way to reason about code used to generate various notebook data artifacts is to compute a *program slice*, but traditional static approaches to slicing can be both inaccurate (failing to contain relevant code for artifacts) and conservative (containing unnecessary code for an artifacts). We present NBSLICER, a dynamic slicer optimized for the notebook setting whose instrumentation for resolving dynamic data dependencies is both *bolt-on* (and therefore portable) and *switchable* (allowing it to be selectively disabled in order to reduce instrumentation overhead). We demonstrate NBSLICER's ability to construct small and accurate *backward slices* (i.e., historical cell dependencies) and *forward slices* (i.e., cells affected by the "rerun" of an earlier cell), thereby improving reproducibility in notebooks and enabling faster reactive re-execution, respectively. Comparing NBSLICER with a static slicer on 374 real notebook sessions, we found that NBSLICER filters out far more superfluous program statements while maintaining slice correctness, giving slices that are, on average, 66% and 54% smaller for backward and forward slices, respectively.

1 INTRODUCTION

Computational notebooks, and Project Jupyter [37] in particular, have revolutionized the workflows of data scientists [48, 49]. Notebooks admit a flexible execution model that segments units of computation into so-called "cells" that can easily be back-referenced for editing, duplication, or reordering, with intermediate program state persisted to memory between subsequent cell executions. This iterative cell-based execution modality is ideal for rapid prototyping and testing of hypotheses, a cornerstone of typical data science work, and has led to their extensive usage—as of October 2020, there were nearly 10 million notebooks available on GitHub [24].

Notebooks are Messy. The popularity of notebooks has come with increased scrutiny; as such, notebooks now have a number of well-documented disadvantages, related primarily to their tendency to accumulate cruft in the form of both visible notebook code [31, 35, 53] and invisible in-memory program state [22]. Data science workflows, in particular, tend to be exploratory in nature [35]. These flaws can make execution behavior in notebooks difficult to reason about and cause misleading or incorrect findings, leading to confusion during ad-hoc exploration, prototyping, and iteration.

Organizing Notebook Iteration with Program Slicing. To address notebook shortcomings, recent work has developed approaches based on *backward program slicing* to gather code in messy notebooks [27, 33], thereby making it easier for data scientists to retrace their steps. Traditionally applied to program debugging [7, 61], *program slicing* determines a (typically smaller) subset of program statements that affect some other program statement(s). In the context of notebooks, backward program slicing captures the lineage required to reproduce the outputs of one or more cells; e.g., to "gather" code that was written in an ad-hoc fashion, potentially out-of-order across multiple notebook cells, into a clean script that reliably reproduces the data scientist's analyses. The more compact this slice is, the smaller the lineage, meaning it is more efficient to re-execute for reproducibility while preserving accuracy.

Forward program slicing also has applications in data science workflows in computational notebooks. In the context of notebooks, a *forward program slice* determines the set of cells that are affected by a given cell, and can be combined with other analysis techniques to automatically (or *reactively* [14, 40, 44, 58]) re-execute all the cells that could be affected (via data dependencies) by some other cell, ensuring that cells do not become stale. A reactivity tool for notebooks can be thought of as performing a form of materialized view maintenance — specifically, "refreshing" the notebook after an earlier cell is rerun by re-executing dependent cells. Such reactivity features help to push the burden of tracking what cells have become stale away from the user and down into the notebook kernel. This feature is particularly helpful for data science workflows, which can involve toggling many values (e.g., hyperparameters) and re-executing what might be dozens of downstream data transformation dependencies. Once again, correctness (i.e., making sure we are rerunning everything affected) while minimizing slice size (i.e.,only rerunning what is needed) is key to ensuring interactivity during exploratory data analysis.

Notebook-Centric Slicing: Challenges. Backward and forward slicing can help automate away some of the messiness inherent in notebook-resident data science workflows and thereby improve downstream reproducibility. Ideally, slices should be as small as possible in order to eliminate extraneous computation, while preserving correctness of the underlying program. However, computation of slices that are simultaneous *small* and *accurate*, and without noticeable degradation of existing notebook behavior and performance, is difficult to achieve in practice. We now outline challenges we faced while developing NBSLICER, a state-of-the-art dynamic slicer optimized specifically for the notebook setting, along with contributions that addressed each challenge.

Challenge 1: Small and accurate program slices. Backward slicing was first explored in the context of code gathering in notebooks by Head et al. [27]. However, it is not difficult to construct cases wherein the static slicing technique used in [27] will yield overly-conservative (and therefore larger than necessary) slices. One approach to reduce slice size while preserving correctness is to leverage *dynamic slicing* [39]. In contrast with static slicing, which infers dependencies between cells solely from their static contents, dynamic slicing infers such dependencies using the actual runtime state of the notebook's execution log.

We now illustrate such dynamic behavior with an example. Consider an abridged verison of a real example session from the replay dataset described in Section 4.1 wherein NBSLICER, which uses dynamic slicing, yields a more compact slice than a static slicer. In this session, a user explores a dataset and fits a least-squares model to it:

```
In [1]:  import numpy as np
         import pandas as pd

         housing = pd.read_csv('housing.txt', sep='\s+')
```

```
In [2]:  corr = housing.corr()
         np.abs(corr.sort(columns=['crime', 'residential']))
```

```
In [3]:  def LR_solve(X, y):
             """ Solves a linear regression problem."""
             if len(X.shape) == 2:
                 A = np.hstock( (np.ones((X.shape[0],1)), X) )
             else:
                 A = np.hstack(
                 (np.ones((X.shape[0],1)), np.expand_dims(X,1)) )
             rtn = np.linalg.lstsq(A, y)
             return rtn[0]
```

```
In [4]:  X = housing[:, :(- 1)]
         b = housing[:, (- 1)]
         LR_solve(X, b)
```

The user runs into an error because there was a typo of `np.hstock` in cell 3, motivating them to rerun cells 3 and 4 (as cells 5 and 6, respectively) after fixing.

```
In [5]:  def LR_solve(X, y):
             """ Solves a linear regression problem."""
             if len(X.shape) == 2:
                 A = np.hstack( (np.ones((X.shape[0],1)), X) )
             else:
                 A = np.hstack(
                 (np.ones((X.shape[0],1)), np.expand_dims(X,1)) )
             rtn = np.linalg.lstsq(A, y)
             return rtn[0]
```

```
In [6]:  X = housing[:, :(- 1)]
         b = housing[:, (- 1)]
         LR_solve(X, b)
```

When running NBSLICER to reconstruct the output of the last cell (i.e., cell 6), we see that the dynamic slice contains only cells 1, 5, and 6. However, a static slicer would include *the entire execution log in its slice*, including cells 3 and 4, which were run before the typo was fixed. We discuss why the additional cells are included in the static slicer below:

- Cell 2 calls functions from external libraries `numpy` and `pandas`. A dynamic slicer can figure out that the `np` and `pd` objects reference these external libraries and do not modify state; thus NBSLICER is able to exclude these cells.
- Cells 4 and 5 attempt to fit a least squares model to the data. However, the typo in cell 4 — `np.hstock` (instead of `np.hstack`) — still parses and is included in the static slice. Again, these cells also include calls to external libraries and are therefore included in the static slice even though they do not modify state.

Though dynamic slicing can reduce slice size while preserving correctness, prior approaches require deep modifications to the underlying interpreter or virtual machine responsible for running code in the target language [16, 60], presenting problems from both performance and portability standpoints, which we describe next.

Challenge 2: Balancing performance with portability. Dynamic slicing for smaller and more accurate slices is itself not a new idea, but it comes with the drawback of significant overhead, since it requires visibility into executing code in order to infer runtime data dependencies. For example, Python ≥ 3.8 supports tracing of individual bytecode operations via builtin system tracing utilities [5] — functionality upon which runtime data dependency tracking could be implemented — but we found that leveraging this approach leads to slowdowns of 100× or more. Such overhead is clearly infeasible for typical data science workloads, which are interactive in nature.

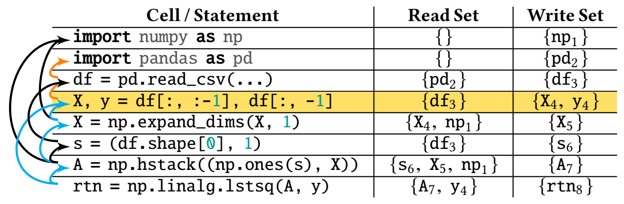
An alternative approach is to instrument the interpreter itself in a lower-level language such as C, as in prior work [16]. However, if a dynamic slicer were to require a custom-compiled Python interpreter, data scientists wishing to use said slicer would need to re-download a new interpreter every time a new version of Python is released — a process with considerably more friction than the typical workflow of grabbing a library from PyPI or Anaconda. From a maintainer standpoint, supporting multiple versions of Python is also less than ideal. Indeed, previous Python dynamic slicers [16] instrument much older versions of the Python interpreter and seem to no longer be publicly available. Overall, implementing a dynamic slicer that can be installed portably as a simple Python library, while simultaneously introducing negligible overhead and retaining interactive latencies, is entirely nontrivial.

NBSLICER: A Hybrid Static-Dynamic Slicer. To make dynamic slicing work for the notebook setting, wherein interactivity and portability are key, we developed *NBSLICER*, a novel hybrid static-dynamic slicing tool optimized for the notebook setting. The key technical contribution of NBSLICER is to instrument data scientists' code at the level of the *abstract syntax tree* (AST), rather than at the level of Python bytecode, allowing for portable and *switchable* instrumentation that can be selectively enabled and disabled in order to bound the amount of overhead induced by such instrumentation, all while retaining most of its benefits regarding compactness and accuracy of slices.

When instrumentation is enabled, nodes in a program's AST are transformed in a way that injects additional observability into the data referenced by the program, but without changing the program's behavior. The aforementioned "additional observability" takes the form of a *data dependency resolver* that decorates portions of the *uninstrumented* AST with the data dependencies that would be difficult to infer purely statically. From here, a static analyzer operates on this enriched AST and short-circuits whenever it encounters subtrees that have been marked as having dynamically-resolved data dependencies. For example, NBSLICER will attempt to dynamically resolve data dependencies involved in `Call` nodes; e.g., in our earlier example, it will resolve the dependencies involved with the statement

```
A = np.hstack( (np.ones((X.shape[0],1)), X) )
```

as edges from `np` and `X` to `A`, and it will *not* include an edge from `np` to `np` as it is aware that calls to `numpy` functions such as `hstack` and `ones` are non-mutating. If the AST were not enriched with this



Cell / Statement	Read Set	Write Set
`import numpy as np`	{}	{np_1}
`import pandas as pd`	{}	{pd_2}
`df = pd.read_csv(...)`	{pd_2}	{df_3}
`X, y = df[:, :-1], df[:, -1]`	{df_3}	{X_4, y_4}
`X = np.expand_dims(X, 1)`	{X_4, np_1}	{X_5}
`s = (df.shape[0], 1)`	{df_3}	{s_6}
`A = np.hstack((np.ones(s), X))`	{s_6, X_5, np_1}	{A_7}
`rtn = np.linalg.lstsq(A, y)`	{A_7, y_4}	{rtn_8}

Figure 1: Example of how statements' read and write sets induce dependencies between cells, which are then used for slice construction. The "sliced" cell is highlighted. Orange arrows represent backward slice construction. Cyan arrows represent forward slice construction.

dynamically-resolved information, the fallback static slicer might conclude that the call to `np.hstack` could modify some internal state to the `np` object, and include that statement in slices that involve subsequent usages of `np`, even if they do not involve `A`.

Switchable Instrumentation. The key property of our hybrid approach to slicing that admits interactive latencies in spite of instrumentation is that this instrumentation can be disabled when performance would suffer (e.g., in tight loops or repeated function calls) and re-enabled when the additional overhead would no longer adversely impact the user experience (e.g., at the start of a new basic block). In order to facilitate this *switchable instrumentation*, it is vital to maintain a correspondence between executing instructions and the program's AST (which is where the static component of slicing operates). It is for this reason that our instrumentation operates at the level of the program's AST rather than at the level of bytecode, since an instrumented AST can more easily reference its uninstrumented counterpart (as such back-pointers can be inserted when the original, uninstrumented AST is transformed). In contrast, bytecode instrumentation operates over streams of Python opcodes, which lack metadata that would allow for constructing a correspondence with the program AST. Although it is possible to get some information such as the current line in the program text, it is not easily possible to deduce, e.g., to which attribute reference a `LOAD_ATTR` instruction corresponds to in a given line, should multiple be present.

Outline. The rest of this paper is organized as follows. Section 2 gives relevant background context on program slicing along with our formal problem definition. Section 3 presents our hybrid static-dynamic dependency resolution techniques for portable and low-overhead program slicing in computational notebooks, which we evaluate in terms of speed, slice size, and correctness in Section 4. We compare and contrast NBSLICER with related work in Section 5 before concluding in Section 6.

2 PROBLEM

In this section, we define requisite terminology and introduce the backward and forward slicing problems in the context of computational notebooks. Figure 1 will serve as a running example on which we anchor the discussion.

2.1 Preliminaries

We begin by defining *forward* and *backward slices*, and show how they are constructed from *data dependencies*.

Definition 1 [Forward and Backward Slices]**.** *A* backward slice *for a cell c consists of the transitive closure over cells that may affect any of the variables read by c. A* forward slice *for c consists of the transitive closure over cells that reference variables that are affected by c.*

In Figure 1, the sliced cell, c_4, is highlighted. The backward slice is constructed by transitively following the arrows pointing from c_4 (colored in orange), and the forward slice is constructed by transitively following the arrows pointing *to* c_4 (colored in blue).

A cell c affects a variable `v` whenever there exists a *data dependency* from variables that are read by c to `v`:

Definition 2 [Data Dependency]**.** *A variable* `y` *is* data-dependent *on another variable* `x` *if the same program statement both reads from* `x` *and writes to* `y`*.*

In Figure 1, for the highlighted cell c_4, there is a data dependency from df_3 to each of X_4 and y_4, and in general every single-statement cell has data dependencies from each variable in its read set to each variable in its write set.

Our notion of data dependency deviates from the traditional definition in that it is *data-centric* (i.e., the dependency exists between pieces of data), whereas the traditional definition is *statement-centric* (i.e., the dependency exists between two statements in a read-after-write / RAW scenario). The two notions are equivalent in that our definition just swap the roles of nodes and edges, so that data take the role of nodes and statements take the role of edges (i.e., the opposite of what is depicted in Figure 1). Our definition focuses on data because data is the component that is more difficult to determine statically, while statements are always known ahead-of-time. Section 3.2 is devoted to discussing how to pin down the data referred to by program statements.

Cell-Level Slicing. For traditional program slicing, one has to consider both data dependencies and *control dependencies* to determine which statements actually get executed; however, in NBSLICER, we only consider data dependencies. This particularity stems from the design decision to construct slices at cell-granularity, since cells already logically separate the notebook and admit a natural granularity to use for backward or forward slices. By designing our slicer to include or exclude blocks of code at cell-level granularity, we circumvent the issue of control dependencies altogether without sacrificing soundness / correctness, since control dependencies never exist between statements in separate cells. If finer granularity is desired, note that we can simply treat each module-level, or *outermost* program statement as being in its own "virtual" cell — affording slices with relatively high resolution despite the lack of control dependencies, since cells in a notebook are themselves typically dominated by basic block code.

This decision simplified the design of our slicer, in that it does not need to consider control dependencies during slice construction; i.e., NBSLICER does not need to consider whether individual statements in, e.g., the body of a loop or `if` statement need to be included — they are rather included or excluded in an all-or-nothing fashion along with the cell. For example, consider this program:

```
lst = list(range(1, 10))   # Cell 1
##########################
if reduction == "power":   # Cell 2
    base = 1
    func = lambda x, y: x * y
else:
    base = 0
    func = lambda x, y: x + y
##########################
for val in lst:            # Cell 3
    base = func(base, val)
```

This example has three cells (demarcated with comments). Indented statements that appear inside of `if` blocks or loop bodies are not considered for slice inclusion independently; thus, the backward slice for cell 3 will contain cells 1 and 2 in entirety, even though only one of the branches of the `if` statement will have executed.

Timestamps. To disambiguate between different versions of the same variable appearing in some data dependency, each variable is associated with a *timestamp* corresponding to the cell that most recently modified it:

Definition 3 [Timestamp]**.** *A cell has timestamp k if it is the kth cell to execute; i.e., its execution counter is k. A variable* `v` *has timestamp k if the most recent cell with a statement to write to* `v` *has timestamp k.*

By our definition of timestamp, there is exactly one cell with timestamp k, which we denote by c_k. If c_k writes variable `v`, we similarly use $\mathtt{v}_k$ to denote its resultant value, when the extra disambiguation is needed (as for the read / write sets of Figure 1).

2.2 Problem Statements

In this work, we focus on two applications of slices in the notebook context: backward and forward. Suppose we have a notebook session S whose execution log is composed of cells $\{c_1, c_2, c_3, \ldots, c_n\}$, and we are interested in computing slice T.

Problem 1 (Backward Slicing)**.** *For any $c_k \in S$, construct a minimal (in terms of the number of cells) slice $T \subseteq S$ such that T includes every cell upon which c_k (transitively) depends.*

In a notebook setting, backward slicing can be used to gather messy code into a coherent script, to aid in later reproducibility. *Forward slicing*, in contrast, can be used to enable *reactive semantics*, wherein dependent cells are automatically re-executed when upstream statements are rerun (also aiding reproducibility, by keeping the notebook state consistent with the program text):

Problem 2 (Forward Slicing)**.** *For any $c_k \in S$, construct a minimal (in terms of the number of cells) slice $T \subseteq S$ such that T includes every cell that (transitively) depends on c_k.*

In order to solve these problems, NBSLICER keeps the metadata presented in Figure 1 materialized in memory and updates it as users execute statements in the notebook. When users want to construct a slice with a given cell c, we perform a "lineage query" in the appropriate direction, either forward or backward, to compute the cells reachable from c (or cells from which c is reachable, respectively).

The success of a slicing algorithm depends on the ability to rapidly and precisely identify data dependencies. Despite prior work for program slicing in Python [16, 27], there are no approaches for doing so *dynamically* in a notebook context, wherein portability and interactivity are vital. In Section 3, we contribute such an approach for dynamically identifying data dependencies. Our approach leverages instrumentation that is both *bolt-on* (not requiring a reimplementation of the Python interpreter) as well as *switchable* (allowing it to be low-overhead).

Soundness and Completeness Assumptions of NBSLICER. In the context of program slicing, a slicer is *sound* if it never incorrectly excludes code that should be in the slice, and it is *complete* if it correctly excludes all the code that should not appear in the slice [26]. Our goal in designing NBSLICER is to improve completeness over static implementations, while maintaining soundness in *most* cases and keeping overhead low.

To maintain soundness, NBSLICER requires that any used libraries external to the notebook be annotated to describe their side effects. We provide a DSL described in Section 3.3 for these purposes. Additionally, NBSLICER does allow trading soundness for completeness in some cases (also described in Section 3.3) as a configuration option. Finally, NBSLICER makes no guarantees regarding soundness when interacting with state external to the notebook, such as files that could be mutated by external processes. We found that, despite these limitations, NBSLICER is still capable of producing useful slices that reproduce desired outputs (see Section 4). Further addressing these limitations provides a useful direction for future work.

3 HYBRID SLICING

In this section, we present the key components underlying NBSLICER; namely, its traditional AST-level static data dependency resolver, as well as its dynamic data dependency resolver, and how these two components work together in the context of NBSLICER's data model to implement a dynamic slicer with static fallback. We show how the dependency resolver, which uses AST-level instrumentation, facilitates portable and *switchable* instrumentation that can be disabled in order to keep overhead bounded, thereby allowing data scientists to use NBSLICER without adjusting their interactive workflows.

3.1 Static Data Dependency Resolution

Before discussing how NBSLICER statically captures data dependencies, we begin by introducing its model used to represent such dependencies.

Data Model. NBSLICER uses `nbsafety`'s [43] abstraction of a "symbol" to capture the basic unit of data capable of participating in a data dependency. In this model, symbols are used to capture data at the resolution of unqualified variable names (e.g., `lst`), as well as data nested as subscripts or attributes (e.g., `lst[42]` or `foo.bar`, respectively). We will use the following code snippet to illustrate:

```
In []:  lst = [1, 2, 3, 4, 5]   # line 1
        lst[3] = 42            # line 2
        y = lst[2] + 7         # line 3
        lst = [6, 7, 8, 9, 10]  # line 4
        y = lst[2] + 7         # line 5
```

Each symbol is associated with up to two namespaces, for representing data nested as either attributes or subscripts. For example, the symbol `lst` is associated with a namespace containing nested symbols for `lst[2]` and `lst[3]`. Symbols nested inside of another symbol's namespace are implicitly data-dependent on that symbol. For example, `lst[2]` is dependent on `lst`.

Slicing with Field-Granular Data Dependencies. In the above example, line 3 has data dependency from the symbol `lst[2]` to `y`. These *record*- or *field*-granular dependencies allow NBSLICER to compute the backward slice for line 3 as just the first and third lines. Field-granular data dependencies allow NBSLICER to capture the property that a write to `lst[3]` will not affect something that depends on `lst[2]`, thereby allowing it to compute smaller slices.

Note that the implicit data dependencies of nested symbols on their parents ensures correctness of slices that involve both nested and non-nested data. For example, the backward slice for line 5 contains lines 4 and 5, since `y` depends on `lst[2]` and `lst[2]` depends on `lst`.

Static Resolution. For simple cases, data dependencies can be inferred solely via statically analyzing code. Consider, for example, the snippet `x = lst[42]`. In this case, NBSLICER's static dependency resolver will operate on an AST `Assign` node, whose target consists of a `Name` in a store context, `x`, and whose right hand side consists of a subscripted `Name` node in a load context, `lst[42]`. For such a case, NBSLICER will draw a data dependency between `x` and `lst[42]`.

However, static analysis can get us only so far in the case of additional dynamic behavior. Suppose the snippet instead uses a variable for the list index: `x = lst[y]`. Here it is still possible to draw the data dependency just by statically inspecting the code, if we have the additional knowledge that `y` is equal to `42` at runtime. However, for the case `x = lst[f()]`, wherein `f` is a function that is called and which eventually returns `42`, we quickly reach the limits of what static code inspection can give us, since in the general case there is no way to know exactly what `f` will do without actually running it.

3.2 Dynamic Data Dependency Resolution

From the example in Section 3.1, we see that NBSLICER must rely on additional instrumentation in order to capture data dependencies from code with dynamic behavior, such as function calls. Providing additional instrumentation without modifying the interpreter (in order to satisfy our portability requirement) is challenging, however. As mentioned in Section 1, one approach built into Python ≥ 3.8 is to attach a tracing handler that runs for each `opcode` event; i.e., for every single bytecode instruction, but this approach introduces unacceptable overhead. In fact, *any* kind of instrumentation introduces significant overhead. The key innovation of NBSLICER for coping with this issue is to make it possible to toggle instrumentation on and off, in order to bound the resultant overhead. When instrumentation is active, NBSLICER uses a dynamic dependency resolver to handle cases such as function calls; when it is inactive, NBSLICER falls back to a static dependency resolver.

We considered trying to develop such "switchable" instrumentation for tracing `opcode` events, but found it difficult practically to connect the stream of running Python bytecode back to the program source code. Such functionality is necessary before we can attach resolved dynamic dependencies to the program's AST, which in turn is needed in order to facilitate dynamic slicing with static fallback. Furthermore, Python opcodes change between different versions of Python, complicating the portability aspect. Though we suspect it is possible in theory to leverage opcode tracing, we found that a much simpler and straightforward approach was to rewrite the program's AST in order to provide switchable instrumentation for dynamic dependency resolution, which we describe next.

Embedded Instrumentation via AST Transformations. The solution we take for NBSLICER is to embed tracing instrumentation directly into notebook programs' source code. Suppose we have two functions, `f` and `g`, which return a list `lst` and the constant `5`, respectively. At a high level, code such as `x = f()[g()]` is rewritten as

```
x = resolve(f())[resolve(g())]
```

where the `resolve(...)` calls handle non-statically-resolvable subtrees of the abstract syntax tree and decorate their roots with symbol information. This symbol information then allows the static dependency resolver to function as a de-facto *dynamic* dependency resolver by short-circuiting when it reaches these non-statically-resolvable subtrees and drawing edges for the dynamically-resolved symbols. This process is depicted in Figure 2. To massage the ASTs into their instrumented forms, our implementation leverages `Pyccolo` [2], a framework providing lightweight and declarative wrappers around Python's built-in AST transformation utilities [3].

3.3 Switchable Instrumentation

While instrumentation is enabled, overhead is high, even for the AST-level instrumentation described in Section 3.2. The key advantage of AST-level instrumentation is that it can be temporarily disabled without requiring changes to the behavior of the data dependency resolver, in order to avoid paying the additional overhead from instrumentation. When enabled for some portion of the program, the corresponding AST is decorated with inferred dynamic dependencies; when disabled, this decoration is absent, and the resolver falls back to conservative static behavior.

When should we disable instrumentation in order to bring performance to a level on par with uninstrumented notebook code, and without affecting the quality of program slices? We developed two heuristics for this decision that work well in practice. These heuristics disable instrumentation (i) for program statements that have run before (e.g., due to their appearance in a loop body or a function that is called repeatedly), and (ii) for calls to external libraries, whose code is defined outside the notebook context. Thanks to these heuristics, the additional overhead from instrumentation is bounded by an amount proportional to the size of the notebook program text. We now discuss these heuristics in more detail, with particular attention to how they interact with data dependency resolution.

Trace-Once Semantics. To reduce instrumentation overhead, NBSLICER only runs a statement with instrumentation the first time the statement in which it is nested runs. This is realized by rewriting, e.g., `for` loops according to the below example:

```
In []:  first_loop_iteration = True
        for ... in ...:
            if first_loop_iteration:
                ... # instrumented loop body
            else:
                ... # uninstrumented loop body
            first_loop_iteration = False
```

If the loop body contains conditionals with multiple branches, any branches not initially taken will have corresponding AST subtrees that lack dynamically-resolved dependencies; for such cases, the dependency resolution will fall back to static behavior.

What about for statements that did run during the first iteration (and whose AST subtrees therefore have resolved dynamic dependencies attached)? The attached dependency information, while accurate for the first loop iteration, may be incomplete. Two choices for handling this incompleteness are to (i) leverage both the incomplete resolved dynamic dependencies in conjunction with static dependencies, or (ii) to just use the incomplete dynamic dependencies in the hope that they are "good enough".

During experimentation on a real corpus of notebook sessions, we found that, empirically, it always sufficed to take option (ii) in practice. For example, consider the below code snippet:

```
In []:  for i in range(len(10)):
            lst[i] += x[i]()
        print(", ".join(lst[i] for i in range(10)))
```

In the above example, the loop body will only resolve dynamic dependencies for `x[0]()`, and not for other entries of `x`. This strategy will fail to produce correct slices whenever sliced code references subscripts of `x` other than `x[0]` without also referencing `x[0]`; e.g., `print(x[1])`. We never observed such cases in our experiments (Section 4), for which NBSLICER always yielded slices that produced output identical to that of the entire notebook session.

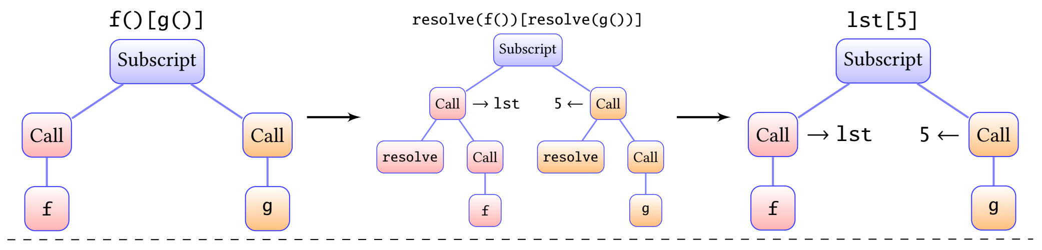


Figure 2: AST transformation and symbol resolution example

```
# builtins.pyi
class list:
  def append(self, e) -> ListAppend[self, e]: ...
  def extend(self, es) -> ListExtend[self, es]: ...
  def insert(self, i, e) -> ListInsert[self, i, e]: ...
  def remove(self, e) -> ListRemove[self, e]: ...
  def pop(self, i) -> ListPop[self, i]: ...
...
# io.pyi
class IOBase:
  def read(self) -> Mutate[self]: ...
  def write(self) -> Mutate[self]: ...
...
# matplotlib.pyplot.pyi
def plot() -> NoEffect: ...
```

Figure 3: Example usage of DSL for extra-notebook dataflow

That said, data scientists who do not wish to trade slice soundness for completeness in this way can configure NBSLICER to take option (i) instead and leverage both static and dynamic dependency resolution for statements that execute multiple times, if desired.

Rule-Based Dependency Inference for Libraries. To reduce overhead, NBSLICER switches tracing off for code external to the notebook (so that API calls to libraries such as, e.g., `numpy` or `Pandas` are uninstrumented). Indeed, it is actually impossible to instrument third-party libraries with extensions written in lower-level code like C or C++ without significantly compromising portability. However, such calls can still introduce data dependencies. NBSLICER uses a simple rule-based algorithm to cope with this reality. We give one example below. Suppose we are running code involving a dataframe `df` with column `A`:

```
In []:  df.A.dropna(inplace=inplace)
        print(df.A.avg())
```

Depending on the value of `inplace`, the call to `dropna(...)` will either return a value, or **`None`**. The default behavior of NBSLICER is to assume that external library calls that return a value do not introduce data dependencies, while calls that return **`None`** will introduce edges from the old object value (i.e. previous value of `df.A` in this case) and any arguments to the new object value. In this way, when `inplace` is **`False`** above, the backward slice for the last statement is just the last statement, but when it is **`True`**, the backward slice for the last statement includes the previous call the `dropna(...)`, which is the correct behavior. Note that without a dynamic dependency resolver capable of checking return values of functions, we would not know which case holds.

DSL for Exceptions to the Rule-Based Resolver. Although our rule-based heuristics are accurate for most calls to external libraries, they do not capture all cases perfectly. Consider the popular matplotlib library [30], oftentimes invoked via calls like `plt.plot(...)`. The function `plt.plot` does not return anything, but its side effect does not mutate the `plt` module object; it only displays figures on the data scientist's screen (overriding any previous such figures). Naïve application of NBSLICER's rules for external libraries would introduce a dependency between old and new versions of the `plt` symbol, resulting in slices with unnecessary cells containing calls to `plt.plot(...)`. To handle exceptions, NBSLICER provides a declarative DSL based on Python annotations for specifying how functions and class methods from modules outside the notebook mutate the caller or other arguments and when they do not. When NBSLICER encounters a call to such a function or method, it first tries to match the arguments specified in the signature to symbol metadata originating from the notebook, and it takes an action corresponding to the return annotation.

For example, according to the usage of the DSL in Figure 3, NBSLICER will not attempt to match arguments given to the `plot` function from the `matplotlib.pyplot` module against notebook symbols, since no arguments are specified in the DSL declaration. According to the return annotation, calls to `plot` should have no effect. The effect of the DSL declaration is thus to override the default rule-based behavior, since calls to `plt.plot` return **`None`**. As another example, for calls to `read` on file handles, NBSLICER will bind the notebook symbol for the file handle to the DSL variable `self`, and it will mark this symbol as being mutated by the method call, even though such calls to not return **`None`**. Finally, certain methods or functions may result in side effects that are too complicated to be specified declaratively — for these, we allow custom handlers to be registered. For example, calls to `list.insert` will leave symbols before the insertion point unchanged, but will shuffle symbols after the insertion point forward in the list's subscript namespace. This logic is encoded in the `ListInsert` handler, to which NBSLICER passes the symbol metadata for the `self`, `i`, and `e` arguments.

One matter we have not yet discussed is the following: when the dynamic slicer encounters calls to `plt.plot`, how does it know that the `plt` object refers to the `matplotlib.pyplot` module, and not some other object for which calls to `plt.plot` *do* mutate the `plt` object? One approach is to eagerly import `matplotlib.pyplot` at the start of the session and compare its identity with the `plt` object whenever a call to `plt.plot` is encountered, but this approach does not scale, as it would require up-front importing of *all* third-library libraries for which exceptions have been specified. Rather, we register a custom finder with Python's `sys.meta_path` whose loader implements a post-import hook [4] that registers the `matplotlib.pyplot` module object as the object to compare against to determine whether the rule for `plt.plot` applies. If the post-import hook has not been activated, NBSLICER never checks for whether the rule applies, since it cannot apply if the `matplotlib.pyplot` module has not yet been imported.

3.4 Notebook Environment Considerations

Notebooks are especially dynamic as programming environments, allowing users to not only rerun existing cells, but to edit, reorder, add, and remove cells. Such operations can impact the code available for reactive execution. Slicing at the granularity of cells also requires consideration of what happens in the case of errors partway through cell execution.

Handling Edits. Edits to cells that follow a given re-executed cell can impact the code availability and ordering for forward slicing (and therefore reactive re-execution), as well as the dependency relationship between cells. In the absence of edits to potentially affected cells, NBSLICER uses the dynamic forward slice directly to determine what cells to reactively rerun, but if edits were detected, it uses a static liveness checker to determine whether potentially affected cells contain live references to any modified data (and that therefore require re-execution).

Handling Errors. When a cell is reactively re-executed as the result of a prior cell execution, it could throw an error. In such cases, it may be desired to stop further reactive execution. This is an implementation detail that we allow data scientists to configure.

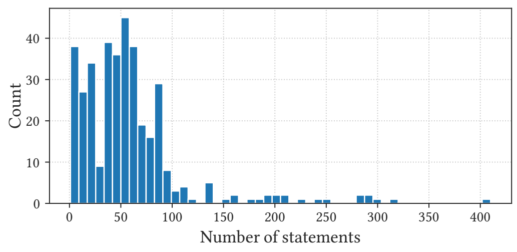
Handling Non-Idempotent Mutations. Some cells may mutate a given variable's value in a non-idempotent fashion; e.g., `a+=b`. Under our current implementation, the cell containing `a+=b` will be rerun whenever the value of `b` changes, but an alternative implementation could require the `a+=b` cell to only reactively rerun when the versions of `a` and `b` referenced were last updated by the same cell(s) as when the `a+=b` cell ran originally, refusing to run otherwise. We leave evaluation of the most user-intuitive behavior for such cases to future work.

4 EMPIRICAL STUDY

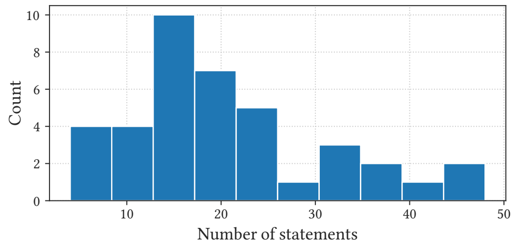
We evaluated NBSLICER's effectiveness in two dimensions: slice size and computational overhead. We ran two sets of experiments: one to evaluate slice size by comparing NBSLICER to `nbgather` [27], a static slicer for computational notebooks in Python, and another to evaluate NBSLICER's computational overhead compared to a regular `IPython` kernel with no extensions. Our experiments are based off the version of the static slicer used in `nbgather` at the time of the CHI 2019 paper [27]. Since then, Microsoft has made some improvements to the `nbgather` notebook extension; however, we were unable to run the latest version of `nbgather` due to incompatibility issues with newer Jupyter versions. For all experiments, we programmatically executed code cells in a Python script instead of manually executing them in the Jupyter interface to eliminate human-created delays between running cells.

4.1 Notebook Data

For each evaluation aspect —slice size and computational overhead— we collected a dataset of notebooks. The first dataset consists of notebook *sessions*, or ordered lists of code cells run by a user while their `IPython` kernels were active. We found that most of these notebooks executed in under a tenth of a second, motivating us to collect a second dataset of more computationally-intensive notebooks to measure NBSLICER's runtime overhead on.



(a) Session Replay Dataset



(b) D2L Dataset

Figure 4: Histograms of statement counts in each dataset.

Session Replay Data. We obtained notebook sessions from the repository collected in the `nbsafety` paper [43]. This repository consists of sessions corresponding to `IPython` notebooks scraped from public GitHub repositories using Github's public search API. Macke et al. implemented a cleaning process to ensure the notebooks did not contain malicicious code or calls to external services (e.g., AWS, Spark, Postgres, MySQL), removing about $\frac{1}{6}$ of the notebooks [43]. The notebook sessions included in our experiments did not require filesystem use (e.g., reads or writes to external files). Each notebook session in the dataset had at least 50 cell executions.

A session begins when the notebook kernel starts, and ends when the kernel shuts down. Session data was extracted from `history.sqlite` files produced by `IPython` and includes information about individual cell executions, including the source code and execution counter for every cell execution. We filtered the repository to include only 374 sessions we could deterministically run without errors. Filtering scripts are available on GitHub [42]. Many of these sessions contain cells with typos, syntax errors, or runtime errors, reflecting programming errors and exceptions encountered as the notebooks were created. Figure 4(a) represents a histogram of the number of statements in these notebook sessions.

Dive into Deep Learning Data. Many of the notebooks collected in Section 4.3 analyze only small amounts of data and have short runtimes. To conduct the computational overhead analysis, we obtained a set of deep learning tutorial notebooks from the Dive Into Deep Learning (D2L) initiative that have longer runtimes. These notebooks do not contain cell replay data; each notebook is only represented by an `.ipynb` file, or a `.json` containing an ordered list of source code cells visible in the notebook environment.

We downloaded the PyTorch notebooks from the D2L repository and successfully ran 71 notebooks locally. Our dataset for the computational overhead experiments consists of these 71 notebooks.

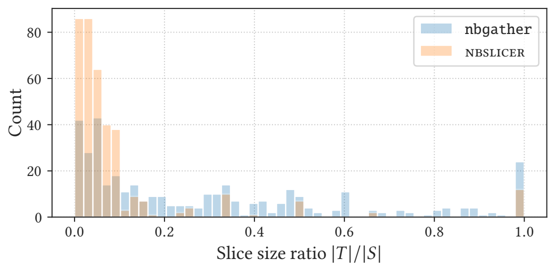


Figure 5: Histogram of slice size ratios captured in all 374 sessions in the session replay dataset.There are 50 evenly spaced buckets.

Figure 4(b) represents a histogram of the number of statements in the notebooks.

4.2 Metrics

We now describe metrics used to evaluate our dynamic slicer.

Slice Size Ratio. For a notebook session S and slice $T \subseteq S$, the *slice size ratio* is defined as the number of statements in that slice divided by the number of statements in the full session, $|T|/|S|$. Assuming equivalent execution results, smaller slice sizes are preferred because they require less time to execute and are more readable.

Session Latency and Overhead. The *session latency* is defined as the notebook execution time under different methods (i.e., under the `nbsafety` kernel with NBSLICER enabled versus the notebook execution time under the regular `IPython` kernel). Measuring the session latency allows us to quantify *overhead* from instrumentation, which we define as the NBSLICER session latency divided by the `IPython` session latency.

Slice Runtime. The *slice runtime* of a slice T, denoted by $\mathcal{T}$, measures the wall clock execution time of the slice under the CPython interpreter (without instrumentation). Measuring slice runtime allows us to measure the end-to-end performance benefit of smaller slices.

4.3 Static vs Dynamic Slicer Results

Both slicers constructed a list of statement dependencies. For each slice, we concatenate the statements to compute slice size.

Slice Correctness. For each of the 374 notebook sessions, we verified that NBSLICER's slices correctly reproduced any printed output of the last statement in the original program. One benefit of our dynamic slicer is that it will not include statements that result in runtime errors in a slice, while static slices might include such statements (for example, if the statement references an undefined variable). Many statements in `.ipynb` sessions can have runtime errors, especially when the notebook author is running a cell for the first time. Only 103 of the statically-generated slices *did not* contain statements with runtime errors, demonstrating the nature of notebook history sessions to contain cells with broken code.

Slice Size Ratios. Figure 5 shows histograms of the backward slice size ratios from the 374 notebook sessions. The "backward" section of Table 1 details statistics gathered for a Wilcoxon signed-rank test for backward slice size ratios (note that the "Δ" column refers to the difference in slice size ratios for a given pair of slices, after which the aggregations in the leftmost column are applied). Overall, backward slice size ratios from NBSLICER are significantly smaller than those from `nbgather` (Wilcoxon test statistic 317.0, $p = 2.0 \times 10^{-39}$).

Table 1: Statistics for `nbgather` and NBSLICER backward and forward slice size ratios and their slice size ratio differences (given in the "Δ" columns) for the session replay dataset. Backward: Wilcoxon test statistic 317.0 and p-value 2.0×10^{-39}. Forward: Wilcoxon test statistic 282.0 and p-value 8.8×10^{-24}.

	Backward			Forward		
	gather	SLICER	Δ	gather	SLICER	Δ
min	0.0053	0.0035	-0.2927	0.0032	0.0032	-0.2386
mean	0.2978	0.1022	0.1956	0.2017	0.0933	0.1084
50%	0.1858	0.0435	0.0704	0.0714	0.0217	0.0000
75%	0.4750	0.0833	0.3333	0.2965	0.0526	0.1333
max	1.0	1.0	0.9756	1.0	1.0	0.9500

We observe that the distribution of slice size differences is long-tailed, with the minimum absolute difference being -12 statements and the maximum absolute difference being 131 statements.

Slice Performance Benefits. To assess the performance benefits of using NBSLICER, we collected the execution times of slices from NBSLICER and `nbgather` and computed the *runtime improvement* for NBSLICER. We denote the set of `nbgather` slice runtimes as $\mathcal{T}$ and the set of NBSLICER slice runtimes as $\mathcal{T}'$. We found that, on average, an NBSLICER slice executed 1.34× faster than an `nbgather` slice (computed as avg($\mathcal{T}$)/avg($\mathcal{T}'$)). As previous work has shown that it is important to mitigate tail latencies in interactive applications [41], we report p90($\mathcal{T}$)/p90($\mathcal{T}'$) and p95($\mathcal{T}$)/p95($\mathcal{T}'$) (i.e., p90 and p95 runtime improvements) of 1.15× and 3.81×, respectively.

Case Study: Unsound Static Slice. In 5 of the 374 notebooks, we observed that the static slicer produces smaller slices than NBSLICER. Upon further inspection, we found that the static slicer occasionally fails to infer data dependencies. We present a truncated example, where the user defines a subclass of `OrderedDict` and adds to it:

```
In [1]:  d = [{"y":132, "o":0, t":0}, {"y":133, "o":40, t":0}]
```

```
In [2]:  from collections import OrderedDict
         class DefaultOrderedDict(OrderedDict):
             def __missing__(self, key):
                 self[key] = list()
                 return list()
         data = DefaultOrderedDict()
```

```
In [3]:  for k in ["y", "o", "t"]:
             data[k].append([o[k] if k in o else -1 for o in d])
```

```
In [4]:  print(data)
```

To reconstruct the output of the `print` statement in cell 4, the dynamic slicer includes all the cells, but the static slicer we used omits cell 3, due to trouble handling a mutating list `append` on a subscripted variable. A static slicer implementation could handle this particular example with a sufficient amount of special casing, but a general solution is out of reach due to Rice's theorem [51].

4.4 Computational Overhead Results

We ran notebooks from the D2L dataset described in Section 4.1 under the `nbsafety` and regular `IPython` kernels. Recall that `nbgather`, which is based on static slicing, uses the `IPython` kernel directly and does not have additional runtime overhead. The median overhead is approximately 2.59.

Table 2: Statistics for session latencies (in seconds) under different methods and the overhead for the D2L dataset. The rightmost three columns show overheads for sessions that lasted at least 1s, 5s, and 10s, respectively. We note that the maximum runtime corresponds to a notebook that downloads some data from the internet; networking delays may account for the difference.

	`IPython`	NBSLICER	Overhead	(≥ 1s)	(≥ 5s)	(≥ 10s)
mean	46.44	47.82	2.39	2.07	1.13	1.06
50%	0.39	1.19	2.56	1.45	1.04	1.02
75%	5.60	7.76	3.05	2.92	1.13	1.11
max	1012	993.0	5.55	5.55	2.07	1.44

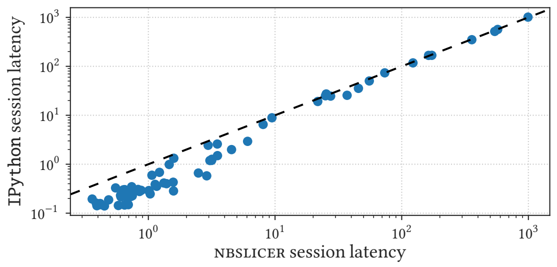


Figure 6: Regular `IPython` vs NBSLICER session latencies for the D2L dataset. Overhead is most noticeable for shorter sessions, after which the latencies are roughly equivalent. Note that the axes are log-scaled.

Session Latencies and Overhead. Table 2 and Figure 6 indicate that NBSLICER session latency is higher than `IPython` session latency for shorter sessions, and roughly equivalent for longer sessions (note the log-scaled axes). There are a few notebooks that execute quickly where the tracing overhead is large, possibly due to networking delays in downloading datasets. Most of our slowdowns probably will not disrupt a user's experience in an interactive notebook setting. For example, the worst case overhead for sessions at most 1 second long was 5.55 — with each session containing at least 50 cells, the additional latency per cell was at most $1/50 * 5.55 = 0.111$s, or 111ms on average, and 100ms is a common heuristic for how much time a UI has to respond for its response to have imperceivable lag [46]. Based on the assumption that end-to-end program execution time is not as important as the ease of experimentation offered by notebook tools, we believe such overhead is acceptable to users. Additionally, although our experimental scripts programmatically executed the code cells in a notebook session, in practice, users manually run cells through an interface, with some time passing in between cell runs. This “think time" [56] would increase the end to end notebook session runtime but not necessarily the NBSLICER overhead, contributing to a smaller effective overhead in practice.

Slice construction times. Since statement dependency graphs are constructed at runtime, computing a slice only requires a depth-first search that can be performed on the fly. The slice construction time represents how long it takes to perform this depth-first search, or compute the slice given the dependency graph. We observed small slice construction times, as described in Table 3. The median slice construction time was approximately 0.3 ms. For notebooks with hundreds of thousands of executed cells, it is possible for the slice construction to take several seconds. However, in practice, such notebooks are unlikely to exist. Slice construction is expected to be negligible, so we did not take measurements for `nbgather` as our goal was just to confirm as such for NBSLICER.

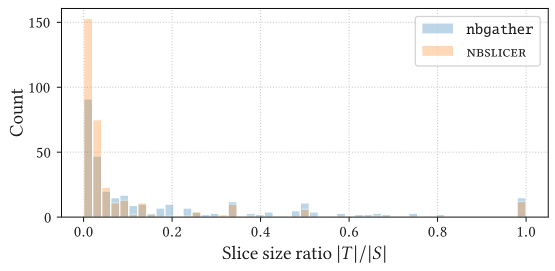


Figure 7: Histogram of forward slice size ratios. There are 50 evenly-spaced buckets.

4.5 Extension: Notebook Reactivity

Reactive notebooks have increased in popularity for data exploration because of their ability to stay up-to-date, like a spreadsheet, at all times [14, 21]. We can use forward slicing to support reactivity: if a user reruns cell c, what cells have statements depending on any of the top-level statements appearing in c and thus need to be run again? Here, we show empirically that a dynamic slicer requires fewer statements of code to rerun.

Methodology. Given data dependencies, NBSLICER computes the forward slice as described in Section 2.2. For our static slicing baseline, we use `nbgather`'s static slicer to get backwards edges in the dependency graph, then reverse directions of the edges. For each notebook in the session replay dataset, we pick a random cell to rerun, compute static and dynamic forward slices, and compare sizes.

Results. Figure 7 shows a histogram of forward slice size ratios for NBSLICER and the static slicer used in `nbgather`. We observe that NBSLICER produces smaller slice sizes, with a median slice size ratio of 0.02 for NBSLICER and a median slice size ratio of 0.07 for `nbgather` (Wilcoxon test statistic 282.0, p-value 8.8×10^{-24}). Detailed results are shown in the “forward” section of Table 1.

5 RELATED WORK

Messy Notebooks. Computational notebooks, being widely used for exploratory programming and data analysis, lend themselves to "messy" programming [27]. Users employ all sorts of poor coding practices while using notebooks, such as storing old versions of code as comments [64] and frequently reordering code cells [36]. Messes accumulate as users iterate on their notebooks, forcing them to refactor their notebooks by deleting and consolidating cells prior to sharing their work with others [54]. This process can be tedious and yield nonreproducible results [52].

Users of notebooks complain that these environments lack adequate support for storing and retrieving historical versions of code [20, 53]. Studies show that data scientists often rely on informal versioning techniques such as copy-pasting code or commenting out code, and these techniques are useful because they support fast retrieval of old versions [32]. Many approaches to provide information about code history are challenging to extend due to the complex nature of dependencies between code, data, and analysis intents in notebooks [34]. Prior works consider automatically versioning code cells and distinct abstract syntax tree (AST) representations at execution time [34, 52]. `nbgather` attempts to clean up versions by performing static analysis on code cells [27]. However, static analysis does not leverage information that could be collected at runtime, such as fine-grained data provenance, motivating NBSLICER.

	construction time
mean	0.583
50%	0.329
75%	0.568
max	6.337

Table 3: NBSLICER slice construction time statistics (ms) for the D2L dataset.

Data Management for Notebooks. Previous work has proposed treating notebooks as dataflow computation graphs [1, 38, 66]. Such work requires restricts flexibility because users are forced to explicitly annotate cells with their ordering or succumb to a forced temporal ordering of cells. `nbsafety` extends these dataflow-based approaches preserves existing notebook semantics (e.g., execution in any order) to keep track of staleness for all symbols in a notebook [43]. `nbsafety` leverages data lineage information to identify stale cells and alert users with potential staleness errors.

The database community has a rich history of work in data versioning and provenance. Coarse-grained provenance is typically explored in data management for workflow systems [9, 15, 19]. For example, Burrito tracks file provenance [23] and noWorkflow analyzes provenance of scripts [45]. While these solutions analyze worklows post-hoc, `nbsafety` and Vizier leverage lineage to track information "online," or at runtime. Computing provenance in an online fashion is beneficial in notebook environments, where users frequently iterate on their code and data. NBSLICER then leverages this provenance information for smaller and more accurate slices.

Time Bounded Instrumentation. A unique aspect of our dynamic slicer when compared with prior dynamic slicing work is that we are particularly concerned with minimizing overhead, as such overhead is exposed to end users since the instrumentation is active during interactive notebook sessions. Prior work introducing instrumentation for performance profiling [8, 25] has encountered similar issues — inserted instrumentation code can add significant overhead, so running it during every iteration of, e.g., an innermost `for` loop can severely slow down the executing process and furthermore over-represent such loops in the profiles [62].

To cope with such challenges, profilers have traditionally relied on *sampling* execution stacks in order to trade off accuracy and efficiency [10, 11, 28, 62]. Such techniques are unfortunately not suitable for slicing. Using terminology from Hirzel et al. [28], sampling-based profilers examine *bursts*; i.e., (non-contiguous) subsequences of the program's entire execution trace. Such bursts can omit program statements that are executed only once, but that introduce dependencies that bridge different parts of the slice. NBSLICER's optimizations, on the other hand, ensure that every module-level programming statement has at least some coverage by ensuring that, e.g., the first iteration of each `for` loop is instrumented. Thus, though the challenges we face are similar, the final applications are orthogonal, and thus motivate different techniques.

Program Slicing. Program slicing techniques have been extensively studied in the programming languages (PL) community to assist with debugging, understanding, and maintaining code [57]. Several studies have shown than conservative static slices are imprecise, and dynamic slices tend to be smaller than static slices [13, 47, 59]. Many existing approaches to dynamic slicing operate at the compiler, bytecode, or VM level [6, 18, 65], which may require a different environment setup. However, Sen et al. argue that "portable" slicing and applicability across different versions of a programming language are desirable for popular programming languages [55]. It is impractical to expect data scientists to compile a specific version of Python instrumented with slicing, link this version to their system path, and configure their development environment appropriately, motivating a "bolt-on" dynamic slicing tool for Python.

Program Slicing for Python. Many popular Python tools leverage static analysis, such as Pylint and flake8. However, dynamic features are widely used in Python [29], and static analysis techniques can fail to produce tight, or short, slices. This can pose problems when users rerun slices that contain code that they did not intend to rerun [12]. Much of the existing work in static and dynamic slicing for Python programs focuses on scripts, not computational notebooks [17, 50, 63]. noWorkflow leverages dynamic program slicing to capture fine-grained provenance in Python scripts but is unable to track dependencies on complex data structures such as lists, collections, or objects [50], which are frequently used in computational notebooks and supported by NBSLICER. We were unable to find any open-source dynamic slicer for Python.

6 CONCLUSION

We introduced NBSLICER, a portable and performant dynamic slicer for computational notebooks. We described its implementation via bolt-on AST instrumentations and discussed how it achieves interactive performance with its hybrid static-dynamic dependency resolver. Finally, for both backward and forward slicing problems, we demonstrated NBSLICER's low runtime overhead as well as its ability to produce smaller slices than a static slicer in a corpus of real notebook sessions. Though our focus in this paper was slicing, our notebook-centric dynamic dataflow analysis techniques could be applied more generally. For example, future work could leverage our techniques to better understand how privacy-sensitive data propagates through data science notebooks and thereby aid in GDPR compliance.

BIBLIOGRAPHY

[1] 2018 (accessed December 1, 2020). *Datalore.* https://datalore.jetbrains.com/.
[2] 2021. Pyccolo: Declarative Instrumentation for Python. https://github.com/smacke/pyccolo.
[3] 2022. AST NodeTransformer. https://docs.python.org/3/library/ast.html#ast.NodeTransformer.
[4] 2022. Python Import System. https://docs.python.org/3/reference/import.html.
[5] 2022. sys: System-specific parameters and functions. https://docs.python.org/3/library/sys.html#sys.settrace. Date accessed: 2022-02-28.
[6] Gagan Agrawal and Liang Guo. 2001. Evaluating Explicitly Context-Sensitive Program Slicing. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Snowbird, Utah, USA) *(PASTE '01).* Association for Computing Machinery, New York, NY, USA, 6–12. https://doi.org/10.1145/379605.379630
[7] Hiralal Agrawal, Richard A DeMillo, and Eugene H Spafford. 1993. Debugging with dynamic slicing and backtracking. *Software: Practice and Experience* 23, 6 (1993), 589–616.
[8] Glenn Ammons, Thomas Ball, and James R Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM Sigplan Notices* 32, 5 (1997), 85–96.
[9] 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.
[10] Jennifer M Anderson, Lance M Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R Henzinger, Shun-Tak A Leung, Richard L Sites, Mark T Vandevoorde, Carl A Waldspurger, and William E Weihl. 1997. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems (TOCS)* 15, 4 (1997), 357–390.
[11] Matthew Arnold and Barbara G Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation.* 168–179.
[12] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2015. ORBS and the limits of static slicing. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM).* 1–10. https://doi.org/10.1109/SCAM.2015.7335396

[13] David W Binkley and Mark Harman. 2004. A survey of empirical results on program slicing. *Adv. Comput.* 62, 105178 (2004), 105–178.

[14] Mike Bostock. 2020 (accessed March 1, 2020). *Observable: The magic notebook for exploring data*. https://observablehq.com/.

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

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

[17] Zhifei Chen, Lin Chen, Yuming Zhou, Zhaogui Xu, William C. Chu, and Baowen Xu. 2014. Dynamic Slicing of Python Programs. In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference (COMPSAC '14)*. IEEE Computer Society, USA, 219–228. https://doi.org/10.1109/COMPSAC.2014.30

[18] Jim Chow, Dominic Lucchetti, Tal Garfinkel, Geoffrey Lefebvre, Ryan Gardner, Joshua Mason, Sam Small, and Peter M. Chen. 2010. Multi-Stage Replay with Crosscut. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Pittsburgh, Pennsylvania, USA) *(VEE '10)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1735997.1736002

[19] 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.

[20] Robert DeLine and Danyel Fisher. 2015. Supporting exploratory data analysis with live programming. In *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 111–119. https://doi.org/10.1109/VLHCC.2015.7357205

[21] Robert DeLine, Danyel Fisher, Badrish Chandramouli, Jonathan Goldstein, Mike Barnett, James F. Terwilliger, and John Robert Wernsing. 2015. Tempe: Live scripting for live data. *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)* (2015), 137–141.

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

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

[24] Alena Guzharina. 2020. We Downloaded 10,000,000 Jupyter Notebooks From Github – This Is What We Learned. https://blog.jetbrains.com/datalore/2020/12/17/we-downloaded-10-000-000-jupyter-notebooks-from-github-this-is-what-we-learned. Date accessed: 2022-02-28.

[25] Robert J Hall and Aaron J Goldberg. 1993. Call Path Profiling of Monotonic Program Resources in {UNIX}. In *USENIX Summer 1993 Technical Conference (USENIX Summer 1993 Technical Conference)*.

[26] Mark Harman. [n.d.]. Carving up bugs. http://www0.cs.ucl.ac.uk/staff/M.Harman/exe2.html. Date accessed: 2022-07-14.

[27] 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.

[28] Martin Hirzel and Trishul Chilimbi. 2001. Bursty tracing: A framework for low-overhead temporal profiling. In *4th ACM workshop on feedback-directed and dynamic optimization (FDDO-4)*. 117–126.

[29] Alex Holkner and James Harland. 2009. Evaluating the Dynamic Behaviour of Python Applications. In *Proceedings of the Thirty-Second Australasian Conference on Computer Science - Volume 91* (Wellington, New Zealand) *(ACSC '09)*. Australian Computer Society, Inc., AUS, 19–28.

[30] J. D. Hunter. 2007. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering* 9, 3 (2007), 90–95. https://doi.org/10.1109/MCSE.2007.55

[31] Sean Kandel, Andreas Paepcke, Joseph M Hellerstein, and Jeffrey Heer. 2012. Enterprise data analysis and visualization: An interview study. *IEEE Transactions on Visualization and Computer Graphics* 18, 12 (2012), 2917–2926.

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

[33] Mary Beth Kery and Brad A Myers. 2017. Exploring exploratory programming. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 25–29.

[34] Mary Beth Kery and Brad A. Myers. 2018. Interactions for Untangling Messy History in a Computational Notebook. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 147–155. https://doi.org/10.1109/VLHCC.2018.8506576

[35] 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.

[36] 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*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3173574.3173748

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

[38] 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)*.

[39] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. *Information processing letters* 29, 3 (1988), 155–163.

[40] 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)*.

[41] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2670979.2670988

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

[43] Stephen Macke, Hongpu Gong, Doris Jung-Lin Lee, Andrew Head, Doris Xin, and Aditya Parameswaran. 2021. Fine-grained lineage for safer notebook interactions. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1093–1101.

[44] Barry McCardel and Glen Takahashi. 2021 (accessed July 8, 2022). *Hex 2.0: Reactivity, Graphs, and a little bit of Magic*. https://hex.tech/blog/hex-two-point-oh.

[45] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. 2014. noWorkflow: capturing and analyzing provenance of scripts. In *International Provenance and Annotation Workshop*. Springer, 71–83.

[46] Jakob Nielsen. 1994. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[47] Akira Nishimatsu, Minoru Jihira, Shinji Kusumoto, and Katsuro Inoue. 1999. Call-mark slicing: an efficient and economical way of reducing slice. *Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002)* (1999), 422–431.

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

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

[50] Joao Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. 2017. noWorkflow: a tool for collecting, analyzing, and managing provenance from python scripts. *Proceedings of the VLDB Endowment* 10, 12 (2017).

[51] H. Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74 (1953), 358–366.

[52] Adam Rule, Ian Drosos, Aurélien Tabard, and James D. Hollan. 2018. Aiding Collaborative Reuse of Computational Notebooks with Annotated Cell Folding. *Proc. ACM Hum.-Comput. Interact.* 2, CSCW, Article 150 (Nov. 2018), 12 pages. https://doi.org/10.1145/3274419

[53] 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.

[54] Adam Rule, Aurélien Tabard, and James D. Hollan. 2018. *Exploration and Explanation in Computational Notebooks*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3173574.3173606

[55] Koushik Sen, Swaroop Kalasapur, Tasneem Brutch, and Simon Gibbs. 2013. Jalangi: A Selective Record-Replay and Dynamic Analysis Framework for JavaScript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 488–498. https://doi.org/10.1145/2491411.2491447

[56] Ben Shneiderman. 1984. Response Time and Display Rate in Human Performance with Computers. *ACM Comput. Surv.* 16, 3 (sep 1984), 265–285. https://doi.org/10.1145/2514.2517

[57] Frank Tip. 1995. A survey of program slicing techniques. *J. Program. Lang.* 3 (1995).

[58] Fons van der Plas. 2020 (accessed July 8, 2022). *Pluto.jl: Simple reactive notebooks for Julia*. https://github.com/fonsp/Pluto.jl.

[59] G. Venkatesh. 1995. Experimental results from dynamic slicing of C programs. *ACM Trans. Program. Lang. Syst.* 17 (1995), 197–216.

[60] Tao Wang and Abhik Roychoudhury. 2008. Dynamic slicing on Java bytecode traces. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 30, 2 (2008), 1–49.

[61] Mark Weiser. 1982. Programmers use slices when debugging. *Commun. ACM* 25, 7 (1982), 446–452.

[62] John Whaley. 2000. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 conference on Java Grande*. 78–87.

[63] Zhaogui Xu, Ju Qian, Lin Chen, Zhifei Chen, and Baowen Xu. 2013. Static Slicing for Python First-Class Objects. *2013 13th International Conference on Quality Software* (2013), 117–124.

[64] YoungSeok Yoon and B. Myers. 2012. An exploratory study of backtracking strategies used by developers. *2012 5th International Workshop on Co-operative and Human Aspects of Software Engineering (CHASE)* (2012), 138–144.

[65] Xiangyu Zhang and Rajiv Gupta. 2004. Cost effective dynamic program slicing. *ACM SIGPLAN Notices* 39, 6 (2004), 94–106.

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