Impact Analysis of Cross-Project Bugs on Software Ecosystems

Wanwangying Ma
State Key Lab. for Novel Software Technology, Nanjing University
Nanjing, China
wwyma@smail.nju.edu.cn

Lin Chen*
State Key Lab. for Novel Software Technology, Nanjing University
Nanjing, China
lchen@nju.edu.cn

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

Yang Feng
Nanjing University
Nanjing, China

Zhaogui Xu
Nanjing University
Nanjing, China

Zhifei Chen
Nanjing University
Nanjing, China

Yuming Zhou
State Key Lab. for Novel Software Technology, Nanjing University
Nanjing, China
zhouyuming@nju.edu.cn

Baowen Xu
State Key Lab. for Novel Software Technology, Nanjing University
Nanjing, China
bwxu@nju.edu.cn

ABSTRACT

Software projects are increasingly forming social-technical ecosystems within which individual projects rely on the infrastructures or functional components provided by other projects, leading to complex inter-dependencies. Through inter-project dependencies, a bug in an upstream project may have profound impact on a large number of downstream projects, resulting in cross-project bugs. This emerging type of bugs has brought new challenges in bug fixing due to their unclear influence on downstream projects. In this paper, we present an approach to estimating the impact of a cross-project bug within its ecosystem by identifying the affected downstream modules (classes/methods). Note that a downstream project that uses a buggy upstream function may not be affected as the usage does not satisfy the failure inducing preconditions. For a reported bug with the known root cause function and failure inducing preconditions, we first collect the candidate downstream modules that call the upstream function through an ecosystem-wide dependence analysis. Then, the paths to the call sites of the buggy upstream function are encoded as symbolic constraints. Solving the constraints, together with the failure inducing preconditions, identifies the affected downstream modules. Our evaluation of 31 existing upstream bugs on the scientific Python ecosystem containing 121 versions of 22 popular projects (with a total of 16 millions LOC) shows that the approach is highly effective: from the 25490 candidate downstream modules that invoke the buggy upstream functions, it identifies 1132 modules where the upstream bugs can be triggered, pruning 95.6% of the candidates. The technique has no false negatives and an average false positive rate of 7.9%. Only 49 downstream modules (out of the 1132 we found) were reported before to be affected.

*Corresponding author

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380442

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; **Maintaining software**; *Open source model*.

KEYWORDS

Software Ecosystems, Cross-project Bugs, Bug Impact, Dependence Analysis, Symbolic Constraints

ACM Reference Format:
Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yang Feng, Zhaogui Xu, Zhifei Chen, Yuming Zhou, and Baowen Xu. 2020. Impact Analysis of Cross-Project Bugs on Software Ecosystems. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380442

1 INTRODUCTION

Recent years have seen a trend that software projects are forming large scale social-technical ecosystems in which projects depend on the infrastructures or functional components provided by other projects [17]. Projects within an ecosystem often have complex inter-dependencies that impose new challenges in software maintenance [9, 27]. As an important indicator of software quality, bugs have long been a focus of study in the field of software engineering and are undoubtedly a more significant concern in ecosystems as bugs found in a project are very likely to affect many other projects in the ecosystem through inter-dependencies [7, 21].

For example, *Scipy* is an upstream library in the scientific Python ecosystem and has a significant number of downstream projects depending on it, such as *Scikit-image* that is a collection of algorithms for image processing and *Nilearn*, which is a Python module for fast and easy statistical learning on neuroimaging data. A bug was found in *Scipy*'s function `scipy.ndimage.interpolation.affine_transform()`(reported in the project *Scipy* with issue id scipy/scipy#1547) and confirmed to affect *Nilearn*. This kind of bug which has impact

on not only the reported project but also its downstream project(s) is called *cross-project bug*. Cross-project bugs are not uncommon in practice. Ma et al. identified hundreds of instances from the scientific Python ecosystem [21]. After inspecting the fixing process of cross-project bugs and conducting an online survey, they reported that compared with intra-project bugs, such bugs have much more severe impact and are extremely more difficult to deal with. Unlike single-project development, different projects within a software ecosystem are developed and maintained separately and asynchronously, thus fixing of a cross-project bug locally is usually not the end of its impact on other projects. Not until a patched version of the upstream project is released will the downstream projects get rid of the bug [9]. Then if the published fix is not satisfactory, the affected projects have to wait for another release cycle to get a new fix, which enlarges the bug impact and requires extra efforts. Therefore, the upstream developers are very cautious when facing a cross-project bug, and they are willing to seek advice from their dependent projects. But even so, the proposed bug-fixes are sometimes unsatisfactory.

Take the aforementioned bug (scipy/scipy#1547) as an example. The function `scipy.ndimage.interpolation.affine_transform()` applies an affine transformation on a given array. It will produce wrong results when the parameter `matrix` is a diagonal-matrix and the parameter `offset` is not (0, 0). Fixing the buggy method requires changing the output interface and hence might break some downstream projects. *Scipy*'s developers were very cautious about how to repair the bug. They asked *Scikit-image*'s developers for feedback because they thought that *Scikit-image* would have called the method frequently. After a lengthy inspection of their code, *Scikit-image*'s developers reported that there is no use of `affine_transform()` that satisfies the failure triggering conditions and hence the bug may not concern their project. Still, *Scikit-image*'s developers provided several suggestions and the bug was eventually fixed after lengthy debate. Unfortunately, six months later, a developer of *Nilearn* reported that they were unhappy about the fix since it broke their code without any warning and they spent much effort in debugging the problem tracing back to the problematic fix. From the discussion, *Scipy*'s developers were completely unaware of the fact that *Nilearn* was using the functionality until then.

The real-world example discloses that it is difficult to provide an appropriate fix if the upstream and downstream sides are unclear about the possibly affected downstream code before starting to design a fix. In other words, in order to repair a cross-project bug effectively and efficiently, it is important to determine its impact. However, it is challenging in software ecosystems with numerous projects and project versions. For the upstream side where a bug occurs, the developers may have no idea which downstream projects are using the buggy function, making it very difficult to solicit bug fix suggestions from the downstream developers. For the downstream side, as a *Scikit-image*'s developer pointed out, they themselves are not sure whether and how much an upstream bug affects their projects unless they comb through the code and figure out how they call the upstream function everywhere they use it. It is time-consuming and error-prone. Thus, some truly affected projects like *Nilearn* may not realize the impact until the projects fail by the bug or its fix, which, however, is too late for the downstream developers to give advice or take part in the fixing.

It is not hard to see that analyzing cross-project bug impact requires lots of painful manual efforts. Therefore to alleviate the problem, we present an ecosystem-wide bug impact analysis to identify the affected downstream modules (classes/methods) of a given upstream bug. *Note that an upstream function containing the bug is invoked in a downstream module does not mean the bug in the function must be triggered as the failure inducing preconditions may not be present in the downstream project.* Hence, our goal is to find out all the *truly impacted* downstream uses from the huge number of modules within an ecosystem, so that the proposed upstream fix is more likely to satisfy all its affected downstream developers. Specifically, *a downstream module is truly impacted if the failure inducing preconditions of the upstream bug can be satisfied in the downstream module such that the buggy logic can be triggered, leading to corrupted states.* For a reported bug with the known root cause function and failure inducing preconditions, we first collect the candidate downstream modules based on an ecosystem-wide dependence analysis. Then, a conservative path-sensitive intra-module impact analysis is performed by encoding the paths to the call sites of the buggy upstream function, together with the preconditions, as symbolic constraints. Solving the constraints discloses if the downstream module can be affected.

We evaluate the approach on the scientific Python ecosystem. We have collected 31 cross-project bugs from projects *Numpy* and *Scipy* in this ecosystem that have at least one confirmed affected downstream project. We then analyze 22 popular projects in 121 different versions in the ecosystem, with a total of 16 millions LOC to identify the impact of these bugs. From 25490 modules that are using the buggy methods, our technique identifies totally 1132 modules that are affected by these bugs, saving 95.6% inspection efforts for the developers (to check the bug impact). Our analysis is conservative in the sense that if it concludes a module is not affected, the module must not be affected. Although it may have false positives due to the conservative symbolic encoding, our experiments show that the average false positive rate is only 7.9%. Among the 1132 modules reported by our tool, only 47 were reported before. Our technique is also efficient. The mean time for estimating every module is only 0.860 seconds.

The ecosystem-wide bug impact analysis is useful for both the upstream and downstream developers in helping them fix cross-project bugs effectively and efficiently, as well as minimizing the bug impact on the entire ecosystem. For the buggy upstream project, once a bug occurs, our approach can tell the developers which downstream projects and how much they are likely to be influenced by a given bug so that they can communicate with the affected projects, especially the most important ones, to understand the downstream requirements, determine the priority of the bug, and decide the ultimate solution. For a downstream project, our approach can tell the developers whether it might be influenced by a given bug and point out all the possibly affected modules (methods or classes), preventing to report duplicated bugs [26]. The approach saves the developers' effort by directing their inspection towards these modules. After inspecting the affected code, the developers can even choose to work around the bug, without waiting for the upstream fix. They can also learn the possible workarounds from the sibling projects affected by the same bug [14].

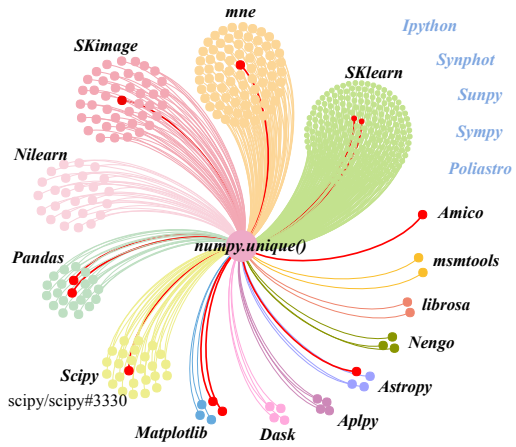


Figure 1: The impact of a bug in `numpy.unique()`. The central point denotes `numpy.unique()`, the surrounding dots of different colors mean the modules of different projects, and the lines indicate the calls from these modules to `numpy.unique()`. The five projects *Ipython*, *Synphot*, *Sunpy*, *Sympy*, and *Poliastro* have no use of `numpy.unique()`. The red dots are the modules potentially affected by numpy/numpy#2655.

2 MOTIVATING EXAMPLE

In this section, we use a real case in the scientific Python ecosystem to illustrate the problem and motivate the design of our approach.

Numpy is a fundamental library in the scientific Python ecosystem, with more than 200,000 projects relying on it. Fig. 1 shows a small part of the dependencies including 19 projects such as *Sunpy* and *Synphot*. The function `numpy.unique()` was reported to produce wrong results when the first parameter `ar` is an array with more than 16 items and the parameter `return_index` is set to be `True` (reported in numpy/numpy#2655). To analyze the impact of this bug on the 19 projects, we first have to identify which of them are using `numpy.unique()`. A direct way is to analyze the call relationships between the downstream projects and the buggy *Numpy* function. By inspecting the code, 14 of the 19 projects in Fig. 1, with totally 453 modules (functions or classes), are found to call `numpy.unique()` in various ways. The central point in Fig. 1 denotes `numpy.unique()`, the surrounding dots of different colors mean the modules of different projects, and the lines indicate the calls from these modules to `numpy.unique()`.

However, not all these modules are affected by the upstream bug, depending on the bug inducing preconditions. For example in *Librosa*, though the function `librosa.core.fmt()` calls `numpy.unique()`, it uses the default value 0 of parameter `return_index`. As such, the bug has no effect on this module. Thus, picking out the affected ones requires an analysis on how the downstream modules use the erroneous upstream function, or more specifically, which downstream uses satisfy the bug inducing preconditions. After reviewing the code, 11 modules from the projects *Mne*, *Skimage*, *Nilearn*, *Pandas*, *Scipy*, *Matplotlib*, *Astropy*, and *Amico* call `numpy.unique()` in the bug-triggering way, i.e., with `len(ar)> 16 and return_index == True` , and thus are affected by numpy/numpy#2655 (shown as red dots in Fig. 1). It is also worth noting that *Scipy*'s developers have reported the impact in scipy/scipy#3330.

The motivating example illustrates two key problems which we have to tackle when analyzing the ecosystem-wide impact of a cross-project bug with known erroneous method and bug inducing preconditions. First, which dependent modules are using the buggy upstream method? Second, which downstream uses may satisfy the bug inducing preconditions?

For the first problem, it is natural to leverage a cross-project call graph to solve it. However, the dynamics and complexity of software ecosystems give rise to many new challenges. As an open environment, new projects may join in an ecosystem at any time by simply importing and using a library within the ecosystem. Therefore, the cross-project call graph should be extended flexibly to include new projects in order to gain a full-scale impact analysis. At the same time, the extension should only induce local changes, without interfering with most of the existing graph structure so that there is no need to reconstruct the ecosystem-wide call graph, which incurs high overhead especially for large ecosystems.

Moreover, the graph shall be version aware, modeling multiple versions of a project as long as they are still used by some downstream projects. Specifically, a buggy upstream function may only be used by some specific versions of a downstream project. For example, the motivating example numpy/numpy#2655 which came up in *Numpy* <1.8.0 does not affect *Scipy* 1.2.0 which requires *Numpy* >=1.13.3.

Taking into account the requirements of scalability, flexibility, and version-sensitivity, we leverage a dependency analysis to extract cross-project call relationships between the upstream functions and the downstream modules (functions or classes). It can precisely identify the particular versions of a downstream project that is affected by an upstream function, and the versions of the upstream project that a downstream project uses. After the analysis, the candidate modules which are using concerned upstream buggy functions can be identified. Note that although our technique can traverse dependences within the whole ecosystem, the identification of dependences is performed in a modular fashion. That is, our algorithm constructs cross-project dependences for each project separately. Such a design is critical to handling the dynamic evolution of an ecosystem. More details can be found in Section 3.2.

For the second problem, processing of individual candidate modules (in downstream projects) is needed to determine whether a module invokes the buggy function in the failure inducing fashion. Considering the number and diversity of candidate modules, several requirements arise when proposing a method for selecting the bug-triggering downstream uses.

Efficiency. Within a software ecosystem, the number of candidate downstream modules identified from the cross-project call graph may be large, especially for a popular upstream function (e.g., 453 modules in the aforementioned case). Since cross-project bug impact analysis is supposed to be used before designing the bug fixes, the analysis on individual modules should be completed in a short time in order not to hold up the fixing process.

Complete Coverage. The analysis is supposed not to miss any affected downstream module so that the upstream fix is more likely to satisfy all their downstream users. Therefore, for each module,

the designed method should examine every possibility of input for invoking the buggy function in order to decide whether it is affected or not. Therefore, simply running the downstream tests is insufficient since downstream projects may not have test cases to expose upstream bugs even though they may have consequences.

Considering the above requirements, we develop a conservative path-sensitive intra-module symbolic analysis to determine whether the values of input parameters of an upstream function invocation meet the failure inducing preconditions. Specifically, we analyze the candidate (downstream) modules and encode paths to the call sites of the upstream function in symbolic constraints. By asserting the conjunction of these constraints and the failure inducing preconditions, we can tell whether the downstream modules are affected.

Our technique is not a symbolic execution engine which explores individual paths and encodes one single path at a time. Instead, our technique encodes all the paths reaching the upstream function call sites at once, without any path exploration. Furthermore, symbolic execution usually can achieve precise encoding as it encodes concrete paths, in which all the dynamic features are unfolded (e.g., dynamic types of variables are known), whereas our encoding is conservative due to the lack of concrete information. In the next section, we show the design of our ecosystem-wide bug impact analysis in details.

3 METHOD DESIGN

3.1 Overview

Fig. 2 presents the overview of our approach, which consists of two main phases: 1) the ecosystem-wide dependence analysis to select downstream module candidates that call the upstream buggy methods and 2) the intra-module impact analysis to identify the affected modules that may meet the bug inducing preconditions.

The goal of the ecosystem dependence analysis is to construct a version-sensitive dependence network over the target ecosystem, which is accomplished by the dependence analyzer and the version handler. Within the target ecosystem, each project is processed independently to better handle the dynamic evolution of the ecosystem and to achieve scalability. First, for each project, with a base version of the project as the input, the dependence analyzer leverages fine-grained function invocation relations to construct a base dependence network. Then, the version handler processes successive versions of the project incrementally by comparing function call changes to add version specific information to the network. Note that while the network is stored as a central database, its construction and update are modular and distributed to individual projects. Hence, when a project is updated, only the dependences related to the project may be updated. Such a design substantially reduces the maintenance overhead. As shown in Section 4.3.2, the size of the network for part of the scientific Python ecosystem is only 248.4MB and hence very manageable. Given an upstream buggy function, the module selector can easily identify the candidate downstream modules using the dependence network.

The intra-module impact analysis takes three resources as inputs, i.e., individual candidate modules, a buggy upstream function, and the failure inducing preconditions extracted from bug reports. It analyzes each candidate and identifies whether the candidate is truly affected. We say a module is truly affected if with some legitimate inputs to the module, the bug in the upstream function can be triggered. Specifically, the intra-module analysis consists of three subcomponents: the code preprocessor, the constraint encoder, and the SMT solver. The code preprocessor reduces code with rich syntax to a simple canonical form. The constraint encoder symbolically encodes paths to call sites of the upstream function as well as the failure inducing conditions. The constraints are then passed to the SMT solver. If it is satisfiable, the analyzed module is considered affected.

3.2 Ecosystem-wide Dependence Analysis

The dependence analysis processes each project in the ecosystem seperately. Consider a software eocsystem *SE* which consists of n projects. Being a member of the ecosystem, a project P_i ($0 \leq i \leq n$) in *SE* plays two roles. As a downstream project, P_i uses the functionalities provided by another project P_j ($0 \leq j \leq n \wedge j \neq i$), and the cross-project call relations form a directed inter-project dependence graph $G(P_i) =< V_{from}, V_{to}, E >$, where V_{from} is a set of source nodes representing modules (classes or functions) in P_i, V_{to} is a set of target nodes representing functions defined in project P_j and called by P_i, and $E \subseteq V_{from} \times V_{to}$ is a set of directed edges representing the call relations. In the remainder of the section, we use $m \in V_{from}$ to denote a downstream module and $f \in V_{to}$ to denote an upstream function. On the other hand, as an upstream project, the functions defined in P_i can be used by P_j and thus serve as the target nodes in $G(P_j)$. Therefore, for any project in *SE*, the dependence analysis cares about its function definitions and inter-project call relations.

In order to identify which versions (of a downstream project) are affected by a cross-project bug in a specific upstream version, we analyze multiple releases of each project. Assume P_i has k versions, i.e., V_{i0}, V_{i1}, ..., and V_{ik}. We first choose a version (such as the oldest version V_{i0}) as the base to extract the inter-project call dependencies and construct a base graph. To represent version information, an attribute tf is maintained for each function f, which is defined as a tuple to indicate the first and last versions where P_i has f. Meanwhile, a hash table c is constructed for each (cross-project) call edge $e = m \rightarrow f \in E$. The table maps a specific version of m to a range of versions of f, indicating the version range of f on which m in that version depends. For example in Fig. 3, the hash table on the edge denotes that m in the V_{i0} version of P_i calls f in versions V_{j1} to V_{j5} of P_j; m in the V_{i1} version of P_i calls f in versions V_{j2} to V_{j5} of P_j; and so on. Note that $f.tf[1]$ is V_{j5}.

After all the n projects in the ecosystem *SE* are analyzed, the generated individual graphs $G(P_1)$, $G(P_2)$, ..., $G(P_n)$ are combined to form the ecosystem-wide dependence network *G(SE)* by merging the same nodes, i.e, $G(SE) =< G(P_1), G(P_2), \ldots, G(P_n) >$, where $P_1, P_2, ...,$ *and* $P_n \in SE$. Next, we introduce the base graph construction and the version analysis.

3.2.1 *Base Graph Construction.*

The base graph of a project is built by the dependence analyzer, which consists of three subcomponents: the AST parser, the filter, and the dependence graph storage. It processes the base version of every project independently.

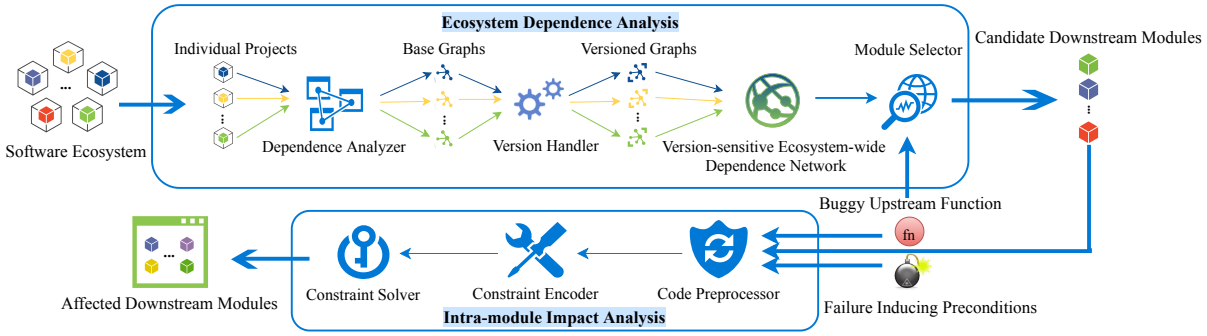


Figure 2: The workflow of our approach

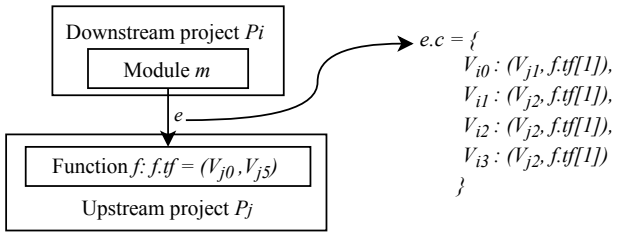
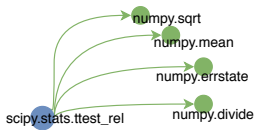


Figure 3: Versioning the upstream and downsteam projects

```
165 import numpy as np
3318 def ttest_rel(…):
       …
3281     dm = np.mean(d, axis)
3282     denom = np.sqrt(v / n)
3383     with np.errstate(…):
3384     t = np.divide(dm, denom)
3385     t, prob = _ttest_finish(df, t)
```

(a) Code snippet



(b) Corresponding call dependencies

Figure 4: Inter-project call graph for `scipy.stats.ttest_rel()`

Given the base version V_{i0} of project P_i, the AST parser first parses the source code into Abstract Syntax Trees (ASTs). The function call entities are further analyzed to extract the callers and callees. Since we focus on the cross-project impact of erroneous upstream functions, we only concern the inter-project call relations, that is, only the callees defined in other projects need to be retained. To do this, we employ a filter to preclude the callees defined within the project based on the import commands, the class definitions, and the function definitions. Fig. 4 illustrates a simple example of extracting the dependency relation. As shown in Fig. 4a, the function `ttest_rel()` in the project *Scipy* calls `np.mean()`, `np.sqrt()`, `np.errstate()`, `np.divide()`, and `_ttest_finish()` at line 3181 to line 3185. Since `__ttest_finish()` is defined within *Scipy*, it is not of interest in this study and thus is precluded from the following analysis. Fig. 4b shows the corresponding inter-project call dependencies.

Meanwhile, the function definition entities are also recorded since they are likely to be the target nodes of the graphs for other projects.

3.2.2 *Version Analysis.*

When constructing the base graph for project P_i of the V_{i0} version, the value of tf (i.e., the version range in which a function is available in P_i) and the entries of c (i.e., hash table on edge $e = m \rightarrow f$) are initialized as (V_{i0}, V_{i0}) and $\{V_{i0}\}$, respectively. To obtain the corresponding value of c, we first try to acquire the information from the configuration files (such as the setup.py in Python projects) in P_i, which may indicate the lower and upper bounds of depending versions of the upstream projects. However, such information may be incomplete due to the lack of configuration files or not specifying versions in these files. In such cases, the lower/upper bound is the first/latest version of the upstream project P_j which contains f with the exact same interface. For the example in Fig. 3, $e.c$ is initialzied as $\{V_{i0}: (V_{j1}, f.tf[1])\}$, where V_{j1} is obtained from the configuration file and $f.tf[1]$ is the lastest version of P_j containing f.

After the base graph is built, the version handler updates the values of tf and c by comparing two subsequent versions of project of P_i incrementally. The version handler consists of two subcomponents: the code comparator to identify the differences between two versions, and the version updater to add version information to base graphs.

From version V_{i1} to the latest version V_{ik}, the code comparator compares the ASTs between V_{ip} and $V_{i(p-1)}$ $(1 \leq p \leq k)$. It records four kinds of changes: function call deletion, function call insertion, function definition deletion, and function definition insertion. These changes are sent to the version updater to update the values of tf and c.

Updating *tf*. When a new function is defined in the V_{ip} version, its tf is initialized as (V_{ip}, V_{ip}). If an existing function is deleted from V_{ip}, its tf remains unchanged. For other functions in V_{ip}, the second item of their tf is updated from $V_{i(p-1)}$ to V_{ip}.

Updating *c*. In the version V_{ip} of project P_i, if its module m is modified so that it no longer calls the upstream function f, the hash table on edge e from m to f does not need to be updated. Otherwise, an entry is added to c to denote the invocation in version V_{ip}. In Fig. 3, since version V_4 the call from m to f is deleted, c_i has four entries, denoting versions V_{i0}, V_{i1}, V_{i2}, and V_{i3}. The values for new entries are obtained from the configuration files or $f.tf$ as described above.

After all the versions are processed, the values of tf and c are obtained, and the version-sensitive dependence network is constructed once every interested project is analyzed. With the network, we can tell in which versions a downstream module is invoking an upstream buggy function of a specific version.

The dependence network serves as a database for the whole analysis. The incremental processing of code indicates that the

dependence network can be easily updated with code changes of a project. Anytime a developer commits some change, the network can be easily modified accordingly without interfering with other projects that do not have direct dependences. Moreover, with the version information, the network can show the profile of inter-project dependency relationship of a software ecosystem at any interesting point of time. Last, the network features the capabilities of dealing with ecosystem dynamics (i.e., projects join and leave).

After the ecosystem-wide dependence network is constructed, for a given cross-project bug with known root cause buggy function, the candidate selector identifies all the downstream modules which are using the buggy upstream function. Then the candidates are sent to the intra-module impact analyzer one by one to check whether they are truly affected.

3.3 Intra-module Impact Analysis

The intra-module impact analysis symbolically encodes each module to assess whether the given upstream buggy function can be invoked with the failure inducing preconditions. Specifically, for each concerned downstream module, the analyzer first utilizes the code preprocessor to normalize the module code, and then it replaces the body of the buggy function with a simple check of the failure inducing conditions. Then, the constraint encoder encodes the possible paths from the module entry to the call sites of the buggy upstream function. Last, all the constraints including the encoded failure inducing conditions are sent to the constraint solver. A module is affected by the bug if the result obtained from the solver is satisfiable.

3.3.1 *Code Preprocessing.*

a) Code normalization.
In a source code file, one source code line may contain multiple statements and a long statement may cross multiple lines. For the simplicity of the subsequent analysis, we first normalize the downstream module so that each line only contains a simple operation. While such normalization is well supported in languages such as C/C++ and Java (e.g., through compiler IR), it is lacking for our target language Python. As such, we develop our own normalization methods. The following two kinds of normalization are currently performed.

Linearizing nested expressions. Nested expressions combine multiple operations together (e.g., `foo(a) + b`). We linearize these nested expressions as a set of simple assignments, each containing a simple expression.

Simplifying complex constructs. Python provides expressive syntax to represent complex semantics concisely. Despite of its convenience for developers, it brings difficulties for analysis. The code preprocessor transforms several kinds of advanced constructs to a small set of basic operations. Fig. 5 illustrates how we process "list comprehension" in Python as an example. The statement "`x = [i+1 for i in range(5) if i%2==0]`" shows a concise method to construct a list, which is transformed into several basic statements, such as a for statement that contains an if statement.

b) Code integration.
After normalizing the downstream code, the code preprocessor abstracts the buggy upstream function to a simple check of the

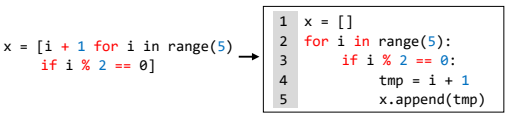


Figure 5: Simplifying list comprehension

failure inducing conditions, which is sufficient for our purpose. The procedure is shown in Fig. 6 and explained as follows.

First, the code preprocessor only retains the interface of the upstream function `upfunc()`, and replaces its body with an if-else statement. The failure inducing preconditions obtained from the bug report act as the conditions of the if-else statement. Then, the code preprocessor retains the body of the downstream module `downfunc()` from the first line to the call sites of the buggy upstream function. It modifies the statement at the downstream call site by replacing the original return target variable as `tmpResult` (or adding the target variable if there is not one). An example of the resulting code is shown in Fig. 6c. It is then sent to the symbolic encoding component.

3.3.2 *Symbolic Analysis.*

a) Input variables initializing.
For dynamic programming languages like Python, the static type information of variables is not explicitly indicated. A specific input variable of a module can be of different data types. For example, the project *Numpy* specifies that the input parameter `axis` of its function `numpy.nanpercentile()` can be an integer, a list of integers, or `None`. This poses challenges to symbolic analysis, in which symbolic variables need to be explicitly typed. To handle such situation, we use multiple symbolic variables of different types to denote the value of an input variable x. For a statement s involving x, all the symbolic variables representing x are updated to encode the possible behavior of s along with the different types of x.

b) Constraint encoding.
The constraint encoder symbolically encodes possible paths from the module entry to the call sites. During encoding, it transforms each path into its single static assignment (SSA) form so that every variable is defined exactly once. Next we describe how we encode several typical constructs.

Assignments. For a constant assignment $x = v$ (v represents a typical object literal including the number, string, list, etc.), the encoder transforms it into the SSA form and encodes it to constraints. For a simple assignment $x = y$, the encoder looks for the most recent definition of y. If it is found, the encoder first resolves the definition and then encodes it after transformation to the SSA form. Otherwise, we initialize y with multiple symbolic variables of different types. For binary operation such as $x = y + z$, the encoder processes it similarly.

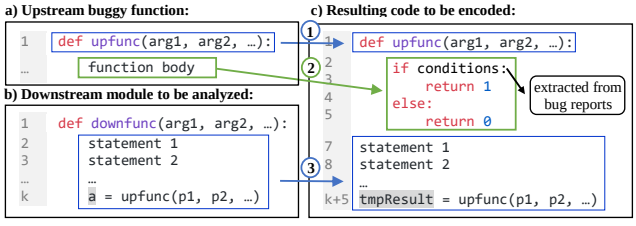


Figure 6: The procedure of code integration

Calls. To ensure the efficiency of impact analysis, the encoding is performed within the target module. As the module may call other functions in addition to the buggy upstream function, we manually provide symbolic models for a set of commonly used library functions. Note that this is normal in symbolic analysis. We resolve other calls to un-modeled external functions in a conservative way. We assume these functions can change the values of their input parameters and return any value of any type. Therefore, for a statement `x = f(p1, ..., pn)`, the return target variable `x` and input variables `pi` ($1 \leq i \leq n$) are reinitialized with multiple symbolic variables of different types. In this way, our analysis is conservative, meaning that if a buggy upstream function can affect a downstream module, our tool must report it. We consider this to be more desirable than a typical aggressive no-false-positive strategy in symbolic bug finding as identifying the potentially affected downstream modules is critical for upstream bug fixing. In Section 4.3, our results show that even though our analysis is conservative, it allows pruning 95.6% of the downstream modules that invoke the buggy functions.

Conditionals. For a branch *if conditions* S_1 *else* S_2, we first resolve the predicate and then aggressively process each statement of S_1 and S_2. During encoding, paths are classified into two categories: *dead-end* and *active*. A dead-end path goes into a direction that can not reach the call sites of the buggy upstream function and thus is discarded during encoding. An active path can lead to the concerned point. We only encode the active paths.

Loops. We unroll all the constant loops to their bounds. For a loop with variable bound (e.g., while loops), we unroll it 10 times. To be conservative, in the else branch of the last round of unrolling, we re-initialize all the variables.

Example. Consider the example in Fig. 7 where `uf()` is the buggy upstream function. The constraint encoder aims to encode the path to the call sites of `uf()` (at line 7) within the module. It first transforms each statement into its SSA form before encoding. At Line 2, since the encoder does not find a definition of `k` (`k` is the input parameter of the module), it initializes `k` with multiple symbolic variables of different types, such as integer and string. Due to the unsupported addition operation between an integer and a string, the symbolic variable of the string type is discarded from the following analysis. At line 3, an external function `ef()` is called. The encoder encodes `m` and `b` with variables of multiple types like `k`. For the if statement at line 4, the predicate is resolved and two paths are generated with the constraints `m > k` and `m <= k` separately, with the former path not containing line 7. Therefore, lines 1, 2, 3, 6, and 7 form the only active path to `uf()` within the module and only the constraints along this path, together with the failure inducing preconditions, are then sent to the SMT solver.

c) Condition types.
During symbolic execution, the failure inducing preconditions are also encoded. Through inspecting a number of cross-project bugs in the scientific Python ecosystem, we summarize three kinds of the most common conditions under which upstream buggy functions will exhibit unexpected behaviors.

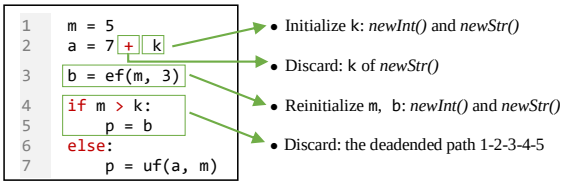


Figure 7: An example for encoding

Type condition. A bug will occur when the type of a certain input parameter for the upstream function falls out of an acceptable set. For example, the function `numpy.fix()` can not correctly process a scalar, which means that when the input parameter `x` is an integer or a float, the bug will be triggered (numpy/numpy#8993).

Value condition. A bug will occur when the value of a certain input parameter for the upstream function falls out of an acceptable range. For example, the *Numpy*'s function `numpy.percentile()` breaks when the input parameter `interpolation` is set to `'midpoint'` (numpy/numpy#7163).

Property condition. A bug will occur when some property of a certain input parameter for the upstream function falls out of an acceptable range. For example, the function `numpy.unique()` in *Numpy* returns wrong results when it processes an array with more than 16 items (numpy/numpy#2655).

We currently support the three kinds of failure inducing preconditions and their combinations. For such bugs, our proposed method reports no false negatives due to its conservative nature. We manually extract the conditions from bug reports and send them in canonical forms to the code preprocessor before encoding. Then all the constraints collected along the active paths including the encoded conditions are passed to the solver. If the solver reports SAT and the value of the variable `tmpResult` is one, we decide that the subject downstream module is possibly impacted by the given upstream bug. The impact-triggering input can also be obtained.

4 EVALUATION

We implemented our approach in a prototype tool in Python. Since Python programs are likely to use a number of external functions and classes implemented in other languages, we manually modeled some commonly used external functions by rewriting them in Python. We used Z3[8] as the SMT solver. We conduct an experiment on the GitHub scientific Python ecosystem to evaluate our approach.

4.1 Research Questions

We attempt to address the following research questions:

RQ1: How effective is our approach in finding the affected downstream modules? For this question, we examine what percentage of the downstream modules using an upstream buggy method is identified to be affected by a confirmed cross-project bug. We also check the false positives of the identified modules.

RQ2: How efficient is our approach? To answer this research question, we monitor the time used to extract the inter-project call dependencies, and the time to analyze the intra-module impact.

4.2 Dataset

In order to answer the above research questions, we evaluate our approach on a set of cross-project bugs which were confirmed to affect some downstream modules. These bugs were collected manually by three steps. First, we focused on two fundamental libraries

Figure 8: An automatic hint indicating that the issue pandas-dev/pandas#12917 reported to the project *Pandas* is related with the *Numpy* bug numpy/numpy#7163.

in the scientific Python ecosystem, i.e., *Numpy* [10] and Scipy[11]. They are the core projects used by nearly all the other projects within the ecosystem, so the bugs occurring these projects are very likely to affect other projects. Among the closed bugs reported for the two libraries, we selected all those which had at least one explicit link with issues reported for other projects. The link may be shown in the comments of the bug report, such as *"Bug found by testing astropy – see astropy/astropy#3848"* in numpy/numpy#5962's comments, or an automatic hint in the bug page (shown in Fig. 8). Second, for each selected bug, we examined whether its linked downstream issues were caused by the bug through reading the bug reports and the comments. If so, the bug was confirmed as a cross-project bug, and the linked downstream projects were considered affected. Third, for each cross-project bug, we recorded its buggy function, failure inducing preconditions, and the affected downstream modules. Excluding the bugs which we could not identify all of the three kinds of information, we collected 31 cross-project bugs in total, involving 22 downstream projects belonging to the scientific Python ecosystem.

We extract the call dependencies from a total of 121 version of the 22 projects to construct an inter-project dependence network. The dependence network is stored in *MySql* and the candidate downstream modules are selected through query statements. We select the 22 projects because the 31 cross-project bugs that we manually studied were confirmed to affect them. Therefore, they can be used as a baseline to validate whether our approach can identify the truly affected downstream modules. Note that our evaluation only involves 22 projects because we have to manually check the results produced by our tool, which costs much effort and limits the number of evaluated projects.

4.3 Results

4.3.1 *Effectiveness (RQ1).*

Table 1 shows our experimental results within the sub-ecosystem. The column *Bug* indicates the bug id in the GitHub issue tracker. The prefixes *N* and *S* mean the bug is from *Numpy* and *Scipy*, respectively. The second and third columns show the number of candidates which are using the buggy upstream functions and the number of affected downstream modules identified by our tool from other 21 projects. The column *%affs* shows the percentage of the affected downstream modules over all candidates.

For the 31 cross-project bugs, our approach first filters downstream modules that do not call buggy upstream functions through an ecosystem-wide call graph, leaving 25490 candidates for intra-module symbolic analysis. Then, the number of affected modules identified by our technique ranges from 3 to 262, with a total number of 1132 and an average number of 36.5. The percentage of impacted modules ranges from 1.08% to 100%, with an average of 33.5%. For 18 bugs, the affected modules are less than 30.0% of all the modules using the buggy upstream methods (highlighted in

Table 1: Statistics of Experimental Results

Bug	#cans	#affs	%affs	#FPs	P	#constraints		time(s)	
						max.	avg.	avg.	total
N#2056	15	6	40.0%	0	100%	22	17.0	0.428	22.4
N#2655	1357	37	2.73%	5	83.3%	464	13.5	0.385	535
N#3484	90	5	5.56%	0	100%	120	26.7	0.948	94.3
N#4225	30	13	43.3%	5	61.5%	14	6.22	0.328	24.8
N#5251	15	6	40.0%	0	100%	30	12.8	2.37	45.6
N#5300	151	26	17.2%	3	88.5%	184	23.8	0.353	68.3
N#5655	330	36	10.9%	0	100%	136	8.42	0.817	288
N#5672	10	3	30.0%	0	100%	49	29.0	0.906	34.1
N#5766	13	3	23.1%	0	100%	14	8.25	0.622	18.1
N#6238	9803	220	2.24%	4	86.7%	702	7.37	0.992	9734
N#6590	159	14	8.81%	0	100%	1134	44.2	0.610	107
N#7163	122	15	12.3%	0	100%	1282	43.6	0.677	93.6
N#8220	3	3	100%	0	100%	19	19.0	0.391	11.2
N#8340	19	8	42.1%	0	100%	73	22.7	0.501	22.6
N#8409	17	5	29.4%	0	100%	68	20.3	0.482	17.8
N#8516	1710	262	15.3%	5	83.3%	5664	26.7	0.985	1654
N#8600	24	9	37.5%	2	77.8%	102	32.8	0.969	35.6
N#8974	76	21	27.6%	5	76.2%	32	4.50	0.124	23.7
N#8993	19	14	73.7%	0	100%	244	60.0	1.73	42.9
N#9560	111	21	18.9%	3	85.7%	864	37.8	0.758	94.1
N#9874	183	39	21.3%	4	86.7%	156	16.8	0.434	99.3
N#10899	145	20	13.8%	5	75.0%	16	4.21	0.146	34.5
N#11103	9018	97	1.08%	3	90.0%	699	7.04	0.911	8245
N#11407	1893	198	10.5%	7	76.7%	50	9.96	0.378	736
N#11426	92	4	4.35%	0	100%	85	18.3	0.616	67.9
S#7278	3	3	100%	0	100%	6	6.00	0.312	10.9
S#7991	10	6	60.0%	0	100%	864	296	4.72	56.7
S#8118	19	9	47.4%	1	88.9%	54	13.0	0.323	19.3
S#8142	11	6	54.5%	0	100%	12	5.67	0.193	12.5
S#9103	7	7	100%	0	100%	9	4.50	0.133	10.9
S#9591	37	17	45.9%	1	94.1%	616	54.0	1.99	81.8
avg.	822	36.5	33.5%	1.71	92.1%	445	10.1	0.860	727
total	25490	1132	4.4%	53	88.7%	–		–	

The column *Bug* indicates the bug id in the GitHub issue tracker. The prefixes *N* and *S* mean the bug is from *Numpy* and *Scipy*, respectively. The second and third columns show the number of candidates which are using the buggy upstream functions and the number of affected downstream modules. The column *%affs* shows the percentage of the affected downstream modules over all candidates. The column *#FPs* lists the false positives for each bug and the column *P* shows the precision = 1 - *#FPs*/min(30, *#affs*). The two columns of *#constraints* indicate the maximum and average numbers of constraints collected from the module entries to the call sites of the buggy upstream function across all the candidates. The column *avg.* of *time(s)* shows the mean time that the intra-module impact analyzer needs to process a candidate module for the bug.

Table 1). For the affected modules, the analyzer identifies the impact by providing the inputs that can trigger the cross-project bugs. Due to the conservativeness of the intra-module analysis, our approach has 100% recall and the modules filtered out from the candidates are ensured to be unaffected.

Among the 1132 identified downstream modules, only 47 modules have ever been reported in the past. The number of newly found modules for the 31 bugs ranges from 1 to 260, with a total number of 1085. To examine the false positives of them, we take three steps to decide whether they are indeed affected. For each reported module, a test case is first generated with the inputs provided by our tool. Then, we configure the upstream project to the version where the cross-project bug happens and the downstream project to the version that our tool specifies. In addition, other depending projects are also configured to the appropriate versions for normal operation of the module. Last, we run the test case and observe whether the values of the input parameters of the buggy upstream function conform to the bug inducing preconditions when the function is invoked. If so, the module is considered truly affected; otherwise, the module is a false positive.

Due to the high cost of the examination, we can not validate all 1085 modules. For the bugs with *#affs* $\leq$ 30, we examine all the affected modules identified by our tool. For other bugs, we

Table 2: Number of Encoded Constructs in Various Types

FuncDef	If	Call	While	Assign	AugAssign	Return	Others
22627	33921	271893	97	116094	1699	37910	126

randomly select 30 modules from the identified ones and ensure each involved downstream project is under inspection. We totally validate 470 modules. Table 1 shows the results. The column *#FPs* lists the false positives for each bug and the column *P* shows the precision = 1 - *#FPs*/min(30, *#affs*). It can be seen that the precision of our approach is high, with an average of 92.1%. Apart from the bug N#4225 with a precision of 61.5%, the precisions for other bugs are not lower than 75.0%. Moreover, no false positives are found for 17 bugs (highlighted in Table 1). Bug N#4225 reports that *Numpy*'s function `numpy.log1p()` returns a wrong result with the input as an infinite number. For this case, the five false positives are all found in the project *Nilearn*. The reported *Nilearn* modules call the function `numpy.empty_like()` to randomly produce an array before using `numpy.log1p()`. We cannot obtain an infinite number after `numpy.empty_like()` is invoked when running the test cases, and thus the failure inducing precondition cannot be met. However, we did not model the behavior of `numpy.empty_like()` such that our tool reports that an infinite number is possible.

Seen from Table 1, our method filters out a large number of unaffected candidate modules that use the buggy methods. With our tool, developers can save at most 98.9% (=1-1.08% for N#8993) of the effort for a single bug by not inspecting the unaffected ones. For 31 bugs, the developers save 66.5% (=1-33.5%) of the effort in code review on average. The result means that our technique can effectively help the downstream developers to focus their efforts on a limited number of truly affected modules instead of wasting time on inspecting a mass of modules immune to the bugs.

To gain a broader view of our symbolic anlysis, Table 2 lists the number of various types of constructs that we encode, including function definition, if statement, assignment, augmented assignment, while statement, function call, return, and others (such as try statement and raise statement). Totally, we symbolically encode 25490 candidate downstream modules with 327 KLOC. Among those types of constructs, function calls are most frequently encountered, followed by if statements and assignments. Among these callees, 46.9% are commonly used libray functions that we have manually modeled, while others are resolved in the conservative way as described in 3.3.2. Considering the high precision of our approach, the considerable number of un-modeled function calls do not result in too many false positives, indicating that the conservative processing of calls is suitable in bug impact analyis. It ensures that the symbolic analysis is performed in a limited code range for individual modules (e.g., 13 LOC per candidate module on average) without much precision loss.

In addition to the number of encoded constructs of various types, we also provide Fig. 9 to show the number of symbolic constraints collected from each candidate module during impact analysis. Each bar in Fig. 9 represents the number of modules in which the number of collected constraints falls into the specific range. It can be seen that paths to the buggy upstream functions in 42.3% of candidate modules are encoded into 6 to 10 constraints.

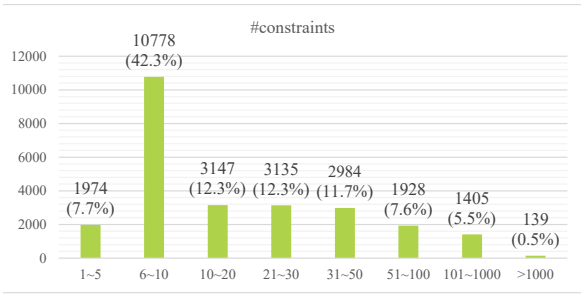


Figure 9: The distribution of numbers of constraints collected from candidate modules.

4.3.2 *Efficiency (RQ2).*

Dependence analysis. Table 3 shows the time used to analyze inter-project call dependencies. Due to limitations on space, we only present the information for six projects, i.e., *Numpy*, *Scipy*, *Astropy* [1], *Scikit-learn* [25], *Pandas* [23], and *Matplotlib* [12]. They are the core libraries in the scientific Python ecosystem and involve the most number of analyzed cross-project bugs in our study. The first column shows the name of the project. The columns *SLOC*, *#functions*, and *#calls* mean the numbers of the source lines of code, functions, and method calls to other projects in base version of the project, respectively. The column *#versions* shows the number of analyzed versions. The last two columns present the time used to extract the inter-project call relations from the base version of each project and the time to incrementally analyze all the versions. It can be seen that the time for processing base versions ranges from 0.59 hours to 1.36 hours with an average of 0.81 hours. The total time for each project ranges from 1.98 to 9.33 hours. Combining the graphs from individual projects, we construct an ecosystem-wide call dependence network for 22 projects of 121 versions, with a time cost of 50.13 hours. The network consists of 129,948 directed edges and takes 248.4MB in *MySql*. The time cost is acceptable because the dependence network is constructed in advance. It is only updated by updating the graphs of individual projects when new versions are committed.

Intra-module impact analysis. The two columns of *#constraints* in Table 1 indicate the maximum and average numbers of constraints collected from the module entries to the call sites of the buggy upstream function across all the candidates. The column *avg.* of *time(s)* shows the mean time that the intra-module impact analyzer needs to process a candidate module for the bug. The last column presents the total time used to analyze the impact of the bug, including the time for selecting the candidates from the ecosystem-wide dependence network and the time for processing all the candidates. The time cost of our approach is reasonable. It takes 0.860 seconds on average to analyze the local impact for every module. For 24 bugs, it takes no more than 100 seconds to estimate

Table 3: Time Used in Dependence Analysis

Project	SLOC(K)	#functions	#calls	#versions	B(h)	T(h)
Numpy	141	6730	56531	8	0.6	3.45
Scipy	190	10523	70060	9	0.88	4.71
Astropy	164	8820	53366	6	0.69	2.06
Scikit-learn	118	5003	42932	9	0.59	3.11
Pandas	213	13690	121577	11	1.36	9.33
Matplotlib	153	8166	52308	6	0.66	1.98

a) scikit-image/skimage/feature/register_translation.py

```
def register_translation(src_image, target_image, upsample_factor=1,
space="real"):
201     float upsampled_region_size = numpy.ceil(upsample_factor * 1.5)
203         dftshift = numpy.fix(upsampled_region_size / 2.0)
207         sample_region_offset = dftshift - shifts * upsample_factor
                                   numpy.ndarray
```

b) mne/io/egi/events.py

```
37 def _read_mff_events(filename, sfreq, nsamples):
65             marker = {                                        float
67                     'start_sample': int(np.fix(start * sfreq)),
```

Figure 10: Examples of how the impact of a cross-project bug is worked around (Unrelated code is omitted.)

the ecosystem-wide impact for each of them. The analysis time for N#6238 and N#11103 is raletively long since the two bugs each involves more than 9000 candidate modules. The result suggests that our technique is sufficiently fast to be used during fixing cross-project bugs. By integrating the approach into bug trackers, it can tell the impact of a cross-project bug in a short time.

4.4 Threats To Validity

We discuss the threats to validity during the evaluation. First, to validate the affected modules reported by our tool, we manually configured individual modules and their dependencies to the suitable versions to run the generated test cases. It is very time-consuming and costs substantial human efforts. Therefore, we only validated all the reported modules for 23 bugs. For each of the remaining eight bugs, we randomly sampled 30 modules. In total, we checked 470 modules. While we believe the results indicate the effectiveness of our technique, it is possible that results may be different for the unsampled modules and untested bugs.

Another threat concerns the generalization of our experimental results. We evaluated our tool using 31 cross-project bugs. These bugs were collected after lengthy manual inspection of bug reports to confirm whether they are cross-project ones. The failure inducing preconditions were also manually extracted from bug reports, which may be error-prone as we are dealing with others' projects. To mitigate the threat, we were very cautious during data collection and only retained the bugs with preconditions clearly indicated in bug reports. In addition, due to the scale of the project, the evaluation was conducted only on the scientific Python ecosystem. It is possible that our results may not generalize to other ecosystems. However, the individual components of our technique are designed in an ecosystem-agnostic fashion.

5 DISCUSSION

In this section, we discuss our findings from the evaluation and their inspiration for our future work.

5.1 Cross-project Impact Could Be Got Around

As mentioned above, a large percentage of the affected downstream modules identified by our approach have not been reported before, which to some extent indicates that the cross-project impact of upstream bugs is often hard to be noticed by developers. After inspecting these modules, we find that the wrong results produced by buggy upstream functions are sometimes worked around by downstream modules.

Fig. 10 shows such a real case. The function `fix()` in *Numpy* was reported to falsely return a *Numpy* array (numpy.ndarray) with zero dimension when it was applied to a scalar (numpy/numpy#8993). As *Numpy*'s documentation indicated, `numpy.fix()` should produce a scalar. The function `register_translation()` in the project *Scikit-image* used the buggy upstream function `numpy.fix()` with a float input (shown at line 201 in Fig. 10a). At line 203, the variable `dftshift` was the returned value of `numpy.fix()` with the data type of `numpy.ndarray`. The variable `dftshift` was then only used at line 205 as an operand. Since a 0-dimension `numpy.ndarray` was seen as a scalar in this operation, the wrong data type did not lead to a wrong output of `register_translation()` in *Scikit-image*. Therefore, though *Scikit-image* used the buggy upstream function `numpy.fix()` with the impact-triggering input, its users or developers were not likely to be aware of the wrong result produced by `numpy.fix()`.

Another downstream function `_read_mff_events()` in the project *Mne* also called `numpy.fix()`. Fig. 10b shows how `_read_mff_events()` used `numpy.fix()` (used as `np.fix()` in the code) and processed its output. The result of `start * sfreq` was a scalar. When it was passed to `numpy.fix()`, the buggy function falsely returns a wrong output of a 0-dimension array rather than a scalar. However, an explicit type cast `int()` was applied to the output and converted it to a correct type. Therefore, the wrong output is suppressed.

The two downstream modules worked around the upstream bug by using the wrong results in insensitive operation or processing the wrong results to suppress the bug for the time being. However, it is unclear if the downstream developers are aware of the possible wrong outputs and put the work-arounds intentionally. Nonetheless, the wrong output of the buggy upstream function may put the downstream module in risk. Without the awareness of the cross-project bug, if developers of *Scikit-image* modify the code to use `dftshift` in a type-sensitive operation in future, it may lead to some unpredictable consequences.

This case shows the possibility of designing the future bug impact analyzer to further tell whether the bug impact is worked around in affected downstream modules. This can be done by analyzing how the modules process and use the returned values of the buggy upstream functions. For example, considering the aforementioned case that the returned value is of a wrong data type, if the value is then used in a type-insensitive operation or data type convention is applied, we have the reason to conjecture that the wrong returned output will not interfere with later code of the module. To design the approach, we need to get hints from code or developers' experience. More specifically, we first have to classify the differences between the expected and the wrong outputs of the buggy upstream functions into several categories, such as data type difference or value difference. Then, we need to summarize which operations are likely to be insensitive to each type of difference or to eliminate the difference. Last, the analysis system needs to learn these hints and apply them to the downstream modules to decide whether they will get around the unexpected results of buggy upstream functions.

5.2 Keep the Ecosystem in Focus When Considering Bug Impact

As we have discussed in Section 1, the cross-project impact of bugs indicates the necessity of changing the developers' points of view during bug fixing. With the popularity of the collaboration in software development and the increasing trend of software ecosystem, understanding and repairing a bug can not be limited within its rooted project. Both the upstream and downstream sides should consider the bugs and their fixes from the perspective from the whole ecosystem. However in most cases, only when the cross-project bugs are submitted by the downstream projects, the upstream developers would check whether the fixes satisfied the demands of the reported downstream projects. For other bugs, they rarely ask for the suggestions or feedbacks for the fixing from other projects mostly due to the lack of the awareness of cross-project impact of bugs, and thus the proposed patches may not be satisfactory.

For the downstream projects, the developers should also take into account the potential threat for their located ecosystems when dealing with upstream bugs. As we have discussed, a part of downstream modules seldom use the upstream erroneous functions in the way that triggers the bugs within the projects. Therefore, the downstream developers may not be aware of the impact or even deal with the bugs. However, as members in the ecosystem, the affected modules are likely to be used by other projects. Once the input parameters that they pass to the buggy upstream functions via the directly affected downstream modules trigger the impact, it may cause unpredicted consequences. Such uncertain use of the downstream modules put the ecosystem in potential threat.

6 RELATED WORK

6.1 Cross-project Bugs

The increasing number of cross-project bugs have attracted growing attention from researchers. Canfora et al. [7] proposed an approach to identifying Cross-System-Bug-Fixings (CSBFs) in FreeBSD and OpenBSD kernels. They also employed social network analysis to associate the occurrences of CSBFs with the social characteristics of contributors. Ma et al. [21] concentrated on the common practices of developers in fixing cross-project bugs. They especially focused on downstream developers, and addressed the questions about how they found the root causes and coordinated to deal with upstream bugs. Ding et al. [14] studied the characteristics of workarounds which were usually proposed by downstream developers when facing cross-project bugs. Liu [20] studied third-party library upgrade bugs and developed an automated tool to repair them. Decan et al. [9] reported that failures in upstream packages brought more and more troubles to downstream projects. In contrast, our study focuses on the ecosystem-wide impact of cross-project bugs, and proposes a technique to identify the affected downstream modules.

6.2 Change Impact within Software Ecosystems

Projects within a software ecosystem co-evolve with each other through their inter-dependencies. Changes to one project including fixing a bug may cause ripple effects to many other downstream ones. Bavota et al. [2] found that upstream upgrades have strong effects on downstream projects when a downstream project depends on upstream frameworks or general services.

A large number of studies focus on the API changes in libraries [3, 4, 6, 13, 16, 18, 24, 28]. Hora et al. [16] characterized the impact of API evolution in the Pharo ecosystem by observing to what extent API changes propagate to other projects. Robbes et al. [24] investigated the ripple effects of API deprecations across a Smalltalk ecosystem, considering five aspects including the frequency, magnitude, duration, adaptation, and consistency. Bogart et al. [3] studied how developers reasoned about and applied changes in three software ecosystems: Eclipse, R/CRAN, and Node.js/npm by observing their differences in practice, polices, and tools applied when performing/avoiding a breaking change. Xavier et al. [28] analyzed the impact of API breaking changes on client projects in Java ecosystem. They concluded that most breaking changes did not have a massive impact on clients. Tools have also been developed to make breaking changes less harmful by easily applying patches to downstream projects [5, 28]. Additionally, API changes in Android ecosystem have also been investigated. McDonnell et al. [22] concluded that Android APIs evolved faster than client migration. Linares-Vasquez [19] analyzed how the number of questions in StackOverflow increased when APIs were changed. They showed Android developers were more active when they faced API modifications. To the best of our knowledge, although these studies have empirically confirmed that upstream changes have ripple effects on downstream projects, no existing work has proposed methods to automatically analyze cross-project bug impact with such fine-granularity and precision.

Recently, Hejderup et al. [15] proposed to construct an ecosystem call graph for dependency management. They made an initial evaluation on npm-based projects by executing test cases of npm packages in Jalangi. Compared with their work, our ecosystem-wide dependence analysis especially considered the scalability and dynamics of software ecosystems by processing each project independently and handling versions on an incremental basis.

7 CONCLUSION

We present an approach to analyzing the impact of cross-project bugs on software ecosystems by identifying the affected downstream modules (classes/methods). For a confirmed bug with known root cause function and failure inducing preconditions, we first leverage an ecosystem-wide dependence analysis to collect the candidate downstream modules. Then, we perform an intra-module analysis to encode the paths to the call sites of the buggy upstream function as symbolic constraints. By solving the constraints together with the failure inducing preconditions, the affected downstream modules are identified. Our evaluation on the scientific Python ecosystem shows that the approach is highly effective.

ACKNOWLEDGMENTS

This work is partially supported by the National Key R&D Program of China (2018YFB1003901), the National Natural Science Foundation of China (61872177, 61772259, 61832009, and 61772263), and NSF 1748764, 1901242, and 1910300.

REFERENCES

[1] Astropy. 2018. A community Python library for Astronomy. http://www.astropy.org/

[2] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (oct 2015), 1275–1317. https://doi.org/10.1007/s10664-014-9325-9

[3] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120. https://doi.org/10.1145/2950290.2950325

[4] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. 2016. Do developers deprecate APIs with replacement messages? A large-scale analysis on Java systems. In *Procedings of 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*. 360–369. https://doi.org/10.1109/SANER.2016.99

[5] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. 2018. On the use of replacement messages in API deprecation: An empirical study. *Journal of Systems and Software* 137 (2018), 306–321. https://doi.org/10.1016/j.jss.2017.12.007

[6] John Businge, Alexander Serebrenik, and Mark G.J. van den Brand. 2015. Eclipse API usage: the good and the bad. *Software Quality Journal* 23, 1 (2015), 107–141. https://doi.org/10.1007/s11219-013-9221-3

[7] Gerardo Canfora, Luigi Cerulo, Marta Cimitile, and Massimiliano Di Penta. 2011. Social interactions around cross-system bug fixings: The case of FreeBSD and OpenBSD. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. ACM Press, New York, New York, USA, 143–152. https://doi.org/10.1145/1985441.1985463

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT Solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. Vol. 4963 LNCS. 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[9] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. 2016. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In *Procedings of International Conference on Software Analysis, Evolution, and Reengineering*. 493–504. https://doi.org/10.1109/SANER.2016.12

[10] NumPy developers. 2018. NumPy. http://www.numpy.org/

[11] SciPy developers. 2019. SciPy library. https://www.scipy.org/scipylib/index.html

[12] The Matplotlib development team. 2018. Matplotlib: Python plotting. https://matplotlib.org/

[13] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. In *Journal of Software Maintenance and Evolution*, Vol. 18. 83–107. https://doi.org/10.1002/smr.328

[14] Hui Ding, Wanwangying Ma, Lin Chen, Yuming Zhou, and Baowen Xu. 2017. An empirical study on downstream workarounds for cross-project bugs. In *Proceedings of 2017 24th Asia-Pacific Software Engineering Conference*. IEEE, 318–327. https://doi.org/10.1109/APSEC.2017.38

[15] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER '18)*. ACM, New York, NY, USA, 101–104. https://doi.org/10.1145/3183399.3183417

[16] Andre Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stephane Ducasse, and Marco Tulio Valente. 2015. How do developers react to API evolution? The Pharo ecosystem case. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 251–260. https://doi.org/10.1109/ICSM.2015.7332471

[17] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. 2009. A sense of community: A research agenda for software ecosystems. In *Proceedings of the 31st International Conference on Software Engineering - Companion Volume*. 187–190. https://doi.org/10.1109/ICSE-COMPANION.2009.5070978

[18] Kamil Jezek, Jens Dietrich, and Premek Brada. 2015. How Java APIs break - An empirical study. *Information and Software Technology* 65, C (2015), 129–146. https://doi.org/10.1016/j.infsof.2015.02.014

[19] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. 2014. How do API changes trigger stack overflow discussions? a study on the Android SDK. In *Proceedings of the 22nd International Conference on Program Comprehension*. 83–94. https://doi.org/10.1145/2597008.2597155

[20] Yuefei Liu. 2017. *Understanding and Generating Patches for Bugs Introduced by Third-party Library Upgrades*. Ph.D. Dissertation. http://hdl.handle.net/10012/12762

[21] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yuming Zhou, and Baowen Xu. 2017. How do developers fix cross-project correlated bugs?: A case study on the GitHub scientific python ecosystem. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, Piscataway, NJ, USA, 381–392. https://doi.org/10.1109/ICSE.2017.42

[22] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of API stability and adoption in the android ecosystem. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*. 70–79. https://doi.org/10.1109/ICSM.2013.18

[23] Pydata. 2019. Pandas: Python data analysis library. https://pandas.pydata.org/

[24] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 56:1—-56:11. https://doi.org/10.1145/2393596.2393662

[25] Scikit-learn. 2018. Scikit-learn: Machine learning in Python. http://scikit-learn.github.io/stable

[26] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, New York, NY, USA, 45–54. https://doi.org/10.1145/1806799.1806811

[27] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the Dependency Conflicts in My Project Matter?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 319–330. https://doi.org/10.1145/3236024.3236056

[28] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *Proceedings of 24th IEEE International Conference on Software Analysis, Evolution, and Reengineering*. 138–147. https://doi.org/10.1109/SANER.2017.7884616