UPGRADVISOR: Early Adopting Dependency Updates Using Hybrid Program Analysis and Hardware Tracing

Yaniv David, *Columbia University;* Xudong Sun, *Nanjing University;* Raphael J. Sofaer, *Columbia University;* Aditya Senthilnathan, *IIT, Delhi;* Junfeng Yang, *Columbia University;* Zhiqiang Zuo, *Nanjing University;* Guoqing Harry Xu, *UCLA;* Jason Nieh and Ronghui Gu, *Columbia University*

https://www.usenix.org/conference/osdi22/presentation/david

This paper is included in the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-28-1

Open access to the Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

UPGRADVISOR: Early Adopting Dependency Updates Using Hybrid Program Analysis and Hardware Tracing

Yaniv David[1], Xudong Sun*[2], Raphael J Sofaer[1],
Aditya Senthilnathan[3], Junfeng Yang[1], Zhiqiang Zuo*[2], Guoqing Harry Xu[4], Jason Nieh[1] and Ronghui Gu†[1]

[1]Columbia University, [2]Nanjing University, [3]IIT, Delhi, [4]UCLA

Abstract

Applications often have fast-paced release schedules, but adoption of software dependency updates can lag by years, leaving applications susceptible to security risks and unexpected breakage. To address this problem, we present UPGRADVISOR, a system that reduces developer effort in evaluating dependency updates and can, in many cases, automatically determine which updates are backward-compatible versus API-breaking. UPGRADVISOR introduces a novel co-designed static analysis and dynamic tracing mechanism to gauge the scope and effect of dependency updates on an application. Static analysis prunes changes irrelevant to an application and clusters relevant ones into *targets*. Dynamic tracing needs to focus only on whether targets affect an application, making it fast and accurate. UPGRADVISOR handles dynamic interpreted languages and introduces call graph over-approximation to account for their lack of type information and selective hardware tracing to capture program execution while ignoring interpreter machinery.

We have implemented UPGRADVISOR for Python and evaluated it on 172 dependency updates previously blocked from being adopted in widely-used open-source software, including `Django`, `aws-cli`, `tfx`, and `Celery`. UPGRADVISOR automatically determined that 56% of dependencies were safe to update and reduced by more than an order of magnitude the number of code changes that needed to be considered by dynamic tracing. Evaluating UPGRADVISOR's tracer in a production-like environment incurred only 3% overhead on average, making it fast enough to deploy in practice. We submitted safe updates that were previously blocked as pull requests for nine projects, and their developers have already merged most of them.

1 Introduction

Powered by agile development methodologies and supported by continuous integration and testing infrastructure, modern software companies achieve blazing fast release cycles, quickly pushing bug fixes and new features to production servers or client devices. For instance, Google's Chrome ships a new major version to the stable channel every four weeks [3], while Facebook publishes updates to their front-end three times a day and releases a new version for iOS and Android every week [7].

A key enabler to this fast development cycle is the large collection of preexisting frameworks and libraries to build on. One open source software (OSS) discovery service tracking popular libraries in leading package managers lists almost 5 million open-source libraries [40]. We surveyed OSS projects developed with prominent interpreted languages[1] (§2) and found that an application, on average, depends on tens to hundreds of frameworks and libraries; these are known as dependencies.

Unfortunately, our survey shows that despite the fast pace of application updates, the adoption of dependency updates is delayed by years, and this delay is getting worse (see Fig. 1 in §2). We believe a key reason behind this dichotomy is the knowledge gap between application and dependency developers. Although dependency developers invest significant effort in creating robust and often backward-compatible updates, they typically have no direct access to the dependent applications, hindering their ability to gauge potential update risks. Application developers want the security fixes and performance enhancements in dependency updates, but lack knowledge of the dependency internals and therefore fear that dependency updates may cause the application to malfunction.

The effect of dependency update delays aggregates across projects and even whole software ecosystems. For a given installation composed of an ensemble of software components, even if only one component requires an older version of a dependency, the entire installation is forced to use the same older version. This older version might accumulate unpatched vulnerabilities over time or break unexpectedly due to deprecation. Moreover, when many older dependency versions are involved, attempts to update subsets of the dependency graph become impossible due to dependency

*Also with State Key Laboratory for Novel Software Technology.
†Also Founder of CertiK with an equity interest.

[1]We surveyed Python, JavaScript, and Ruby projects from GitHub.

conflicts (a.k.a "dependency hell" [30]).

Ideally, an application's test suite should discover any malfunctions due to interactions with dependencies, but this is sadly not the reality. Application and dependency developers strive to make their unit, integration, and system tests have high coverage of their projects. However, state of the art tools for coverage metrics do not examine the difficult-to-measure interfaces between applications and their dependencies. Thus, it is dangerous to rely on application test suites to detect dependency update incompatibilities. The problem is worse for dynamic interpreted languages, as without compilation, API breaking changes not discovered during testing become runtime errors on production servers.

We present UPGRADVISOR, a system for maximizing the safety of and reducing developer efforts invested in dependency updates. UPGRADVISOR is based on the observation that changed dependency code that does not run cannot affect application semantics. UPGRADVISOR works by combining sound static analysis with efficient dynamic tracing to aid developers in the timely adoption of dependency updates. Given a dependency update that developers want to adopt, UPGRADVISOR computes the code difference between its old and new versions and then employs static analysis to discard semantically irrelevant differences and cluster potentially meaningful ones into tracing targets.

To enable this process for modern applications written in widely-used interpreted languages, UPGRADVISOR first builds an over-approximating call graph that accurately accounts for the lack of type information in variables and function arguments in these languages as well as handling implicit language-specific call-site creation features. It then creates a fused abstract syntax tree (AST) representing both the old and new versions of the dependency and tags all changes on a per statement basis. The change tags are propagated up the AST to the call graph, clustering code differences into *call targets* (Python functions or methods) for later tracing. UPGRADVISOR can then statically discard unreachable or semantically irrelevant code changes, such as backward-compatible changes to API signatures and changes in imports location (see §7). If there are no call targets tagged with change tags, the dependency update is safe because it has no changes that can possible affect application execution. Unlike test suites, the static analysis provides complete code coverage, allowing UPGRADVISOR to accurately determine if a dependency update is safe.

While static analysis may be sufficient in many cases to determine the safety of a dependency update, it is conservative, identifying calls not actually used in practice. UPGRADVISOR therefore performs dynamic tracing to determine if call targets with change tags remaining after static analysis actually influence application execution. Dynamic tracing is performed without applying the dependency update and is designed to incur little overhead. Both of these features allow it to be used in a production environment, giving a complete trace of a production server over a substantial amount of time to serve as the ground truth of application-dependency interactions. Running UPGRADVISOR on production servers allows mitigating the inherent unsoundness of dynamic analysis.

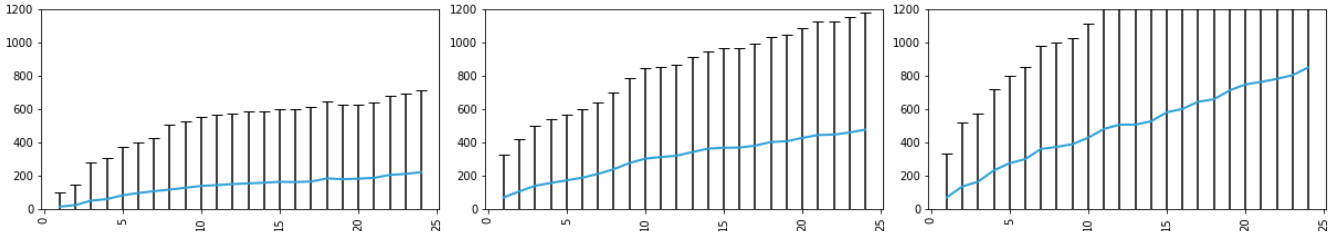
UPGRADVISOR achieves low-overhead tracing using two key mechanisms. First, UPGRADVISOR can select which parts of application execution to trace, tracing only the call targets with change tags identified through static analysis. Second, UPGRADVISOR leverages the hardware tracing module in modern CPUs using a novel coarse-grained tracing technique to collect data only for chosen bytecodes while ignoring unnecessary low-level interpreter instructions. In particular, using our technique, each bytecode branch executes exactly one native branch. Tracing only one branch creates one trace record, reducing tracing data size and runtime overhead. Combining the two allows lowering overhead while retaining precision: we only collect the minimal information required to fully capture control flow in the updated parts of the dependency.

We have built an UPGRADVISOR prototype that supports dependency updates for Python programs. It contains an analysis framework and a tracer implanted into our fork of the Python 3.7 interpreter. We evaluated UPGRADVISOR on 172 potential dependency updates that were previously blocked from being adopted by applications in top-starred OSS repositories on GitHub. The dependency updates include popular frameworks such as `Django`, `aws-cli`, `tfx` and `Celery`. Our results show that UPGRADVISOR is effective. UPGRADVISOR determines through static analysis that 98 (56%) dependencies can be automatically updated, meaning the majority of blocked dependency updates can be adopted without manual inspection. When static analysis cannot completely determine if an update is safe, the analysis reduces the code differences that must still be reviewed by an order of magnitude compared to the overall changes between old and new dependency versions. UPGRADVISOR determines through dynamic tracing that various dependencies not automatically deemed safe through static analysis can still be updated. (see §7.1)

We randomly sampled several dependency updates deemed safe: although we were not developers of either the applications or the dependencies, we were able to quickly submit pull request (PR)s, most of which were subsequently merged by the corresponding developers. The PRs that were merged included dependency updates deemed safe by just static analysis as well as by combining static and dynamic analysis, demonstrating that dynamic tracing can indeed provide additional upgrade opportunities beyond static analysis.

Finally, we performed an extensive performance evaluation, including running a production Django workload published by Instagram and Intel [8]. Our measurements show that our tracer incurs an average overhead of 3%, much lower than other tools.

UPGRADVISOR's code, evaluation datasets, and other resources are available at `http://upgradvisor.github.io`.



(a) Python (b) JavaScript (c) Ruby

Figure 1: The average delay days for all the projects surveyed for each month between August 2019 and November 2021.

2 Survey of Dependency Usage in OSS

Modern applications declare their dependency requirements in metadata files as a list of `(package name, version specifier)` tuples. These direct dependencies also have their own dependencies, creating a graph of transitive dependencies for the application. The version specifier follows a common version structure, "MAJOR.MINOR.PATCH", where MAJOR increments signal API breakage, MINOR increments signal backwards-compatible feature additions, and PATCH increments signal backwards-compatible bug fixes. For example, if the old API is `foo(int a, int b)` and the new one is `foo(int a, int b, int c)` the API was broken. A version specifier in a metadata file can be expressed as conditions, which can directly point to a specific version, also called pinning, or use a combination of lower/upper-bounding terms to define a range of possible versions. Out of a range of allowed versions, the latest one is selected. A given dependency may be specified by the application and by any number of dependencies. All these specifiers must overlap to have a viable dependency set. Using range-defining conditions allow developers to block a version update if they deem it not compatible with their code, e.g., `<=2.5.1`.

Language	Projects	Dependencies					
		Direct			Transitive		
		max	avg	std	max	avg	std
Python	389	118	7.1	11.5	480	15.9	41.3
JS	462	130	17.1	23.3	>1000		
Ruby	501	91	12.3	17.3	548	28.1	103.9

Table 1: Dependency usage in OSS projects on GitHub.

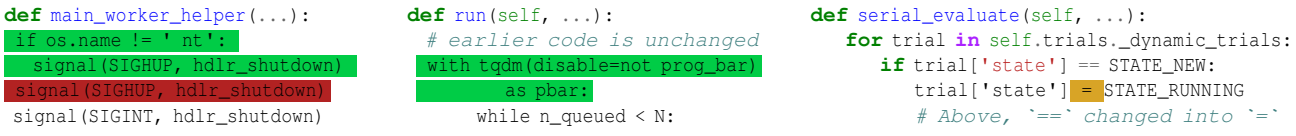
To better understand dependency usage patterns in the leading dynamic interpreted languages, Python, Ruby, and JavaScript (JS), we performed a survey of OSS projects using them. We randomly sampled top starred (>1k) project repositories on GitHub, for which Python, Ruby, or JS, was the primary programming language. Starting from 1,382 Python, 913 Ruby, and 1,144 JS projects, we examined the dependency requirement conditions of the latest version of each project and filtered those with no direct dependents as of November 2021.[2] Table 1 summarizes the results for projects with dependencies, showing the maximum, average, and standard deviation in the number of dependencies per project. Each project is considered an application. For example, Python applications averaged seven direct dependencies and 16 transitive ones for an average of 23 total dependencies per application, but the standard deviations (STDs) show significant differences among applications. The number of direct and transitive dependencies for a Python application was as high as 118 and 480, respectively.

For the 389 Python applications, we examined the dependency requirement conditions for not just the latest version of the application, but also earlier versions published from August 2019 to October 2021. 2% have no restrictions (latest), 29% are lower-bound only, 38% are double-bound (both lower- and upper-bound), and 31% are pinned version specifiers. In other words, more than two-thirds of the version specifiers, double-bound and pinned, may block available updates. A developer whose application may have dozens of dependencies, including transitive dependencies, cannot update dependency X unless every other dependency which depends on X also includes the new version in the specifier.

We measured the historical delay for Python, Ruby, and JS applications in updating their dependencies by examining all versions of the applications published from August 2019 to November 2021. For each released application version, we examine direct dependency requirements. Considering only the dependency versions which existed on the application version's release date, we check if the dependency offered an updated version. If an updated version exists, we consider the application to be delaying updates and measure the number of delay days. Delay days are counted from the dependency's new version release date up to the application's release date. If an application has multiple delayed dependencies, we consider only the most severely delayed dependency.

Fig. 1 shows the delay days for all applications as measured each month from August 2019 to November 2021. We show both the average delay days as well as the standard deviation. For example, Fig. 1a shows that Python applications start from

[2] Unlike JS and Ruby, Python projects declare dependencies implicitly in their setup scripts. We discarded projects when we could not extract dependency constraints.



(a) (b) (c)

Figure 2: Three code change snippets from `hyperopt`'s update from version 0.1.1 to version 0.1.2.

an average of roughly 20 delays days for August 2019 and balloon to reach roughly 200 delay days by August 2021, an order of magnitude increase in delay over two years. Fig. 1 shows that this pattern of increasing delay in adopting dependency updates persists across applications in all languages, indicating that the problem of timely adoption of dependency updates worsens over time. Digging into the data shows that while some projects invest consistently in dependency upkeep, other projects struggle. This difference leads to the significant variations as expressed by the standard deviation bars in Fig. 1. The standard deviation in delay days is so large for Ruby applications that they exceed the visible range in Fig. 1c for most months; the visible maximum was capped at 1,200 delay days to provide a consistent visual comparison across languages while keeping the graphs readable. Because dependency requirements cater to the lowest-common-denominator, having even one such struggling project as a dependency forces the use of an old version.

We designed UPGRADVISOR to address this problem.

3 UPGRADVISOR Overview

We use `Qlib`, a popular Python AI-oriented quantitative investment platform developed by Microsoft, as a motivating example of the dependency update problem and show how UPGRADVISOR solves it. `Qlib` version 0.7.1, released on 15-Sep-2021, relies on 30 direct dependencies. One of them is `hyperopt` 0.1.1, released on 27-Aug-2018, a distributed asynchronous hyper-parameter optimization library for Python.

3.1 An Example Dependency Update Problem

`hyperopt`'s developers changed 828 line of code (LOC) spanning 14 files to go from version 0.1.1 to 0.1.2. Because `Qlib` uses a pinned version specifier "hyperopt==0.1.1", it did not adopt version 0.1.2. Counting the days between `hyperopt`'s version 0.1.2 release on 21-Feb-2019 to `Qlib`'s 0.7.1 release on 15-Sep-2021, the number of delay days for `Qlib` due to not updating `hyperopt` is 937.

To update `Qlib` to use `hyperopt` version 0.1.2, `Qlib`'s developers need to ensure the update is safe. It should not cause `Qlib` to crash, experience other silent failures, or change `Qlib`'s API. A change in `hyperopt`'s output content or structure could propagate to `Qlib`'s output. An update solving a bug in `hyperopt` might benefit `Qlib`, yet still requires `Qlib`'s developers to check for unexpected side effects. Ideally, `Qlib`'s developers can use the opportunity of updating to a new `hyperopt` version to incorporate improvements in `hyperopt`'s functionality they already use or explore its new features.

This process offers the developers a tradeoff between short-term safety by not updating versus investing efforts towards gaining long-term safety and quality. We aim to maximize the safety of the update and its benefits while reducing the developer's efforts required to examine the dependency update.

The easiest way to evaluate the updated dependency is to run `Qlib`'s test suite with `hyperopt`'s new version. It turns out that all of `Qlib`'s tests pass. Sadly, this result is ambiguous as it can not differentiate between the tests not covering `hyperopt` and the update being safe. In fact, measuring the coverage of `hyperopt` when running `Qlib`'s test suites shows that no line in `hyperopt`'s code is covered.

Instead of using its test suite, `Qlib`'s developers can examine all code changes made to `hyperopt` to assess the safety of the update. Fig. 2 shows a few changed code snippets from `hyperopt`'s update. Fig. 2a shows a change in the way worker helpers initialize signal handlers. After the update, when the code runs in a Windows environment, the `SIGHUP` handler is no longer set. Fig. 2b shows a change in the `run` method in charge of running the trial's computations, adding an optional progress bar (controlled by the `disable` flag). Fig. 2c shows a change to the `serial_evaluation` method, turning the condition on `trial['state']`, which was never assigned (effectively redundant code), into an assignment.

Fig. 2a and Fig. 2c are bug fixes, which might benefit `Qlib`, but Fig. 2b constitutes a change to `Qlib`'s CLI, which might break other systems using the CLI output.

The changes in these procedures[3] require human review. Doing this for a few changes may be manageable, but is too difficult for all changes on each update; manual code examination is not scalable.

3.2 Using UPGRADVISOR to Update `Qlib`

UPGRADVISOR is based on the observation that changed dependency code that does not run cannot affect application semantics. If we can show such code is unreachable, we can ignore it. Fig. 3 shows UPGRADVISOR's process for analyzing the dependency update (steps 1-3), employing the tracer (steps 4-6), and gathering and summarising results towards update advice (steps 7-8).

[3] For brevity, we use procedure in place of "method or function".

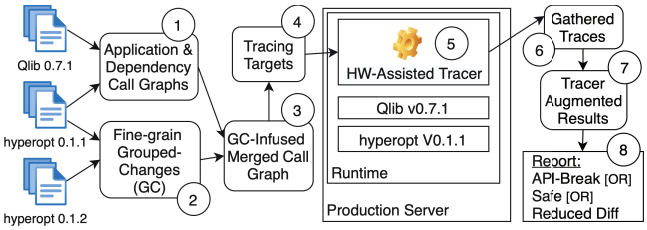


Figure 3: UPGRADVISOR's process of analyzing, tracing and providing update advice for our motivating example.

Analyzing the dependency update. Our analysis goal is to determine statically if the update is safe, or if not possible, reduce the number of changed procedures that need to be tracked by the tracer or examined by a developer. Static analysis involves the following steps.

Step 1: Build call graphs for `Qlib` (the application) and `hyperopt`'s old version (the dependency). Call graph nodes represent procedures, and directed edges represent call relations.

Step 2: Compare `hyperopt` versions to create a fused abstract syntax tree (AST) containing a set of fine-grain change tags. Change tags label the affected AST subtree with the type of change made and the change position in the source code. As we see later, specific change types and location combinations will be handled differently. A change can be a statement modification (e.g., Fig. 2c), an addition of several statements to an existing procedure (e.g., Fig. 2a), or the deletion of a method from a class, possibly breaking any code calling it. We group all change tags in the same procedure because UPGRADVISOR traces at procedural level.

All non-semantic changes, adding a space or changing comment text, are ignored by using an AST representation. Change tags are discarded by employing a language-specific analysis using the AST-subtrees content. For example, for Python, any changes involving type annotations and order changes between unrelated import statements are discarded.

Step 3: Merge the application and dependency graphs, connecting all interfaces between `Qlib` and `hyperopt` in the graph, and infuse the grouped changes into the relevant graph nodes. We discard `hyperopt` nodes that are not reachable from any of `Qlib`'s nodes, along with any change tags connected to these nodes. For example, the changes depicted in Fig. 2a are discarded as no graph path from `Qlib` into `hyperopt` leads to the `main_worker_helper` function.

Starting from 72 changed procedures in `hyperopt`, performing steps 1-3 leaves only four nodes with change tags in the merged graph. Fig. 4 shows part of the merged call graph containing these four nodes, which represent changed procedures. `Qlib`'s only procedure calling into a changed procedure in `hyperopt` is `contrib.tuner.(...)` (in green), calling `fmin`, a part of `hyperopt`'s API (in orange). The changed procedures, e.g., `FMinIter.init` are marked as red stars, while other non-changed `hyperopt` procedures connecting them, e.g., `(...).exhaust`, are shown as well (in red).

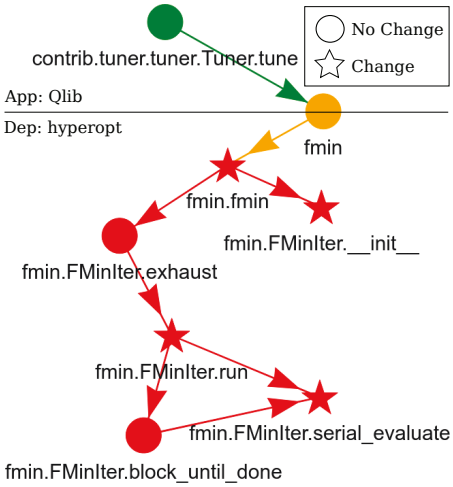


Figure 4: The graph of `hyperopt`'s code changes reduced to only show changes affecting `Qlib`.

Employing the tracer. UPGRADVISOR traces the existing dependency code, ideally running on a production server. These traces can then be used to simulate the dependency update, which can catch breaking changes and discard changes to unreached parts of the dependency.

Step 4: After statically determining the four changed `hyperopt` procedures which might be reachable from `Qlib`'s code, their names are sent on the fly to the tracer already running on the production server.

Step 5: The tracer then starts tracking them by logging every control-flow decision in the procedure, including conditional branches and exceptions. `Qlib`'s test suite did not cover any of `hyperopt`'s code and specifically did not exercise `contrib.tuner.(...)` which calls the changed part of `hyperopt` from `Qlib`. However, if a production environment is not available to trace, the static analysis provides insight on what kind of test cases should be created to provide better coverage. For this example, we manually created a production-like workload which covered calls to `hyperopt` and ran them on the traced system. Specifically, we made `Qlib` use `hyperopt`'s asynchronous computation mode. Tracing relevant methods in `hyperopt` incurs only ~5% runtime overhead on the system.

Step 6: The tracer's output is decoded offline to reconstruct execution traces for tracked procedures.

Gathering and summarizing results. Step 7: The graph created in step 3 is augmented with the collected traces. Any change tag is excluded if its code location is not present in the traces. If all tags in a group are excluded, the whole procedure is discarded. The changed statement in `serial_evaluate`, shown in Fig. 2c, does not exist in the traces, so it is discarded.

At this stage, only three changed procedures, including `run` shown in Fig. 2c, require manual examination. Taking a closer look at the changes in these three methods shows

that all changes relate to the addition of the progress bar in `run`. `fmin.fmin`'s signatures adds a new variable: `def fmin(... , prog_bar=True): ...`. Its value is then propagated to the `fmin.FminIter` class, which then uses it when calling `tqdm(disable=prog_bar)`.

Adding arguments to a function declaration might be a source for API breakage, as a change to required positional arguments might cause a runtime error. In this example, because the new argument has a default value ("True"), a runtime error will not occur. UPGRADVISOR still marks this update as a "possible API break" due to the change in `Qlib`'s output caused by the progress bar. Specifically, this can be avoided by changing `Qlib`'s code to assign "False" to `prog_bar` when calling `fmin.fmin`. Following this, developers can move forward with the updating `hyperopt` to version v0.1.2.

We submitted a PR to the `Qlib` project, recommending the changes described above. This PR was adopted quickly by the maintainers and merged into the `Qlib`'s main code branch within five hours, even though `hyperopt`'s version 0.1.2 had been available almost three years (released 21-Feb-2019) at the time of the PR.

4 Static Analysis of Dependency Updates

As shown in steps 1-3 from Fig. 3, UPGRADVISOR uses static analysis to determine if a dependency update is safe, or identify what procedures may be affected by the update so they can be further considered by dynamic tracing. The inputs are the application code, A, and dependency code D in two versions before and after the update, D_{Before} and D_{After}, respectively.

Throughout this section, we use Python terminology for methods and functions, where a method is a block of code associated with a class and a function is a block of code that can be called but is not associated with a class.

4.1 Application and Dependency Call Graphs

UPGRADVISOR first builds call graphs for A and D_{Before}, which are merged into one graph G. Building an accurate call graph requires: (1) mapping call sites and (2) detecting callees (call targets). However, dynamic interpreted languages such as Python typically do not require specifying types, causing callee uncertainty. Consider the following Python snippet:

```
def foo(a):
    return a.get_size()
```

The function `foo` has an untyped argument `a`, and it calls `a`'s method `get_size`. `a` can be any class that has a method `get_size`, and there is no type information to help narrow down the potential callees. We refer to `get_size` as a named method with an unknown class because the method name called is known but the class to which it belongs to is unknown. Alternatively, consider the following Python code snippet:

```
def foo(a):
    return a()
```

The function `foo` has an untyped argument `a`, and it calls `a`. `a` can resolve to any function in the code, and there is no type information to help narrow down the potential callees. We refer to `a` as an anonymous function. There are ways to explicitly specify types in Python using type annotations [36], as in the following code snippet:

```
def foo(a:arg_type) -> ret_type:
    return a.get_size()
```

However, this is optional in Python, so call graph construction must account for the absence of types.

We use call graphs to decide if an update is safe or identify tracing targets, so their soundness is crucial. While false edges can be tolerated (false positives), there cannot be missing edges (false negatives). We achieve this by over-approximating calls in the graph. The basic idea is to use type information when available to build a context-sensitive [15] call graph to pinpoint the exact method called, but then combine this with context-insensitive analysis for missing targets. We split missing targets into two types: (1) named methods with unknown class and (2) anonymous functions. To express the first type of missing targets in our call graph, we create an edge with a "magic" prefix followed by the callee name, e.g., `UNK.get_size`. To express the second type in the graph, we create a magic edge from the node to `ANON`.

Using the process described above, we construct call graphs for A and D_{Before}, and merge them into one graph $G = (V,E)$ with V nodes and E edges. We split V into two groups depicting M methods or F functions, respectively: $V = M \cup F$. To make G over-approximate for missing call targets we apply the following edge adding rules:

1. $(n,UNK.x) \in E, \exists y.x \in M \Rightarrow E = E \cup \{(n,y.x)\}$
2. $(n,ANON) \in E, x \in F \Rightarrow E = E \cup \{(n,x)\}$.

The first rule adds edges from the respective node to all methods with the same name as the named method with unknown class. The second rule adds edges from the respective node to all functions. These rules add all possible call targets for named and anonymous missing targets. Exploring the Python projects discussed in §2, we find a limited amount of named method missing targets exist in almost every project, while anonymous function missing targets were scarce.

Due to their scripting-oriented roots, most dynamic languages allow placing statements in the source-code file outside of procedures or classes. Running this file as a script or importing it from another file will execute these statements. For example, given a file named "h.py" including `print("Hello World")`, putting the import statement `from h import *` in another file will result in "Hello World" printed on the screen. To represent these statements in the call graph, we place them into a special `module_ctor` pseudo-procedure node and add an edge to relevant importing files.

A graph will contain an edge from a procedure to `module_ctor` if the procedure contains the relevant import statement.

Similarly, we place class fields and their optional initialization in a special pseudo-class-initializer `X_cinit` node, adding edges to and from it between every call site to any class constructor. For example, a statement creating a new class instance, `ClassA()`, placed inside a procedure named `foo` will create the following call path: `foo` → `ClassA_cinit` → `ClassA_ctor`.

We treat other language-specific container for representing code, such as Python's decorators (see §6), similarly.

4.2 Grouping Changes

UPGRADVISOR introduces a novel static approach for creating grouped fine-grain changes. We introduce *change tags*, used for tagging individual statements that have changed between D_{BEFORE} and D_{AFTER} as additions, deletions, or modifications. These fine-grain per statement tags are then grouped together by the lowest-level procedure that contains the respective tags.

UPGRADVISOR fuses the code in D_{BEFORE} and D_{AFTER} into one AST and marks changes with change tags. For example, for Python, we create one AST per Python module. Each module is contained in a file and has procedures, classes, and other statements. The fused AST contains all deleted and added statements, while modified statements contain the code from D_{BEFORE}. For modified code, the code in D_{BEFORE}'s copy is stored in the AST because UPGRADVISOR will later need to identify D_{BEFORE} code when combining it with collected traces generated by running D_{BEFORE}, as discussed in §5. Each change tag represents a change in a statement and contains a pointer and a type, the type being either addition, deletion, or modification. The pointer points to the affected statement, i.e., the lowest statement-tree-node containing the change. For example, in Fig. 2c, the modification tag is applied to AST node representing `trial['state'] = ...`, while in Fig. 2b an addition tag is applied to the node representing `with tqdm(...)`, and no tag is applied to the node representing `while n_queued`. Changes to procedure declarations, such as adding an argument or default value for one, are represented as a tag on the procedure's declaration node in the AST. If a file was deleted or added, we create an AST with all statements and procedure declarations containing deletion or addition tags to represent it. Change tags are then grouped by the lowest procedure, class, or module containing them by following each AST pointer and moving up the tree.

4.3 Clustering Changes Into Call Targets

UPGRADVISOR attaches the grouped changes to nodes in the call graph G, discussed in §4.1. As grouped changes are associated with the lowest procedure, class, or module containing them, it is straightforward to attach them to nodes in the call graph. Any node with at least one change tag attached to it is considered a changed node. Note that changed nodes exclusively appear in the part of G constructed from D_{BEFORE}.

UPGRADVISOR then performs the following two steps. First, it discards change tags that, in G's context, do not affect the semantics of the code. Examples include (1) called APIs adding unused default values, and (2) changes in import location or procedures moving between files. If all change tags in a specific group were discarded, the node associated with this group is no longer considered a changed node. Second, UPGRADVISOR discards any changed node not reachable from an application node. Any changed nodes remaining after this two-step process are marked as call targets, and their corresponding procedures will then be sent to the tracer. These call targets represent changes that can potentially affect the application. If there are no call targets, static analysis alone was successful in automatically determining that the update is safe.

Propagating the indirect effects of direct updates to data is currently out of scope for UPGRADVISOR. These include direct updates to external data used by the code, such as HTML templates, or changes to data in the code itself, such as data used for initialization. UPGRADVISOR can be configured to report on changes to external data. As changes to data in the code are necessarily a changes to the code, these will be detected statically through the call graph if it is reachable from the application, ensuring the correctness of the static analysis. However, any effects due to changed data on other non-changed parts of the code will not be propogated. For example, if a dependency's internal state, such as a global variable, is updated and an unchanged method reads this global variable, UPGRADVISOR will not identify the unchanged method as a call target. UPGRADVISOR can be expanded to propagate the effect of the changed state and mark these methods for tracing or report more methods for developer inspection, and we intend to explore this in future work. As discussed in §7, we find such transitive state changes in the code to be rare.

5 Dynamic Hardware Tracing

UPGRADVISOR uses dynamic tracing to determine what an application actually does in practice. By tracing application execution in a production environment, we can obtain the ground truth of application-dependency interactions and see which call targets are actually used. To allow dynamic tracing in production environments, it is crucial that tracing have minimal impact in production, including avoiding application changes and incurring minimal overhead. For the former, UPGRADVISOR traces the existing application without applying any dependency updates, so no application changes are required. For the latter, UPGRADVISOR introduces two key mechanisms, target-focused tracing and hardware-assisted coarse-grained tracing for interpreted languages.

```
// given a code block with a sequence of bytecode
for (each opcode in code block){
  switch (opcode) {
    case opcode_1:
      subroutine_1(); //interpretation logic for opcode_1
      break;
    case opcode_2:
      subroutine_2(); //interpretation logic for opcode_2
      break;
    ...
    case opcode_i:
      subroutine_i(); //interpretation logic for opcode_i
      break;
    ...
  }
}
```

Figure 5: Original interpretation loop inside an interpreter.

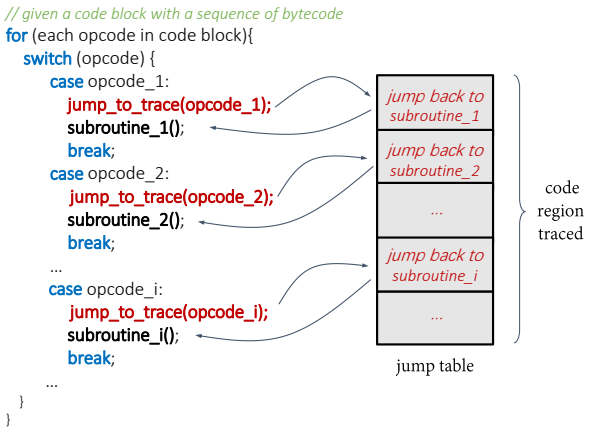


Figure 6: Traced interpretation loop inside an interpreter.

5.1 Target-focused Tracing

UPGRADVISOR does not need to trace the entire application execution, but needs only to trace call targets generated from the static analysis discussed in §4.3. This small handful of methods is not known in advance, and may change for different updates. For languages such as Python, a compiler compiles the program written in the interpreted language to a sequence of bytecode instructions, and the interpreter runs a loop that interprets bytecodes one by one at runtime, as shown in Figure 5. UPGRADVISOR enables on-the-fly selection of which methods are traced by interposing on the interpretation loop used to interpret the intermediate bytecode for dynamic interpreted languages. This logic is illustrated in Listing 1.

```
// given a code block with a sequence of bytecodes
   maintain set of methods to be traced;
   if(signature of code block is in the set)
       { goto traced loop; }
   else{ goto original loop; }
   original loop:
       loop code shown as Figure 5;
   traced loop:
       loop code shown as Figure 6;
```

Listing 1: UPGRADVISOR's target-focused tracing check logic.

We modify the interpreter to allow running a traced version of the loop on demand. UPGRADVISOR maintains a set, updatable during runtime, of signatures for all methods marked for tracing. Before running any method, the interpreted checks if it is part of this set, directing the execution to the traced or original version (where no tracing is enabled) of the loop accordingly. The traced loop is shown in Figure 6, which only differs from the original loop by adding a jump instruction before each call to a subroutine in the interpreter loop, which enables tracing as discussed further in §5.2.

5.2 Coarse-grained Hardware Tracing

To further reduce tracing overhead, UPGRADVISOR leverages hardware tracing mechanisms widely available in modern CPUs, specifically Intel Processor Trace (PT) [21]. Intel PT records dynamic control-flow information such as branch targets and branch taken indications, encoding them as trace packets. With the trace packets collected and the program's native code as input, a software decoder [20] can then be invoked to reconstruct the control flow of the program executed. Although hardware tracing has advantages in terms of low overhead and the absence of intrusiveness, a key challenge is how to leverage it to meaningfully trace interpreted languages since it can only profile native instructions directly running on physical CPUs [43]. For a native program, native instructions can be readily mapped back to the source code with the aid of compilation metadata. This is not the case for programs written in interpreted languages. For interpreted languages such as Python running in a virtual machine, the intermediate bytecode corresponds to the source code, but the native instructions executed by the CPU are those of the interpreter.

To leverage the efficiency of hardware tracing, we need to develop tracing support that can bridge the gap by relating hardware traces generated by CPU to bytecode instructions of interpreted languages that developers can understand. A naive way to obtain the execution trace at the bytecode level is to trace the execution of the entire interpreter code and then reconstruct the execution flow of high-level bytecode based on the mapping between bytecode types and their respective interpreter subroutines. For example, Intel PT generates trace packets with instruction pointers (IPs) to identify the address range for each instruction. In Figure 5, interpreter subroutines such as `subrountine_1` and `subrountine_2` have static address ranges for their instructions. Knowing that the executed instructions are within the address range of a particular function suggests which bytecode opcode is being interpreted.

Unfortunately, this approach may suffer from data loss as it can record a huge amount of unnecessary low-level trace data.

Intel PT uses a memory buffer to store trace data. Data loss occurs when there is more trace data generated than can be written into the buffer. It is extremely challenging to determine after the fact what data is lost and how to recover it [43]. However, what we are interested in is only the sequence of bytecode instructions executed, not the low-level control flow of the interpreter subroutines. What is needed is a *coarse-grained* tracing mechanism that focuses on the collection of the high-level bytecode sequences without capturing extraneous details of the subroutine implementations.

To this end, we developed a novel coarse-grained tracing mechanism that avoids capturing low-level interpretation instructions. We leverage a feature of Intel PT that allows trace packets to be filtered based on their IPs. An address range can be specified such that packets whose IPs are not in the range will be filtered out by the CPU. We create a trampoline (i.e., *jump table*) and use it as a special memory region that allows us to quickly filter out irrelevant instructions while retaining those that correspond to the bytecode. As shown in Figure 6, the jump table consists of a sequence of contiguously allocated *tablets*, each corresponding to a particular opcode. A tablet contains only one single jump instruction that jumps back to the call to the subroutine for the opcode. The traced interpretation loop has a jump instruction before each call to a subroutine in the interpreter. This instruction takes the control to its corresponding tablet; executing the instruction in the tablet takes the control back to the interpreter code. Essentially, the interpreter takes a "detour" to visit a specific (a priori known) address range defined by the jump table. We use this address range to allow Intel PT to filter out all instructions whose IPs are not in the range. As a result, the trace that PT ends up generating contains only the executed jump instructions in the tablets, and these instructions immediately reveal the bytecode opcodes due to their one-to-one mapping.

5.3 Gather Trace Results

Once hardware traces are collected, we decode them offline to reconstruct the dynamic control-flow of the program execution and deduce the code executed at runtime. The decoder decompresses the hardware trace data as a sequence of executed jump instructions, each corresponding to one tablet in the jump table. Using the one-to-one mapping between tablets and bytecodes, we reconstruct a partial sequence of bytecodes interpreted at runtime. Using the static control-flow graph for each traced method and partial bytecode sequence we project the sequence of bytecodes onto the graph so as to reconstruct the dynamic control flow executed. Once the concrete dynamic control-flow is determined at the bytecode level, we then leverage the available compilation metadata to obtain the exact lines of source code executed.

We then return to the call graph discussed in §4.3 and discard additional changed nodes based on the trace results. Specifically, UPGRADVISOR discards any change tag not associated with a statement present in the traces. Any remaining changed nodes are used to create a reduced diff file, containing differences between D_{BEFORE} and D_{AFTER} where only reachable changes appear. This reduced diff file is then made available to the developer for further examination to determine if the update is safe for adoption. If there are no changed nodes remaining, the update is considered safe.

The current version of UPGRADVISOR lacks support for exceptions. Once an exception is raised, the exception mechanism's unusual execution flow affects the control-flow reconstruction mentioned earlier. In future work, we would like to support exception handling. In brief, an exception redirects execution to a dedicated block inside the interpreter. This block is responsible for directing the execution flow back to the corresponding exception handling bytecode determined by the point where the exception occurs. Supporting hardware tracing of exceptions requires tracing that redirection block to bridge the exception control flow gap.

Apart from interpretation, certain language runtimes also enable just-in-time (JIT) compilation mode for the sake of performance. Our design focuses on interpreted mode. Adding a similar design to the one we proposed by [43] will allow for hardware tracing JITed code.

6 Implementation

We have implemented an UPGRADVISOR prototype for Python 3 applications. We built the static analyzer on top of Pyre-check [9], a type-checker for Python 3. Pyre infers missing types and generates a set of calling targets for each call site it soundly resolves. For non-resolved targets, we inserted the magic edges explained in §4.1. To perform an AST-based code comparison, we used `GumTreeDiff` [10], a state-of-the-art code differencing tool employing its JSON-edit scripts creation function to help generate fused and tagged AST.

UPGRADVISOR handles Python decorators [35] by defining them as procedures so they are represented as nodes in the call graph. For example, given a function `bar` decorated with `@dec`, a function `foo` calling `bar` will result in the following graph path: `foo` → `dec` → `bar`. We leave for future work a more subtle analysis allowing separation of the different parts of the decorator logic (i.e., set up, wrapper and cleanup) and subsequent graph edge creation. Any change (add, modify or delete) to a procedure's decorator or its arguments is handled similarly to a procedure declaration change.

We built the hardware tracer on top of CPython [12], the default and most widely used interpreter of Python. In CPython, the interpretation functionality is directly written as a loop in C code and Python code is compiled into executables once the interpretation starts. We modified the interpretation loop as explained in §5. Instead of allocating a buffer, we statically inserted a trampoline block (equivalent to a jump table) into the interpreter's codebase. As CPython does not feature any JIT-related optimizations, we only need to monitor bytecodes

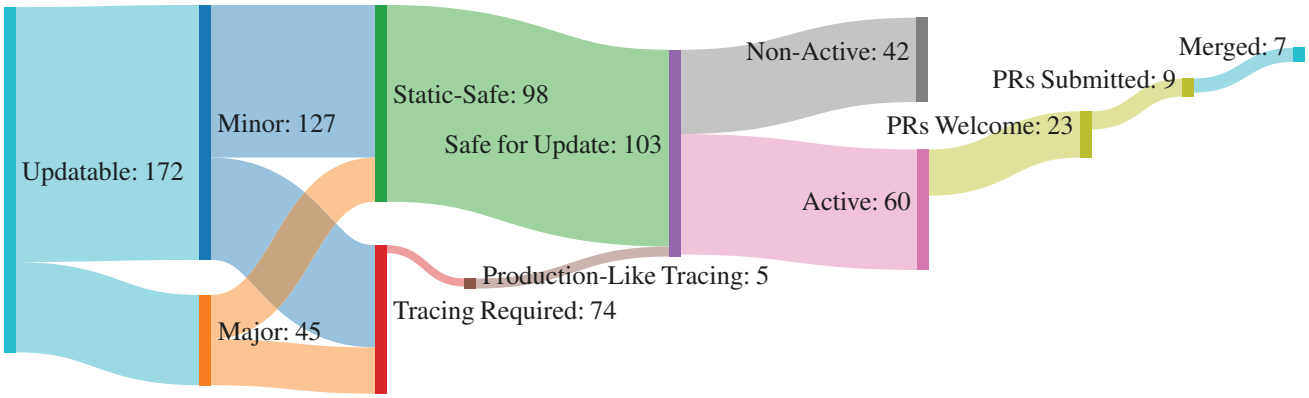


Figure 7: UPGRADVISOR's effectiveness on 172 dependency updates. Its hybrid static and dynamic analysis identified 102 updates as safe. A sample of safe updates were submitted as PRs, almost all of which have been merged.

resulting in control-flow divergence. Five kinds of Python bytecode are taken into account: *POP_JUMP_IF_FALSE*, *POP_JUMP_IF_TRUE*, *JUMP_IF_FALSE_OR_POP*, *JUMP_IF_TRUE_OR_POP*, and *FOR_ITER*. Each of them has two potential branches, true or false. Thus, the trampoline block has ten tablets.

The current prototype supports only applications written entirely in Python. Performance-minded Python projects may convert computation-heavy code into C. A prominent example is `numpy`, a scientific computing package. We leave extending UPGRADVISOR's approach to mixed language projects for future work, and currently UPGRADVISOR will alert and stop processing when C code is detected in the project. The prototype supports tracing only on bare-metal machines. To extend it to run in a virtualized environment (e.g., VMs or containers) will require OS support and further changes in memory mappings for tracing. We note that Intel already added initial support to KVM [19], and leave the rest for future work.

7 Evaluation

We evaluated the effectiveness of UPGRADVISOR in adopting blocked dependency updates and its performance overhead. We first used UPGRADVISOR to examine possible Python dependency updates from our survey discussed in §2. Although the vast majority of the 389 Python applications blocked dependency updates, we only considered those written entirely in Python 3. Altogether, we examined 50 applications with 172 possible dependency updates. We further tested UPGRADVISOR's ability to detect API breakage using known API changing updates. We then measured the performance overhead of UPGRADVISOR's tracer using a subset of the 50 applications with available performance test suites. Finally, we also measured UPGRADVISOR's tracer performance using Instagram's `django-workload` [8], based on a real-world large-scale production workload.

Static analysis was done on a machine with an AMD Opteron 6168 CPU (48 cores) and 62GB of RAM. Dynamic tracing was done on a machine with an Intel i7-10700 CPU (8 cores) with 16 GB of RAM. All machines ran Ubuntu 16.4.

7.1 Facilitating Dependency Updates

We evaluated UPGRADVISOR's ability to adopt 172 previously blocked dependency updates for 50 GitHub projects, including `Django`, `aws-cli`, `tfx` and `Celery`. Some of these projects were also dependencies for other projects. When the latest version of a project blocked a dependency update, by pinning or double-bounding dependency requirement conditions, we explored the possibility of removing the block and updating it to the next version of the dependency. For example, in our motivating example presented in §3, `Qlib` v0.7.1 pinned the dependency `hyperopt` to version v0.1.1, while version v0.1.2 exists. Out of these 172 possible updates, 45 were major version updates, and the other 127 were minor. Fig. 7 depicts the high-level view of this process.

UPGRADVISOR's static analysis was able to determine that the majority of dependency updates, 76 minor and 22 major, were safe and could be automatically updated without further dynamic tracing. These 98 updates are marked as "Static-Safe" in Fig. 7. Referring back to our survey for update delays in Python, Fig. 1a, performing all of these updates to the next available dependency version would save an aggregate of 11,310 delay days, averaging 115 delay days saved per dependency. We further confirmed the "Static-Safe" results by sampling roughly 10% of them, 11 to be exact, and manually validated that the code changes were safe.

We measured the reduction in code differences that still remained to be considered after static analysis versus the entire code differences of the updates. The total number of diff lines in all 172 updated versions we considered for this experiment was 667,604, with the average update constituting 3,881 diff lines (STD 9,078). While not a perfect metric, we use diff size as a proxy for manual developer effort required to study a de-

Project (Dependency)	Diff (LOC)	% Discarded		
		Static	Dynamic	Total
AutoML (distributed)	850	95	5	100
Electrum (qdarkstyle)	641	88	8	96
Flair (gdown)	1500	71	29	100
Qlib (Hyperopt)	828	90	9	99
Scylla (requests)	449	90	8	98

Table 2: Diff reduction for dependency updates, showing diff size in LOC and the percentage of lines discarded statically, using dynamic tracing, and in total.

pendency update. UPGRADVISOR's static analysis was able to reduce the diff sizes by an average of 91%. The reductions are consistently large across updates, with a standard deviation of 17.58%. These reductions also count cases in which UPGRADVISOR finds the update safe, eliminating the whole diff file.

We also quantified the prevalence of direct changes to data such as global variables that could potentially be used by unchanged methods. We found that only 10 out of the 172 updates contained such transitive state changes, indicating that they are infrequent. Furthermore, UPGRADVISOR was able to statically determine 5 of the 10 as safe, so only the remaining 5 still requiring dynamic tracing could be impacted by the current limitation of UPGRADVISOR not identifying unchanged methods using changed data.

Among the remaining 74 dependency updates that could not be resolved statically, denoted "Tracing required" in Fig. 7, we selected a representative sample to evaluate further using dynamic tracing. The specific projects and dependencies evaluated are listed in Table 2.

Unfortunately, we did not have access to actual production environments for these applications, so we used the results of the static analysis to help construct production-like workloads to cover application-dependency interactions for these applications. For `AutoML`, an automated machine learning framework, we ran selected sk-learn tutorials. For `Electrum`, a GUI-based Electrum Bitcoin wallet, we manually interacted with the GUI to try and trigger the relevant parts of the dependency code. For `Flair`, a framework for state-of-the-art (SOTA) Natural Language Processing (NLP), we used publicly available datasets for multiple supported languages (used for training), employed trained models, and ran tutorial examples. For `Qlib`, we set up a MongoDB instance to allow `hyperopt` to conduct asynchronous hyper-parameter optimization, and generated testing inputs for various optimization calculations, as discussed in §3.2. For `Scylla`, a proxy search and connection tool, we scanned for available proxies and used them to crawl major news sites. When applicable, to further increase coverage for possible program behaviors, we used inputs included by the project or created in our environment to drive the `atheris` fuzzer for Python [14].

Table 2 shows the results of running UPGRADVISOR end-to-end process on the project's production-like environments.

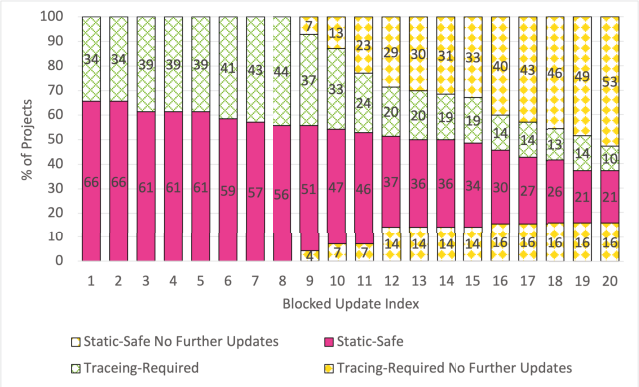


Figure 8: Using UPGRADVISOR on 75 application-dependency pairs with eight or more blocked updates.

On average, using the tracer further reduced diff sizes by 12%. Furthermore, the tracer allowed for classifying more updates as safe. For other updates, e.g., `Qlib`, additional manual inspection was required as not all code changes could be discarded from dynamic tracing, but only ~2% of the original code changes required manual inspection, significantly reducing developer effort in adopting the dependency update.

7.2 Analyzing Multiple Blocked Updates

When applications fail to perform their dependency's first update, subsequent updates are blocked as well. Among the 172 blocked dependency updates, the number of blocked updates per dependency is 12.5 on average, the median being 5, with a standard deviation of 43.67. For example, by pinning `hyperopt` to version 0.1.1, `Qlib` blocked eight updates, from 0.1.2 to 0.2.6. More generally, among the 172 blocked dependency updates, there are 75 dependencies with eight or more blocked updates.

Fig. 8 shows the result of using UPGRADVISOR on each of the eight or more blocked updates for the 75 dependencies. The blocked update index indicates how many versions after the adopted dependency is the update being considered. For example, the first bar shows the next version of the dependency, which is the subset of results from the study in §7.1 limited to just these 75 dependencies. For each blocked update index, we show the percentage of updates UPGRADVISOR requires tracing for as opposed to deeming safe statically. Starting from 34%, this percentage steadily increases to 44% in the eighth update, constituting a ~30% increase. If we count blocked updates as retaining their previous status (static-safe or tracing required) when no further updates are available, this trend continues as the blocked update index increases from 9 to 20.

To test UPGRADVISOR's hybrid approach contribution to the analysis of multiple blocked dependency updates, we employ our production-like testing environment to `Qlib`'s `hyperopt` dependency for all available updates. Fig. 9 shows diff sizes and UPGRADVISOR's ability to statically and dynamically

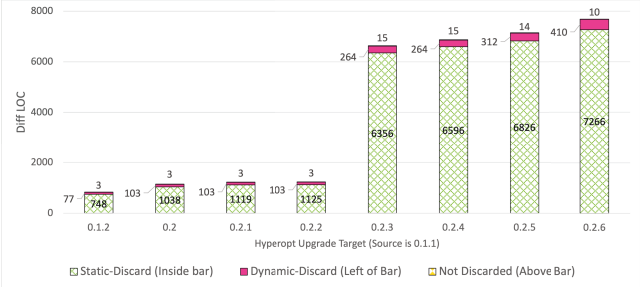


Figure 9: Diff sizes and static and dynamic discards for `hyperopt`'s eight updates.

discard changed code across `hyperopt`'s eight updates. Note that the first bar represents the same `Qlib` data as in Table 2.

7.3 Contributing to the OSS Community

To further validate our results, we selected a sample of dependency updates that UPGRADVISOR considered safe and submitted them to the respective project via a PR. Except `Qlib`, which was the first PR we submitted, all other dependencies were updated to their latest version. As submitting a PR requires manual effort, we focused on active projects welcoming PRs. We deem projects active if their latest commit was made in or after 2021, and PR-welcoming if they accepted a PR from an external developer in the last month and have less than 100 open PRs. Each PR clearly explained UPGRADVISOR's goals and affiliation, provided UPGRADVISOR outputs (e.g., graphs such as the one shown in Fig. 4), and any other relevant information (e.g., dependency change log) allowing the developers to examine the updates and validate our results. In some cases, our PR prompted discussions with the developers providing us with ideas for improving UPGRADVISOR's outputs. Out of nine PRs submitted, seven were merged and two received no response. Furthermore, five of the merged PRs were for dependencies listed in Table 2, validating the results of UPGRADVISOR's dynamic tracing.

7.4 Detecting API Breakage

We noticed that in OSS projects, API breakage is discovered by dependency users in a few days/weeks. The relevant version will quickly be "yanked" from the repositories, so that the API breaking version ends up not being visible in our experiments in §7.1. As a result, none of the dependency updates considered in §7.1 caused API breakage. While this shows the advantages of OSS, for the individual entities, this discovery might have been made at the price of production failures or even data corruption, and UPGRADVISOR's goal is to detect these before they happen.

To evaluate UPGRADVISOR's ability to detect API breakage, we conducted a small controlled experiment with two applications, `django-oscar` and `label-studio`, which were examined by UPGRADVISOR in §7.1. These applications have a dependency on `Django`, which has a well-documented deprecation timeline [4] allowing us to study API breakage. We consider the recent 7-Dec-2021 release of `Django 4.0`, which contains 28 API breaking changes including arguments losing default value, removed APIs, etc. Both `django-oscar` and `label-studio` are stuck on much earlier 3.x versions of `Django`. Instead of considering an update to the next available 3.x version of `Django`, we used UPGRADVISOR to statically analyze the difference between version 4.0 and the 3.x version specified by the application. In these cases, UPGRADVISOR correctly identified all API breaking changes with no false positives or negatives, which we manually confirmed by studying UPGRADVISOR's output and comparing it to the deprecation information. This experiment also showcases UPGRADVISOR ability to direct developers to the relevant portions of their code which will break and provide context for the fix.

7.5 Tracing Overhead

We evaluated UPGRADVISOR's tracer overhead using applications from our previous experiments in §7.1 with test suites that we could set up and execute without errors. Ironically, some test suites failed to run due to broken or conflicting dependencies. We selected a subset of qualifying projects to represent the Python open-source eco-system, including ML (`Qlib` and `Flair`), data-science (`Faust`), blockchain (`Electrum` and `Vyper`), administration tools (`aws-cli`), and website-building (`Django`). `Django` allowed us to experiment with multi-process code and control the number of processes used. We ran Django's test suite using 1, 8, and 16 logical CPUs. Each project had some dependency update among the 172 possible updates considered in §7.1. Specifically, the dependency updates for `Qlib`, `Flair`, `Faust`, `Electrum`, `Vyper`, `aws-cli`, and `Django` were `hyperopt`, `gdown`, `Croniter`, `qdarkstyle`, `asttokens`, `colorama`, and `pytz`, respectively.

We compared the performance of UPGRADVISOR to several other tools, including `cProfile`, `Coverage.py`, and `JPortal4Py`. `cProfile` is a de-facto standard tool for cPython that profiles executions at the method-level. `Coverage.py` is a de-facto standard tool for cPython that tracks statement-level test coverage. Neither of them provide the same functionality of UPGRADVISOR's tracer, but provide useful performance comparisons. `JPortal4Py` is a Python-compatible implementation of a hardware tracer that traces the whole interpreter [43]. We also compared against UPGRADVISOR-SW, an implementation of UPGRADVISOR's tracer that uses software tracing in lieu of Intel PT to trace all procedures. In evaluating UPGRADVISOR, we compared two configurations, UPGRADVISOR-ALL to trace all procedures, and UPGRADVISOR-Targeted to trace only procedures marked by UPGRADVISOR's static analysis. We ran each application on each tool five times and report the average and standard deviation of the overhead measurements.

Fig. 10 shows the performance overhead measurements

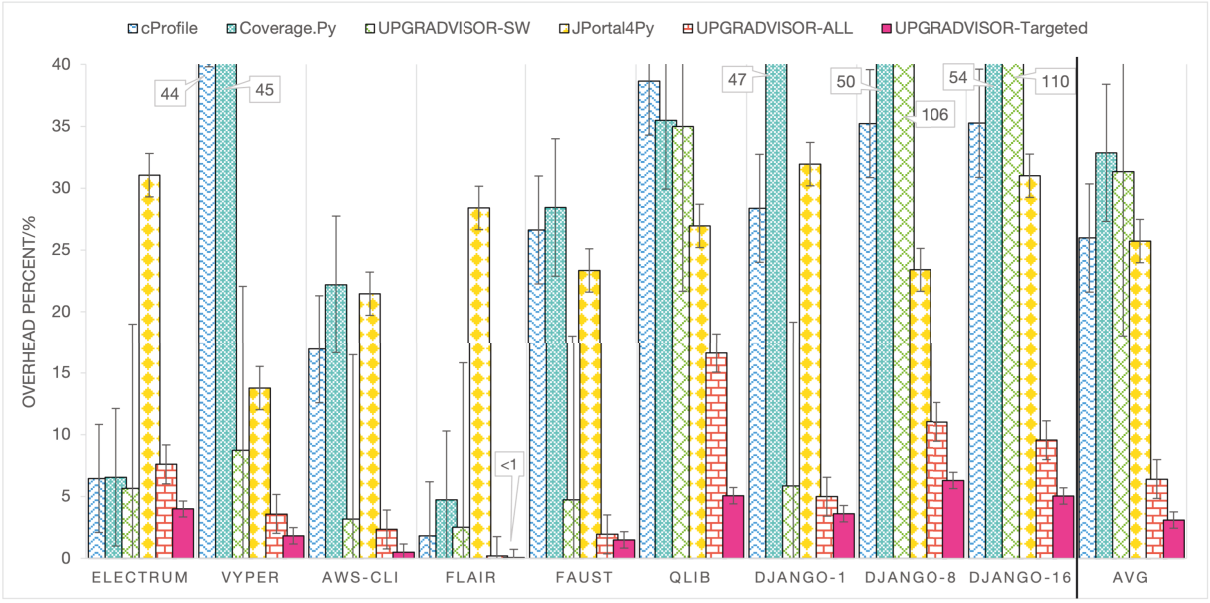


Figure 10: Comparing the performance of UPGRADVISOR's two modes, ALL and Targeted, with `cProfile`, `Coverage.py`, `JPortal4Py`, and UPGRADVISOR-SW, a software-only tracer.

normalized to native execution of the application without any tracing. UPGRADVISOR in targeted mode has the least overhead in all cases, averaging 3%, with a standard deviation of 2.15%. It is an order of magnitude faster than all other tools for some applications, except for UPGRADVISOR-ALL. Django's multiprocess test-suite measurements showcase the advantages of using hardware features for tracing, as all software-based approaches suffer from significant overhead trying to record all control operations made by the interpreter across several processes. Nevertheless hardware tracing is not a panacea as the `JPortal4Py` hardware tracer performs much worse than UPGRADVISOR-SW on most of the single process measurements. This is because `JPortal4Py` traces the whole interpreter as well, flooding the memory buffer with trace packets and causing significant disk I/O.

While tracing all methods, UPGRADVISOR-ALL manages to only incurs an average of 6.4%, over 60% worse than UPGRADVISOR-Targeted but still much better than all other tools. However, because it traces many more methods and fills up the memory buffer quickly, it suffers data loss, which can lead to misdiagnosing unsafe updates as safe. Data loss measures lost tracing events, those overwritten before they could be read from memory by the CPU and written to disk, as a percentage of all tracing events. We calculated data loss rates by comparing UPGRADVISOR-ALL versus UPGRADVISOR-SW, which also traces all methods but does not suffer the data loss of hardware tracing. UPGRADVISOR-ALL's data loss rates across the different applications rose as high as 16% for single process workloads and over 20% for Django running with 16 logical CPUs. In our experiments, we set a memory buffer size limit of 128MB per logical CPU. Increasing this limit or using faster disks/memory might help convert some data loss into overhead. In contrast, UPGRADVISOR-Targeted does not suffer from any data loss due to the reduced amount of trace records generated.

To further stress UPGRADVISOR's tracer, we used Instagram's `django-workload` [8]. This testing environment includes a Cassandra database [2], `memcached` [27] in-memory key-value instance, a Django installation and the Siege load generator [13]. We set up Django according to its recommended configuration for production systems [6] using the WSGI interface. Django depends on `pytz`, a frequently updated package dealing with time-zone related date manipulations, and supports thousands of plugins and sub-packages [5], including `django-cassandra-engine` used by `django-workload`. We measured the performance of UPGRADVISOR's tracer using `django-workload` when evaluating updates to both `pytz` and `django-cassandra-engine`. Running this workload using both UPGRADVISOR-ALL and UPGRADVISOR-Targeted, we found that UPGRADVISOR incurs an average overhead of only 7% and 3%, respectively. These results are consistent with those in Fig. 10, and indicate that our measurements of UPGRADVISOR's tracer overhead provide a good indication of its expected performance when running real-world production workloads.

8 Related Work

Dependency upgrade surveys. Other surveys also show that many projects suffer from dependency update delays [23, 38, 41]. For example, a survey of 7.3K Java projects

reports that 81.5% of projects display dependency update lag [23], and a survey of 610K JS projects in the NPM package repository between 09-11-2010 and 02-11-2017 reports a similar number of delay days as ours [41]. Our survey focuses on three modern dynamic languages and investigates historical dependency upgrade patterns.

AST differencing algorithms. AST differencing algorithms [10, 11, 16, 29] compute an edit script between two versions of an AST. GumTree [10] first finds isomorphic subtrees through a greedy top-down algorithm then executes a bottom-up algorithm to match sub-trees which share a large number of matching nodes. As discussed in §6, UPGRADVISOR uses GumTree's AST-diffing and builds upon its generated edit-script to generate a fused AST representing the dependency before and after the update.

Changeset and impact analysis. Given a set of code changes and test suite runs, change impact analysis tools generate a list of tests affected by the change and re-test them to verify if they pass after the change is adopted. Approaches can be classified based on the techniques used, the granularity of changes considered, and whether static or dynamic analysis is used [24]; only one approach explored statically studying changes at the code-snippet scope (below the method/class level) [32]. Chianti [33] introduced a change impact analysis tool for Java programs, incorporated in the Eclipse IDE. Prior techniques all rely on application test suites and do not scale to allow usage in production servers. UPGRADVISOR expands on these works, representing changes at the statement level and statically discarding them before using a dynamic tracer to validate them on production servers.

Call graph construction. PyCG [34] builds call graphs for Python code using assignment graphs. It prioritizes analysis speed and completeness and thus exhibits unsoundness in its evaluation. UPGRADVISOR prioritizes soundness, achieved by over-approximating call targets. Various approaches dynamically generate call graphs for JS code [17]. NodeProf [39] instruments the code under test and gather information in the face of code generation and other JS-born challenges. UPGRADVISOR records similar information via tracking jumps and calls online and then decoding this information offline to avoid high overhead. We plan to leverage these works to add JS support for UPGRADVISOR.

Hardware tracing. Modern CPUs provide hardware features for tracing, including Intel PT [21] and ARM embedded trace macrocell (ETM) [26, 37]). These have generally only been applicable to native programs. Our previous work, JPortal [43], showed how to enable hardware tracing for Java bytecode, but it suffers from high overhead and data loss from needing to trace the whole virtual machine. UPGRADVISOR improves on JPortal via novel coarse-grained and selective tracing mechanisms which achieve low overhead without data loss.

Statistical debugging. Statistical debugging [25, 42] reduces tracing overhead through randomized sampling and dispersing data collection among different users. UPGRADVISOR achieves low overhead through selective hardware tracing, which maintains completeness.

Multi variant execution (MVE). MVE methods [18, 28] split test suite execution at the point of change, then run the two versions (before and after upgrade) and merge them back to show compliance. MVE concepts have also been applied towards detecting exploitation attempts and test generation [22, 31]. To overcome lacking coverage in test-suites, UPGRADVISOR traces production servers focusing only on parts relevant to the dependency update.

Patch analysis in continuous integration. SubmitQueue [1] is a system for examining simultaneous application code updates. It combines a build dependency graph with a continuously trained statistical model to optimize the order of application code updates to maximize parallelism for integration tests. In contrast, UPGRADVISOR provides decision support for evaluating dependency updates using production traces.

9 Conclusions and Future Work

We have shown that many projects suffer from prolonged delays in adopting dependency updates. We have designed and built UPGRADVISOR, a system for reducing developer effort and error risk in adopting dependency updates. UPGRADVISOR features the co-design of a sound static analysis constructed to pinpoint a carefully selected target set of methods to trace and a low-overhead production-ready tracer to observe dependency usage. Using this hybrid analysis together with hardware tracing, UPGRADVISOR has analyzed 172 upgrade opportunities, determining that ~60% of them can be updated safely. For the rest, UPGRADVISOR benefits developers by reducing the manual effort of going over the changes in the dependency.

We plan to extend UPGRADVISOR to benefit more dynamic languages. Moreover, we wish to build upon UPGRADVISOR's analysis to alert about malicious updates and generate application tests for increasing dependency update coverage. We believe UPGRADVISOR's low-overhead tracing technique can become useful in other domains and intend to explore its use in debugging and fault isolation.

Acknowledgments

Landon Cox provided helpful comments on earlier drafts. Andrew Magid helped with system implementation. This work was supported in part by DARPA contract N66001-21-C-4018; ONR grants N00014-17-1-2788 and N00014-18-1-2037; NSF grants CNS-1564055, CNS-1703598, CNS-1763172, CNS-1907352, CCF-1918400, CNS-2052947, CNS-2007737, CNS-2006437, CNS-2128653, CNS-2106838, and CCF-2124080; Faculty Research Awards from Facebook, JP Morgan, DiDi, Cisco, and Accenture; and a Columbia CAIT Award. (Corresponding authors: Junfeng Yang and Zhiqiang Zuo)

References

[1] Sundaram Ananthanarayanan, Masoud Saeida Ardekani, Denis Haenikel, Balaji Varadarajan, Simon Soriano, Dhaval Patel, and Ali-Reza Adl-Tabatabai. Keeping master green at scale. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys '19)*, March 2019.

[2] Apache. Cassandra - open source nosql database. https://cassandra.apache.org/_/index.html. Accessed: 2022-05-24.

[3] Google Chrome. Chrome release cycle. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/process/release_cycle.md. Accessed: 2022-05-24.

[4] Django. Django deprecation timeline. https://docs.djangoproject.com/en/dev/internals/deprecation/. Accessed: 2022-05-24.

[5] Django. Django Packages is a directory of reusable apps, sites, tools, and more for your Django projects. https://djangopackages.org. Accessed: 2022-05-24.

[6] Django. How to deploy Django. https://docs.djangoproject.com/en/4.0/howto/deployment/. Accessed: 2022-05-24.

[7] Facebook Engineering. Rapid release at massive scale. https://engineering.fb.com/2017/08/31/web/rapid-release-at-massive-scale/. Accessed: 2022-05-24.

[8] Facebook. Django workload by Instagram and Intel, v1.0 RC. https://github.com/facebookarchive/django-workload. Accessed: 2022-05-24.

[9] Facebook. Pyre: A performant type-checking for Python 3. https://pyre-check.org. Accessed: 2022-05-24.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (ASE '14)*, pages 313–324, September 2014.

[11] Beat Fluri, Michael Wursch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, October 2007.

[12] Python Software Foundation. Cpython. https://github.com/python/cpython. Accessed: 2022-05-24.

[13] Jeffrey Fulmer. Siege 4.1.1 - an http load tester and benchmarking utility. https://github.com/JoeDog/siege. Accessed: 2022-05-24.

[14] Google. Atheris: A coverage-guided, native python fuzzer. https://github.com/google/atheris. Accessed: 2022-05-24.

[15] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Notices*, 32(10):108–124, October 1997.

[16] Masatomo Hashimoto and Akira Mori. Diff/ts: A tool for fine-grained structural change analysis. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE '08)*, pages 279–288, October 2008.

[17] Zoltán Herczeg and Gábor Lóki. Evaluation and comparison of dynamic call graph generators for JavaScript. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE '19)*, pages 472–479, May 2019.

[18] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE '13)*, pages 612–621, May 2013.

[19] Intel. Intel Processor Trace virtualization enabling. https://lwn.net/Articles/737839/. Accessed: 2022-05-24.

[20] Intel. libipt: an Intel Processor Trace decoder library. https://github.com/intel/libipt. Accessed: 2020-10-31.

[21] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*, chapter 35: Intel Processor Trace. June 2019.

[22] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, pages 431–442, June 2016.

[23] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, February 2018.

[24] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, December 2013.

[25] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Notices*, 38(5):141–154, May 2003.

[26] Arm Limited. *Arm® Embedded Trace Macrocell Architecture Specification ETMv4.0 to ETMv4.5*, December 2019.

[27] memcached. memcached - a distributed memory object caching system. https://memcached.org. Accessed: 2022-05-24.

[28] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. Exploring variability-aware execution for testing plugin-based web applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, pages 907–918, May 2014.

[29] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, and Tien N. Nguyen. Operation-based, fine-grained version control model for tree-based representation. In *Proceedings of the 13th Conference on Fundamental Approaches to Software Engineering (FASE '10)*, pages 74–90, March 2010.

[30] npm. How npm works: Dependency hell. https://npm.github.io/how-npm-works-docs/theory-and-design/dependency-hell.html. Accessed: 2022-05-24.

[31] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. Shadow of a doubt: Testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*, pages 1181–1192, May 2016.

[32] Maksym Petrenko and Václav Rajlich. Variable granularity for improving precision of impact analysis. In *Proceedings of the IEEE 17th International Conference on Program Comprehension (ICPC '09)*, pages 10–19, May 2009.

[33] Xiaoxia Ren, B.G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for Java programs. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 664–665, May 2005.

[34] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. PyCG: Practical call graph generation in Python. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE '21)*, pages 1646–1657, May 2021.

[35] Python steering council. Pep 318 – decorators for functions and methods. https://peps.python.org/pep-0318/. Accessed: 2022-05-24.

[36] Python steering council. Pep 484 – type hints. https://peps.python.org/pep-0484/. Accessed: 2022-05-24.

[37] Neal Stollon. *ARM ETM*, pages 213–218. Springer US, Boston, MA, October 2010.

[38] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. Technical lag of dependencies in major package managers. In *Proceedings of the 27th Asia-Pacific Software Engineering Conference (APSEC '20)*, pages 228–237, July 2020.

[39] Haiyang Sun, Daniele Bonetta, Christian Humer, and Walter Binder. Efficient dynamic analysis for node.js. In *Proceedings of the 27th International Conference on Compiler Construction (CC '18)*, pages 196–206, February 2018.

[40] TIDELIFT. libraries.io - the open source discovery service. https://libraries.io. Accessed: 2022-05-24.

[41] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *Proceedings of the 17th International Conference for Software Reuse (ICSR '18)*, pages 95–110, May 2018.

[42] Alice X. Zheng, Michael I. Jordan, Ben Liblit, Mayur Naik, and Alex Aiken. Statistical debugging: Simultaneous identification of multiple bugs. In *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pages 1105–1112, June 2006.

[43] Zhiqiang Zuo, Kai Ji, Yifei Wang, Wei Tao, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. JPortal: Precise and efficient control-flow tracing for JVM programs with Intel Processor Trace. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*, pages 1080–1094, June 2021.

A Artifact Appendix

Abstract

The version of UPGRADVISOR used to perform the experiments described in the paper may be downloaded from figshare.com. The artifact contains the code for the package survey, the static analyzer, and the hardware tracer. It also contains scripts to compile the tracer, run the experiments described in the paper, and produce most of the figures. For the most up to date version of UPGRADVISOR and other resources please refer to may be accessed on Github at http://upgradvisor.github.io.

Requirements

We provide the analyzer pre-installed in a docker container. The tracer requires a bare-metal machine. It directly employs a tracing capability found in Intel 5th generation CPUs (Broadwell) and above. Installing the tracer software requires root access to the OS.

This artifact will run on a i7-10700 CPU workstation with 16GB RAM. A slower machine may result in reduced performance. We set up the docker container on the tracer machine and encourage you to do the same.

Scope

The artifact may be used to reproduce the experiments described in the paper, including Fig. 1, Fig. 4, Fig. 8, Fig. 9, Fig. 10, Table 1, and Table 2.

Contents

- AnalyzerDocker.tar.gz: A docker container for running the survey and static analysis portions of Upgradvisor.
- Cache[2].tar.gz: Cached intermediate results of Upgradvisor to serve as examples and troubleshooting aids.
- UpgradvisorArtifact-main.tar.gz: The code of the Upgradvisor analyzer and tracer.
- README.md: Instructions for setting up and running the Upgradvisor experiments.

We recommend following the README's instructions for running the survey, and static analysis, as well as for checking compatibility with, compiling, and running the hardware tracer.