**DepOwl**: Detecting Dependency Bugs to Prevent Compatibility Failures

Zhouyang Jia*,†, Shanshan Li*,‡, Tingting Yu†, Chen Zeng*, Erci Xu*, Xiaodong Liu*,‡, Ji Wang*, Xiangke Liao*

*College of Computer Science
National University of Defense Technology
Changsha, China
{jiazhouyang, shanshanli, zengchen15, xuerci, liuxiaodong, wj, xkliao}@nudt.edu.cn

†Department of Computer Science
University of Kentucky
Lexington, USA
tyu@cs.uky.edu

**Abstract**—Applications depend on libraries to avoid reinventing the wheel. Libraries may have incompatible changes during evolving. As a result, applications will suffer from compatibility failures. There has been much research on addressing detecting incompatible changes in libraries, or helping applications co-evolve with the libraries. The existing solution helps the latest application version work well against the latest library version as an afterthought. However, end users have already been suffering from the failures and have to wait for new versions. In this paper, we propose DepOwl, a practical tool helping users prevent compatibility failures. The key idea is to avoid using incompatible versions from the very beginning. We evaluated DepOwl on 38 known compatibility failures from StackOverflow, and DepOwl can prevent 35 of them. We also evaluated DepOwl using the software repository shipped with Ubuntu-19.10. DepOwl detected 77 unknown dependency bugs, which may lead to compatibility failures.

**Index Terms**—Software dependency, Library incompatibility, Compatibility failure.

I. INTRODUCTION

Applications reuse as much existing code as possible for cost savings. Existing code is often in the form of libraries, which keep evolving and may introduce incompatible changes (e.g., changing interface signatures). Misuses of library versions containing incompatible changes may fail to lead to failures in applications. We refer to these failures as compatibility failures, or CFailures.

A CFailure involves three roles: library developers, application developers, and end users (library and application are relative concepts as an application itself may be a library for another application). As shown in Figure 1 library developers release two versions containing incompatible changes. The changes are classified into two types: backward incompatible change (BIC) (e.g., removing an interface), and forward incompatible change (FIC) (e.g., adding an interface). The solid (dashed) lines show how a BIC (an FIC) causes CFailures: if application developers develop an application based on the old (new) library version, end users may suffer from CFailures when linking the application to the new (old) library version. In either case, the incompatible change causes CFailures.

When incompatible changes happened, the three roles can prevent CFailures with different solutions: 1) library developers can undo the changes in the latest version; 2) application developers can update the application to adapt the changes; 3) end users can avoid using the incompatible library versions. There has been some research on detecting library changes [1]–[6]. These techniques focus on suggesting incompatible changes for library developers (i.e., the first solution). There has also been some work on detecting incompatible API usages in applications [7]–[10], or helping applications adapt library changes [11]–[14]. These techniques focus on helping application developers update the application (i.e., the second solution). In either of the above solutions, end users may have already suffered from CFailures and have to wait for new library/application versions. The third solution, on the other hand, is more light-weighted — end users can avoid CFailures from the very beginning without having to see the CFailures occur. However, there exists no research that can achieve this goal by helping users automatically select compatible library versions.

Some industrial settings use dependency management sys-
tems (DMSs) that can help users select right library versions. Examples include dnf [13] in RPM-based Linux distributions and apt [16] in Debian-based Linux distributions. However, DMSs have several practical limitations (more details in Section [1]):

1) DMSs require manual inputs from either application or library developers, which can be tedious and error-prone. For example, dnf requires application developers to specify version ranges of required libraries. apt asks library developers to maintain a symbol list provided by the library.

2) Manual inputs provided by developers may be outdated as the libraries evolve. For example, application developers specified the version range libfoo>=1.0, after which libfoo-2.0 is released and backward incompatible to libfoo-1.0. The version range should have been updated to 2.0>libfoo>=1.0.

3) Developers may not comply with the requirements of the DMSs. For example, apt requires libraries not to break backward compatibility in a package, but library developers may unintentionally introduce incompatibilities since there is no mechanism to guarantee the requirement.

Since DMSs depend on version ranges specified in specification files (e.g., the control file used by apt, or the spec file used by dnf) to resolve dependencies, the above limitations may introduce incompatible versions being included in the version ranges. In this case, we say there are dependency bugs (or DepBugs) in the specification files.

To address the limitations within DMSs, we propose a new approach, DepOwl, to detect DepBugs and prevent CFailures. DepOwl works at the binary level to check compatibility between libraries and applications instead of analyzing the API usage in source code of applications (e.g., compilers). This is advantageous for end users who prefer to install binary files without having to compile the source code. For example, end users often use the command apt install to download binary files. The source-code level compatibility cannot guarantee the compatibility of the binary files installed by the users.

Specifically, given the binaries of a library and an application, DepOwl automatically checks if the application is compatible to each version of the library, so it can help users select the right library versions to prevent CFailures. DepOwl contains three major steps. In the first step, DepOwl collects all potentially incompatible changes (e.g., add/remove/change interfaces) during the evolution of the library (from an old version to a new version), including both BICs and FICs. Next, DepOwl checks if the API usage in the target application matches the API definitions in either of the old and new library versions. If the change is a BIC (FIC) and the API usage matches the old (new) library version, the new (old) library version is regarded as an incompatible version. In the third step, DepOwl compares the incompatible version to all other library versions. Any version that is both backward and forward compatible to the incompatible version is also identified as an incompatible version. Users can prevent CFailures by avoiding using the reported incompatible versions.

A common usage scenario of DepOwl is to serve as a plugin for DMSs. Taking apt as an example, in Debian-based Linux distributions, apt helps users manage application dependencies. Each application contains a control file indicating its required libraries and version ranges. These ranges, however, may contain incompatible versions. DepOwl is able to detect incompatible versions, so that apt can avoid using incompatible versions when resolving dependencies, and users will be free of CFailures.

We evaluated DepOwl’s ability in preventing both known and unknown CFailures. We first evaluated DepOwl on 38 real-world known CFailures from StackOverflow, and DepOwl can prevent 35 of them. We also applied DepOwl to the software repository shipped with Ubuntu-19.10, the latest Ubuntu stable version at the time of writing. DepOwl detected 77 unknown DepBugs, which may cause CFailures.

In summary, the contributions of this paper are as follows:

1) We propose a lightweight solution to prevent CFailures when incompatible changes happen in libraries. Existing research work mainly focuses on fixing CFailures in new versions, but can not prevent the CFailures. Industrial DMSs can help users resolve dependencies, but still have limitations.

2) We design and implement DepOwl, a practical tool to detect DepBugs and prevent CFailures. DepOwl can collect incompatible changes in libraries, detect DepBugs in applications, and suggest incompatible versions to help users prevent CFailures.

3) DepOwl can prevent 35 out of 38 CFailures selected from StackOverflow, and detect 77 DepBugs in the repository shipped with Ubuntu-19.10. DepOwl is more accurate compared with baseline methods, and requires no human efforts.

II. EXISTING DMSs AND THEIR LIMITATIONS

Manual management of software dependencies is time-consuming and sometimes even error-prone, since an application may depend on many libraries, which keep evolving all the time. In this regard, a common approach, especially in the open-source community, is to use a dependency management system (DMS), e.g., pip [17] for Python, Maven [18] for Java, npm [19] for JavaScript, apt [16] and dnf [15] in Linux distributions.

These DMSs provide interfaces for developers to specify dependencies (i.e., the required libraries and corresponding versions), as well as repositories that contain all libraries. Developers manually specify dependencies, then the DMSs can automatically download and install the libraries from the repositories. For a required library, developers can specify a fixed version or a version range. Using a fixed version is a reliable solution because it has little to virtually zero CFailures, but it lacks flexibility because critical fixes in later versions of the library cannot be automatically included [20]. While using a version range increases flexibility since it can
automatically include critical fixes in later versions of the library, but decreases its reliability because the later versions may also introduce CFailures. There is a tradeoff between flexibility and reliability in these two approaches. Developers struggle to find the sweet spot [21].

Most DMSs leave this choice to application developers, who can manually limit the version range of each required library. Taking dnf as an example, dnf is the DMS used in RPM-based Linux distributions like Fedora. dnf requires application developers to specify the required libraries and version ranges (e.g., ocaml>=3.08), which may be outdated: 1) The version range may be too large as libraries evolve. For example, developers specify libfoo>=1.0 at first, after which libfoo-2.0 is released and backward incompatible with libfoo-1.0. In this case, the version range should be updated to 2.0>libfoo=1.0. 2) The version ranges may be too small as libraries evolve. For example, developers specify libfoo<1.0 at first, after which libfoo-2.0 is released and backward compatible with libfoo-1.0. In this case, the version range should be updated to libfoo<2.0.

To avoid these limitations, another solution is to maintain a symbols file by library developers. This solution is applied in apt, the DMS in Debian-based Linux distributions like Ubuntu. According to Debian policy [22]: 1) "ABI (Application Binary Interface) changes that are not backward-compatible require changing the soname [23] of the library"; 2) "A shared library must be placed in a different package whenever its soname changes". It means that two library versions should be placed in two library packages, when the versions are backward incompatible. These two packages, to some degree, can be regarded as two different libraries, e.g., libssl1.0.0 and libssl1.1. Library developers are required to maintain a symbols file [22], in which each line contains a symbol provided by the library, as well as the minimal version that the symbol is introduced. Then, the version range of this library can be inferred automatically by extracting symbols used by an application. The minimal version of the version range is the maximum value of introducing versions of all used symbols. The maximum version is not necessary since all versions are backward compatible in one package. Finally, the version range is used by apt to help users manage dependencies.

The above solution, however, is still limited since: 1) There is no mechanism to guarantee that library developers comply with the policy. Library developers may unintentionally introduce ABI incompatibilities between two versions, which have the same soname. Existing studies [6], [24] show 26%-33% of library versions violate semantic versioning, meaning libraries frequently introduce incompatibilities during minor version changes. 2) This solution only works for binary packages, since apt needs to analyze binary files to extract symbols used by the application. Application developers have to manually specify version ranges for source packages, which do not have binary files. In this case, apt will suffer from the same limitations as dnf. 3) Library developers need to manually update the symbols file when introducing forward incompatible changes. For example, when a struct type adds a field in a new library version, the introducing version of all symbols using the struct must be increased to the version at which the new field was introduced. Otherwise, a binary built against the new version of the library may be installed with a library version that does not support the new field. This is a common change during library evolutions, failing to update the introducing version of any symbol will lead to DepBugs. We will show a real-world example in Section III.

In summary, the DMSs supporting version ranges may introduce DepBugs — the ranges contain incompatible versions. In this paper, we focus on detecting and fixing DepBugs in the range-based DMSs, so that applications can achieve higher reliability without affecting flexibility.

III. MOTIVATION AND OVERVIEW OF DepOwl

In this section, we show a DepBug example which motivates us to design DepOwl. Based on the example, we introduce how DepOwl works at a high level.

Motivating example. From glib-2.39.1 to glib-2.39.2, the return types of some functions (e.g., g_hash_table_replace, g_hash_table_insert) changed from void to gboolean. These changes are: 1) backward compatible — a binary compiled against the old version will ignore the return value of the new version, and there is no error; 2) forward incompatible — a binary compiled against the new version may use the return value, where the old version returns void.

These changes may cause DepBugs in many applications (e.g., cockpit-202.1, homebank-5.2.2), where the return values of the changed functions are used. Figure 2 shows code

Fig. 3: Overview of DepOwl. DepOwl contains three major steps: collect incompatible changes, detect dependency bugs, and suggest incompatible versions.
TABLE I: Examples of DepOwl results.
(a) Collecting incompatible changes in libraries.

| Library | Change Versions | Change Content |
|---------|-----------------|----------------|
| glib    | <2.39.1, 2.39.2> | g_hash_table_replace adds return values |
| glib    | <2.39.1, 2.39.2> | g_hash_table_insert adds return values |

(b) Detecting DepBugs and suggesting incompatible library versions.

| Application      | Library       | DepBug | Incompatible Versions |
|------------------|---------------|--------|-----------------------|
| cockpit-202.1    | glib>=2.37.6  | v2.0   | 2.37.3<=glib<=2.39.1  |
| homebank-5.2.2   | glib>=2.37.3  | v3.0   | 2.37.3<=glib<=2.39.1  |

snippets of two applications. The usage of return values indicates any glib version returning void will be incompatible to the applications. However, in Ubuntu-19.10, cockpit-202.1 depends on glib>=2.37.6, and homebank-5.2.2 depends on glib>=2.37.3. Both the version ranges contain the incompatible version glib-2.39.1. Therefore, we say cockpit-202.1 and homebank-5.2.2 contain DepBugs since their version ranges contain incompatible versions.

The root cause of the DepBugs is that library developers do not update the introducing versions of the changed functions in the symbols file of the library.

The DepOwl approach. DepOwl can detect DepBugs in the above example, and prevent CFailures caused by the bugs. Figure 3 shows the overview of DepOwl, which contains three major steps. First, the root causes of CFailures are incompatible changes in libraries. DepOwl collects incompatible changes from any two successive library versions, including both BICs and FICs. For example, the above example contains two incompatible changes as shown in Table 1.

Second, one incompatible change may or may not result in CFailures. DepOwl analyzes usages of the changed element (e.g., g_hash_table_replace) in each application, and detects whether the old or new library version of the change is incompatible to the application. If yes, DepOwl reports a DepBug when the incompatible version is included in the required version range of the application. For the above example, the third column of Table 1 shows the incompatible versions that cause DepBugs.

Third, one incompatible change may cause multiple incompatible versions. DepOwl suggests all incompatible versions caused by each incompatible change. Users can prevent CFailures by avoiding using the incompatible versions. In this step, any version that is both backward and forward compatible to the version reported by the second step (e.g., glib-2.39.1 for cockpit-202.1) will also be regarded as an incompatible version. In our example, the changed functions return void in glib-2.39.1 and previous versions. Thus, the incompatible version range is glib<=2.39.1. Then, DepOwl calculates the intersection between the incompatible version range and the required version range. For example, the intersection for cockpit-202.1 is 2.37.3<=glib<=2.39.1.

There are three challenges in the design of DepOwl:

- **DepOwl** collects library changes that break either backward or forward compatibility, whereas existing tools mainly focus on detecting backward incompatibilities. To achieve this, we propose a heuristic rule to help DepOwl detect changes breaking forward compatibilities.

  - DepOwl detects if incompatible changes will cause DepBugs. This is challenging because the changes can involve different types (e.g., add a function, remove a parameter). To address this, we categorize the changes and derive a set of rules to detect DepBugs for each type.

  - DepOwl suggests all incompatible versions caused by each incompatible change. This is non-trivial because multiple changes may affect the same element. In this regard, DepOwl performs a global check across all versions to suggest incompatible ones for a changed element.

IV. DepOwl Approach

There have been some existing techniques (e.g. compilers) on analyzing API usages in applications to check if the application is compatible with a given library version. They work at the source-code level. However, end users often prefer to install binary files directly, instead of downloading source-code files and compiling the applications themselves. Therefore, the users often care more about the binary level compatibility. There has also been some work (e.g. ABI Tracker [25]) on detecting incompatibilities across different library versions at both source-code and binary levels. This work does not analyze the API usages in applications. As shown in Figure 4, in this paper, we focus on detecting binary level compatibility between libraries and applications. The compatibility at the source-code level cannot guarantee the compatibility at the binary level, such as modification of virtual tables of classes, change of type sizes of function parameters, change of values of enumeration elements, change of orders of struct fields, change of compilation directives, and so on.

Figure 5 shows two real-world examples that applications and libraries are compatible at the source-code level, but incompatible at the binary level. In the first example, three
APIs in the library openssl depend on the compilation directive OPENSSL_NO_SSL2. In openssl-1.0.1s, the directive is enabled; thus, the APIs are not available in library binaries. While in other versions, the directive is disabled by default. In this case, the source code of openssl-1.0.1s is the same as the source code of other versions, but applications using the APIs only fail when linking to openssl-1.0.1s. In the second example, the application ruby-2.5.5 depends on the library zlib, which defines z_crc_t as unsigned int after zlib-1.2.7. When compiling ruby against zlib-1.2.6, the compilation directive HAVE_TYPE_Z_CRC_T is not defined; thus, z_crc_t is unsigned long. When compiling ruby-2.5.5 against zlib-1.2.7, the compilation directive is defined; thus, z_crc_t is unsigned int. The application ruby-2.5.5 is source-code compatible with both zlib-1.2.6 and zlib-1.2.7. However, when the application is compiled against one version, it will be incompatible to another version at runtime.

Algorithm 1 shows how DepOwl suggests incompatible versions for each pair of library and application <lib, app> in a software repository (line 1). DepOwl first collects the set of incompatible changes IC from lib (line 2). Table 4 illustrates two examples of incompatible changes. Each incompatible change ic is a three-tuple: <library name, change name, change content>. The change versions contain the old and new versions involved in the change. For each ic (line 3), DepOwl then detects whether ic can cause a DepBug in app, and returns a two-tuple: <vold, vnew> (line 4). If the old (new) version of ic is incompatible to app and included in the version range required by app, vold (vnew) returns the old (new) version number, otherwise vold (vnew) returns -1. If vold (vnew) does not return -1 (line 5, line 8), DepOwl will suggest any version which is both backward and forward compatible to vold (vnew) as an incompatible version (line 6, line 9).

A. Collecting Incompatible Changes

The first component of DepOwl takes the library lib as input, and collects its incompatible changes IC. As shown in Figure 1, both BICs and FICs may result in CFailures. DepOwl needs to collect both kinds of library changes. There are existing tools of detecting compatibility problems in libraries, e.g., ABI-Tracker [25], a tool for checking backward compatibility of a C/C++ library. However, the existing tools mainly focus on backward compatibility problems. DepOwl transfers the forward problems into backward problems.

We refer to incompatible changes from version vold to version vnew as IC(vold, vnew):

\[
IC(v_{old}, v_{new}) = BIC(v_{old}, v_{new}) \cup FIC(v_{old}, v_{new}),
\]

(1)

where BIC(vold, vnew) and FIC(vold, vnew) stand for BICs and FICs from vold to vnew. DepOwl applies a heuristic rule: forward incompatibility from vold to vnew is equivalent to backward incompatibility from vnew to vold, formalized as:

\[
FIC(v_{old}, v_{new}) = BIC(v_{new}, v_{old}).
\]

(2)

According to Equation 1 and Equation 2, we can get:

\[
IC(v_{old}, v_{new}) = BIC(v_{old}, v_{new}) \cup BIC(v_{new}, v_{old}).
\]

(3)

Then, DepOwl collects both BIC(vold, vnew) and BIC(vnew, vold) by using the ABI-Tracker tool. For a library with N versions, DepOwl calculates all incompatible changes IC of lib:

\[
IC = \bigcup_{i=1}^{N-1} IC(v_i, v_{i+1}).
\]

(4)

During collecting library changes, DepOwl also consider the following factors: 1) Library soname [23]. DepOwl will skip the library changes between vold and vnew, if vold and vnew have different sonames. Library versions with different sonames will be packaged into different packages; thus will not lead to DepBugs. 2) Symbol versioning [26]. Symbol versioning supports multiple symbol versions in one library version. For example, in glibc-2.27, the symbol glob has two versions: glob@GLIBC_2.27 and glob@GLIBC_2.2.5 (‘@’ means the default version). DepOwl regards symbols with different versions as different symbols.

For each library version, DepOwl requires its binaries compiled with debug symbols. When the input is not available, DepOwl takes source code as input, and compiles the library with debug symbols itself (we provide compiling scripts to achieve this). DepOwl uses default compilation directives during the compiling process, and accepts custom directives provided by users at the same time.

B. Detecting Dependency Bugs

The second component of DepOwl is to analyze usages of the changed element of each ic in app, and detect whether vold or vnew is incompatible to app. If yes, DepOwl reports a DepBug when the incompatible version (i.e., vold or vnew) is
included in the version range required by `app`. When `app` does not specify any version range, `DepOwl` assumes it accepts all versions. As a common usage scenario of `DepOwl` is to detect DepBugs in a software repository. In this case, `DepOwl` takes the repository as input, and for each application package in the repository, `DepOwl` detects whether the change can lead to a DepBug. It is time consuming to analyze all application packages since a software repository may contain tens of thousands of application packages. In this regard, `DepOwl` splits the detecting process into two phases: filtering phase and detecting phase.

**Filtering phase.** `DepOwl` first filters out the application package that does not accept the library versions where `ic` happened. For example, `app` requires `libfoo>=3.0`, while the `ic` happened from `libfoo-1.0` to `libfoo-2.0`. To achieve this, `DepOwl` analyzes the dependencies of `app` (e.g. from `control` file in Ubuntu or `.spec` file in Fedora), and extracts the libraries required by `app`, as well as corresponding required version ranges. `DepOwl` checks if the library (where `ic` happens) is included in the required libraries, and if `v_{old}` and `v_{new}` of `ic` are included in the corresponding version range. When either of the above two conditions is not satisfied, it means `ic` can never affect `app`. In this case, `DepOwl` reports no DepBugs and stops analyzing.

Then, `DepOwl` filters out the application package that does not use the changed element in `ic`. For example, the library adds a parameter for a symbol, which is not used in `app`. In general, `ic` can be classified into two types according to the changed element: change a symbol (e.g., from "foo()" to "foo(node a)") and change a data type (e.g., from "struct node {int i;}" to "struct node {float f;}"). `DepOwl` analyzes the binary files contained in `app`. When `ic` changes a symbol, `DepOwl` checks if any binary file requires the symbol by using the `readelf` tool. When `ic` changes a data type, `DepOwl` collects all symbols that use the data type in the library, and checks if any binary file requires any symbol. If yes, it means `ic` can potentially lead to CFailures, and `DepOwl` starts the next phase. Otherwise, `DepOwl` stops analyzing, and reports no DepBugs.

**Detecting phase.** `DepOwl` analyzes the usage of the changed element and determines whether `v_{old}` or `v_{new}` is incompatible to `app`. If the change is a BIC (FIC) and the usage matches `v_{old}` (`v_{new}`), then `v_{new}` (`v_{old}`) will be regarded as the incompatible version. `DepOwl` takes the application binary file with debug symbols as input. When `ic` changes a symbol, `DepOwl` extracts the symbol signature from the binary file. When `ic` changes a data type, `DepOwl` extracts the data-type definition from the binary file. After that, `DepOwl` compares if the signature or definition is the same as that of `v_{old}` or `v_{new}`. If the above input is not available, `DepOwl` can also extracts the usage from source code. For example, when working on a software repository, many applications are released without debug symbols. In this case, `DepOwl` automatically downloads the source code of each application package.

When using the application source code, it is hard to extract symbol signatures or data-type definitions, since the header files are not available. `DepOwl` has to apply different rules to determine the incompatible version. For example, when `ic` adds a field in a `struct`, `DepOwl` needs to check if the additional field is used in the source code. When `ic` changes the type of a return value from `void` to `non-void`, `DepOwl` needs to check if the return value is used in the source code.

In this regard, we enumerate all types of incompatible changes in C/C++ libraries and define determination rules for each type. The classification and rules are shown in Table II. We classify library changes into 18 types related to enum (1-3), struct (4-7), variable (8-10), and function (11-18). The struct and enum types are data-type changes, while the variable and function types are symbol changes. For data-type changes (1-7), `DepOwl` needs to confirm that the application uses the changed element in source code, e.g., member for enum or field for struct. For symbol changes (8-18), `DepOwl` has already confirmed that the application uses the changed symbol in the filtering phase. For changes related to "add" or "remove" (1-2, 4-5, 8-9, 11-14, 16-17), once the application uses the changed element, `DepOwl` determines the incompatible version is `v_{old}` or `v_{new}`, respectively. For changes related to "change type" (6, 10, 15, 18), `DepOwl` analyzes the usages of changed element, and infers the type in source code. For example, from `zlib-1.2.6.1` to `zlib-1.2.7`, the return type of the function `get_crc_table` changed from `long` to `int`. In the source code of package `unqlz-0.65`, `DepOwl` finds "long *CRC_TABLE = get_crc_table();", i.e., the return type matches version 1.2.6.1. Thus, `DepOwl` determines 1.2.7 is the incompatible version. As for change type 3 and 7, it is hard to infer the member value or field order from source code. Thus, `DepOwl` cannot determine the incompatible version.

We tried to build a complete table with our best effort. We referenced online resources during the enumeration process. For example, changing an inherited class in C++ will generate two totally different symbols in binaries due to name mangling. In this case, `DepOwl` will report function `add` and function `remove`. Also, `DepOwl` is designed to be flexible to incorporate new rules.

`DepOwl` uses srcML [31], a source-code analysis infrastructure, to achieve the above analyzing. The source code cannot be compiled since the lack of header files, while srcML provides lexical analysis and syntax analysis for non-compilable source code. `DepOwl` returns a two-tuple: `<v_{old}, v_{new}>` in this step. If the old (new) version in `ic` is incompatible to `app` and included in the version range required by `app`, `v_{old}` (`v_{new}`) returns the old (new) version number, otherwise `v_{old}` (`v_{new}`) returns -1.

C. Suggesting Incompatible Versions

We refer to the incompatible version reported in the above step (i.e., `v_{old}` or `v_{new}`) as `v_{bug}`. A library change may lead to multiple incompatible versions beyond `v_{bug}`. In this component, `DepOwl` detects all library versions that are incompatible to `app` caused by `ic`. To achieve this, `DepOwl` cannot simply assume the versions less than or greater than `v_{bug}` as incompatible versions, since the changed element in `ic` may change...
Table II: Rules for determining Dep Bugs.

| ID | Types of Incompatible Changes | DepOwl Rules | Incomp. Version |
|----|-------------------------------|--------------|----------------|
| 1  | Enum adds member              | Use the member | v_{old}         |
| 2  | Enum removes member           | Use the member | v_{new}         |
| 3  | Enum changes member value     | Use the member | -              |
| 4  | Struct adds field             | Use the field | v_{old}         |
| 5  | Struct removes field          | Use the field | v_{new}         |
| 6  | Struct changes field type     | Use the field & Match the filed type | v_{new}, v_{old} |
| 7  | Struct changes field order    | Use the field | -              |
| 8  | Global variable adds          | -             | v_{old}         |
| 9  | Global variable removes       | -             | v_{new}         |
| 10 | Global variable changes type  | Match the var type | v_{old}, v_{new} |
| 11 | Function adds                 | -             | v_{old}         |
| 12 | Function removes              | -             | v_{new}         |
| 13 | Function adds para            | Use the para  | v_{old}         |
| 14 | Function removes para         | Use the para  | v_{new}         |
| 15 | Function changes para type    | Match the para type | v_{old}, v_{new} |
| 16 | Function adds return value    | Use the function ret | v_{old}         |
| 17 | Function removes return value | Use the function ret | v_{new}         |
| 18 | Function changes return type  | Match the ret type | v_{old}, v_{new} |

† The struct related rules (4-7) also apply for union or class.

again in another ic. For example, in zlib, developers remove the function gzgetc (change gzgetc to a macro for speed) from zlib-1.2.5.1 to zlib-1.2.5.2. After that, the developers restore gzgetc for compatibility from zlib-1.2.5.2 to zlib-1.2.5.3 [32]. In this regard, DepOwl checks compatibilities of the changed element of ic across all versions of lib, and any version that is both backward and forward compatible to v_{bag} will be regarded as an incompatible version.

We refer to the changed element in ic as ele. Suppose there are N library versions. For ∀i ∈ [1, N], DepOwl calculates isIV(v_i), a Boolean value indicating whether v_i is an incompatible version:

\[ isIV(v_i) = \neg bce(v_{bag}, v_i, ele) \land \neg bcf(v_{bag}, v_i, ele), \]

where bce(v_{bag}, v_i, ele) and bcf(v_{bag}, v_i, ele) return Boolean values, meaning if ele breaks backward compatibility or breaks forward compatibility from v_{bag} to v_i, respectively. If yes, return 1, otherwise return 0. Similar to Section IV-A we have:

\[ bcf(v_{bag}, v_i, ele) = bce(v_i, v_{bag}, ele). \]

Therefore, DepOwl transforms the above two equations to:

\[ isIV(v_i) = \neg bce(v_{bag}, v_i, ele) \land \neg bcf(v_i, v_{bag}, ele). \]

Then, DepOwl outputs a list of Boolean values ISIV, each of them indicates whether a version is incompatible (i.e., 1) or not (i.e., 0):

\[ ISIV = [isIV(v_1), isIV(v_2), ..., isIV(v_N)]. \]

For each element (e.g. isIV(v_i)) in ISIV, if isIV(v_i) equals to 1, and v_i belongs to the version range required by app, DepOwl regards v_i as an incompatible version. Taking the application cockpit-202.1 as an example, the required version range is glib>=2.37.6; while for ∀j ∈ (glib<=2.39.1), isIV(v_j) equals to 1. DepOwl suggests the incompatible versions are 2.37.6<=glib<=2.39.1. For an application that is not managed in a software repository, DepOwl assumes that it accepts all library versions since there is no version ranges.

For the given app and lib, DepOwl reports a set of incompatible versions for each ic: IV_{lib, app, ic}. Suppose there are M incompatible changes in lib. Finally, DepOwl suggests all incompatible versions between app and lib:

\[ V_{lib, app} = \bigcup_{i=1}^{M} IV_{lib, app, ic_i}, \]

where ic_i stands for the i̇th incompatible change.

V. Evaluation

To evaluate DepOwl, we consider three research questions:

RQ1: How effective is DepOwl at preventing known CFailures? This question examines the recall of DepOwl by calculating the percentage of CFailures that can be prevented by DepOwl among all known CFailures.

RQ2: How effective is DepOwl at preventing unknown CFailures? This question evaluates the precision of DepOwl by calculating the percentage of correct results among all results reported by DepOwl.

RQ3: How does DepOwl compare with existing methods? This question compares DepOwl with two widely used DMSs (i.e., apt and dfn), as well as the dependencies declared in the build systems (e.g., autoconf or cmake) by developers.

A. Datasets and Experiment Designs

For each research question, we introduce the preparation of datasets, and the measurements used during the evaluation.

RQ1: Preventing known CFailures. We collected known CFailures from StackOverflow by using keyword search. However, simple keywords (e.g., library, dependency, version, etc) may result in tens of thousands of issues, and introduce massive manual efforts in the following analysis. Instead, we used the error messages when users came across compatibility problems as keywords. For example, when a library removes a symbol, the application will echo "symbol lookup error" at runtime. When a library symbol adds or removes a parameter, the compiler will complain "too few/many parameter to function" at compiling time. In total, we collected 529 issues by using error-message searching.

We then manually analyzed root causes of these issues and found 69 issues involve incompatible changes in libraries. These changes lead to CFailures through misuses of library versions. While others are mainly caused by dependency problems but not related to compatibility. Among the 69 issues, 38 of them involve C/C++ programs. Since the current version of DepOwl handles C/C++ programs, we used the 38 issues to answer RQ1. The applications of 23 issues are code snippets provided by the original posters, while other issues involved 12 mature projects including servers (e.g., Httpd, MongoDB) and clients (e.g., Eclipse, Qt) from different domains.

The data and source code in this paper are publicly available in https://github.com/ZhouyangJia/DepOwl.
Since the 38 issues were selected by searching error messages, they may not cover certain types of compatibility breaking changes (Table II) that do not produce observable symptoms. For example, in Table II, "changing member values in a enum type (ID 3)" and "changing field orders in a struct type (ID 7)" may result in errors in a program, but will not generate error messages. Therefore, the 38 issues cannot cover the changes of ID 3 and ID 7. It is hard to collect incompatibilities that have no observable failures, since users cannot be sure if they are actual bugs, thus may not report issues.

We measured the effectiveness of preventing known CFailures in terms of whether DepOwl can prevent the CFailures in the 38 C/C++ related issues. To achieve this, DepOwl needs to detect DepBugs in these issues. DepBugs happen when the version ranges required by applications contain incompatible versions. Fixing the DepBugs helps users avoid using incompatible versions and prevent CFailures. When an application does not specify a version range, DepOwl assumes that the application accepts all library versions.

**RQ2: Preventing unknown CFailures.** We used the software repository shipped with Ubuntu-19.10 (the latest stable version at the time of writing) to evaluate DepOwl, since Ubuntu uses apt, which can resolve dependencies automatically, while other DMSs mainly depend on application developers to manually input dependencies. The repository includes 61,068 packages; each package can be either an application package or a library package. There are 32,069 library packages, which are depended by at least one other package. For each library package, we count the number of application packages that depend on it. We choose the top 1% (i.e., 32) library packages, which are from 26 different libraries (one library may generate multiple packages, e.g., the qt library generates libqt5core5a, libqt5gui5 etc.). For each chosen library, we collect its versions released during about last ten years, and get 841 versions in total (i.e., 32.2 versions for each library on average).

It is hard to directly measure the effectiveness of preventing unknown CFailures, since the unknown CFailures do not happen as yet. Instead, we measure the effectiveness in terms of whether DepOwl can detect unknown DepBugs in the software repository, and prevent potential CFailures caused by the DepBugs. In specific, for each application package from the software repository, DepOwl detects whether there are DepBugs with regard to the chosen library packages, i.e., the version ranges required by the application package contain incompatible versions. If yes, DepOwl suggests the incompatible versions that may cause CFailures.

**RQ3: Comparing with existing methods.** We used the same dataset in RQ1 to compare DepOwl with existing methods, and calculated the percentage of issues that can be prevented if the original posters use existing methods.

We first compared DepOwl with two DMSs used in industry: 1) apt, used in RPM-based Linux distributions, where application developers manually specify version ranges of required libraries; 2) apt, used in DEB-based Linux distributions, where library developers maintain a symbols file. We then compared DepOwl with building scripts (e.g., configure.ac or CMakeList.txt) shipped with application source code, since developers often declare version ranges in the scripts.

**RQ1: Preventing known CFailures.** Two authors manually evaluated whether DepOwl can prevent the 38 known CFailures by analyzing if the incompatible versions suggested by DepOwl contain the incompatible version used by the original poster. The result shows DepOwl successfully suggests incompatible versions for 35 of the 38 C/C++ related issues. The complete list of these issues is available in our supplementary materials. Each issue in the list contains the issue ID, the application name, the library name, and the incompatible versions suggested by DepOwl. Taking issue 27561492 as an example, library libpcre adds function pcrecpp::RE::Init from libpcre-5.0 to libpcre-6.0, and changes its parameter type from libpcre-6.7 to libpcre-7.0. Therefore, DepOwl reports two library changes. Meanwhile, the application mongodbb-2.4 uses pcrecpp::RE::Init, and the parameter type is the same as the type from libpcre-6.0 to libpcre-6.7. Thus, DepOwl reports [Vinit, 5.0] ∪ [7.0, Vlast] as the incompatible versions.

On the other hand, DepOwl reported three false negatives. Two cases were caused by compilation directives, e.g., the original poster executed and compiled an application on different OS, where the libraries may be compiled with different directives. DepOwl cannot infer such directives, and thus generates false negatives. The last case missed version information and might have used a very old library version. DepOwl can
prevent CFailures in 35 out of the 38 issues. This result indicates DepOwl can effectively prevent real-world CFailures in terms of recall.

RQ2: Preventing unknown CFailures. DepOwl collected 27,413 incompatible changes from the 841 versions of the 26 libraries. For each change, DepOwl detects if the change can cause a DepBug for each application package. DepOwl detected 77 DepBugs, of which 49 are caused by backward incompatible changes and 28 are caused by forward incompatible changes. These DepBugs involve 69 application packages and 7 library packages. Table II illustrates one bug for each application package. The complete DepBug list is available in our supplementary materials. For example, in the first bug, the application qgis-providers_3.4.10 depends on the library sqlite3_module from 3.7.6.3 to 3.7.7. The application used the new filed; thus 3.7.6.3 is an incompatible version. DepOwl then suggests all incompatible versions: [3.5.9, 3.7.6.3].

We searched evidence from new library versions, new application versions, or software repositories to evaluate if the 77 DepBugs have been handled in different ways. If not, we further reported them to the repository maintainers. Among the 77 DepBugs, library developers undo the library changes of 37 cases in later library version. It means applications may have CFailures when using the library versions before undoing the changes. Application developers update the application to adapt the changes in 3 cases, meaning the old application version may have CFailures. Besides, 24 DepBugs are fixed in the latest version of Ubuntu or Debian. Although these bugs have been handled in different ways, they had been in the system for a long period of time, posing threats to the system reliability. For example, library developers fixed an incompatible version, which had already been released and affected applications. DepOwl is able to prevent these impacts from the very beginning.

For the other 13 cases, we report them to the Ubuntu community, 4 of them have been confirmed by developers, and 8 are pending for response. So far, we only found one potential false-positive case. DepOwl reported that the library kcoreaddons-5.19 is incompatible to the application rtkward, which depends on kcoreaddons>=5.19. The developer agreed that the incompatibility may exist, but kcoreaddons-5.19 is not actually used in any Ubuntu release (Xenial uses kcoreaddons-5.18, Bionic uses kcoreaddons-5.40), thus has zero impact. This result indicates DepOwl can effectively detect real-world DepBugs in terms of precision.

This experiment took about 30 hours in a virtual machine with a dual-core CPU and 4G memory. The filtering and detection phases took about five hours (excluding downloading packages). The majority of time was spent on collecting library changes of history versions. This process is one-time effort, since the latest library version can be analyzed incrementally. The execution time of each library depends on its scale and type. When analyzing large C++ libraries like Qt, DepOwl may need dozens of minutes for each pair of versions. Meanwhile, some other libraries only need several seconds.

Fig. 6: Version ranges of different baselines.

RQ3: Comparing with existing methods. We compared DepOwl with three existing methods, i.e., dnf for .rpm packages, apt for .deb packages, and the building system. For each StackOverflow issue used in RQ1, two authors manually evaluated if the CFailure can be prevented by using existing methods when the original poster used the existing methods at first. Taking issue 30594269 as an example, webkit has "symbol lookup error" when linking to libsoup. The incompatible version range of libsoup is [Vinit, 2.29.6]. The version ranges of libsoup in three baselines accepted by webkit are [2.33.6, Vlast], [2.29.90, Vlast], [2.33.6, Vlast], respectively. Thus, all the three baselines can prevent the failure in this issue. Figure 6 lists the files where we get these version ranges, including the webkitgtk.spec file in the .rpm package, the control file in the .deb package, and the configure.ac file in the building system of source code.

Figure 7 shows the results regarding the comparison among DepOwl and the three baselines. DepOwl can prevent CFailures in 35 issues whereas the baselines can prevent CFailures in 3, 7, 5 issues, respectively. Besides, DepOwl does not report any problems in 3 issues (i.e., 3 false negatives), while the baselines do not report any problems in 27, 27, 26 issues. This is because 23 issues were caused by code snippets provided by the original posters. These code snippets are not managed in any DMSs or build systems. Last but not the least, the baselines report DepBugs in 8, 4, 7 issues (i.e., the version range contains incompatible versions). DepOwl successfully prevents 35 CFailures, whereas the best baseline prevents 7 CFailures. The detailed results are available in our supplementary materials. This result indicates DepOwl is more accurate than the three baselines. In the mean time, DepOwl requires no human efforts, while the baselines require manual inputs from either library developers or application developers.

VI. DISCUSSION AND FUTURE WORKS

In this section, we discuss limitations in the design of DepOwl, as well as future works with regard to the limitations.

Debug symbols of libraries. To collect incompatible changes (in Section IV-A), DepOwl requires all versions of the library as inputs. Each version should be in the source code form or the binary form with debug symbols. For the binary form, most libraries are released without debug symbols, and do not meet the requirement of DepOwl. As
for the source code form, we need to compile the source code so that DepOwl can collect Application Binary Interface (ABI) changes. DepOwl provides scripts to automate the compiling process. This is still limited since DepOwl uses the default compilation directives; thus cannot collect ABI changes triggered by other directives. As a result, developers have to provide the compilation directives, or DepOwl may cause false negatives.

• Future work: The most convenient way to avoid this limitation is to suggest library developers to release binaries with debug symbols when releasing new versions. This practice actually has been applied in some libraries. For example, in the software repository of Ubuntu-19.10, there are 753 packages with the suffix ‘-dbg’ containing debug symbols.

Code analysis in applications. When detecting dependency bugs (in Section IV-B), DepOwl requires application binaries compiled with debug symbols. This input is not available in most applications managed in existing DMSs. Alternatively, DepOwl has to use source code as input, but correct usages in source code do not indicate the application is free of CFailures in the binary form. For example, the second example of Figure 5 shows the usage of get_crc_table in ruby-2.5.5, which works well against both zlib-1.2.6 and zlib-1.2.7 in source code level: when ruby-2.5.5 is compiled against zlib-1.2.7, the return type z_crc_t is int; when ruby-2.5.5 is compiled against zlib-1.2.6, the return type z_crc_t is long. However, ruby-2.5.5 may have CFailures when compiled against one version and linked to another version at runtime. This limitation will lead to false negatives.

• Future work: DepOwl will provide an interface for application developers to indicate a fixed version for each library. This manual effort is the same to most DMSs like pip or Maven. Thus, DepOwl can compile the source code against the fixed library version.

Limitations when using ABI-Tracker. DepOwl uses ABI-Tracker to collect incompatible changes of a target library. ABI-Tracker takes source code of the library history versions as inputs and compiles each version with default directives. This process may introduce both false positives and false negatives. For example, in the first example of Figure 5, ABI-Tracker reports that openssl-1.0.1s removes three symbols. However, users will not encounter failures when disabling OPENSSL_NO_SSL2. In this case, DepOwl may report false positives, although no false positives directly related to ABI-Tracker are generated in our experiment. On the other hand, when incompatible changes can only be triggered by specific directives, ABI-Tracker may generate false negatives and thus cause DepOwl to report false negatives. For example, two out of three false negatives in RQ1 are caused by compilation directives not correctly identified by ABI-Tracker.

Impacts of compilation directives. The compilation directives of a target library may affect the symbols and data types provided by the library, and further affect the results of DepOwl. Since ABI-Tracker uses default compilation directives to compile each library version, it may cause DepOwl to report false negatives (as discussed in the above paragraph). We have mitigated this impact by directly analyzing the binaries of the target libraries without the need of providing compilation directives. In the case where binaries are not available, DepOwl accepts the directives from users for compiling. In our evaluation, we manually input the directives in most cases. For the two cases that we cannot obtain the directives in RQ1, DepOwl reports two false negatives, since the directives are hard to be inferred automatically.

VII. RELATED WORKS

We briefly classify the existing works into three types:

Library changes. Many works are targeted at library changes. Bagherzadeh et al. [33] studied the size, type and bug fixes in 8,770 changes that were made to Linux system calls. Brito et al. [34] identified 59 breaking changes and asked the developers to explain the reasons behind their decision to change the APIs. Dig et al. [35], [36] discovered that over 80% of changes that break existing applications are refactorings. Li et al. [37] investigated the Android framework source code, and found inaccessible APIs are common and neither forward nor backward compatible. Li et al. [38] and Wang et al. [39] studied API deprecation in the Android ecosystem and Python libraries. McDonnell et al. [40] found Android updates 115 API per month, and 28% usages in client applications are outdated with a median lagging of 16 months. Sawant et al. [41] investigated why API producers deprecate features, whether they remove deprecated features, and how they expect consumers to react. Brito et al. [1] identified API breaking and non-breaking changes between two versions of a Java library. Foo et al. [6] presented a static analysis to check if a library upgrade introduces an API incompatibility. Meng et al. [4] aggregated the revision-level rules to obtain framework-evolution rules. Mezzetti et al. [5] proposed type regression testing to determine whether a library update affects its public interfaces. Ponomarenko et al. [2] presented a new method for automatic detection of backward compatibility problems at the binary level. Wu et al. [3] proposed a hybrid approach to identify framework evolution rules.

These works are targeted at detecting changes, refactorings and rules during library evolutions. While DepOwl is targeted at preventing failures caused by the results of these works.

Application failures. Some works focus on CFailures in applications. Cai et al. [42] studied compatibility issues in
62,894 Android app to understand the symptoms and causes of these issues. Cossette et al. [43] studied techniques to help migrate client code between library versions with incompatible APIs. Dietrich et al. [44] studied partially upgrading systems, and found some crucial verification steps are skipped in this process. Jezek et al. [45] studied the compatibility of API changes, and the impact on programs using these libraries. Lamothe et al. [46] reported their experience migrating the use of Android APIs based on documentation and historical code changes. Linares-Vásquez et al. [47] studied how the fault-and change-proneness of APIs relates to applications’ lack of success. Xavier et al. [48] conducted a large-scale study on historical and impact analysis of API breaking changes. Balaban et al. [49] presented an approach to support client refactoring for class library migration. He et al. [7] and Xia et al. [49] studied API compatibility in Android. Henkel et al. [12] captured API refactoring actions, and users of the API can then replay the refactoring engine to bring their client software components up to date. Jezek et al. [10] proposed an approach that analyses the byte-code of Java classes to find type inconsistencies cross components. Li et al. [8] proposed a model for understanding the lifecycle of the Android APIs, and analyzing app that can lead to potential compatibility issues. Perkins et al. [13] proposed a technique to generate client refactoring actions for Java projects subject to risk of dependency conflicts. Xing et al. [14] recognized the API changes of the reused framework, and proposed plausible replacements to the obsolete API based on working examples.

These works focus on detecting incompatible API usages and helping applications co-evolve with library evolutions, so that the latest application version works well. While DepOwl can prevent CFailures for users’ in-use versions.

Application-library dependencies. There are many works address application-library dependencies. Bavota et al. [50] studied the evolution of dependencies between projects in the Java subset of the Apache ecosystem. Bogart et al. [51] studied three software ecosystems to understand how developers make decisions about change and change-related costs. Decan et al. [52] compared semantic-versioning compliance of four software packaging ecosystems, and studied how this compliance evolves over time. Decan et al. [53] analyzed the similarities and differences between the evolution of package dependency networks. Derr et al. [20] studied library upatability in 1,264,118 apps, and found 85.6% libraries could be upgraded by at least one version. Dietrich et al. [21] studied developers’ choices between fixed version and version range from 17 package managers. Jezek et al. [54] provided evidences that four types of problems caused by resolving transitive dependencies do occur in practice. Kikas et al. [55] analyzed the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems. Kula et al. [56] studied 4,600 GitHub projects and 2,700 library dependencies to understand if developers update their library. Mirhosseini et al. [57] studied 7,470 GitHub projects to understand if automated pull requests help to upgrade out-of-date dependencies. Pashchenko et al. [58] studied whether dependencies of 200 OSS Java libraries are affected by vulnerabilities. Raemaekers et al. [24] investigated semantic versioning, and found one third of all releases introduce at least one breaking change. Xian et al. [59] conducted an experience paper to evaluate existing third-party library detection tools. Wang et al. [60] conducted an empirical study on dependency conflict issues to study their manifestation and fixing patterns. Zerouali et al. [61] analyzed the package update practices and technical lag for the npm distributions.

These works mainly assist people in understanding application-library dependencies. While DepOwl is the first research work to help users avoid incompatible application-library dependency automatically. Huang et al. [62] and Wang et al. [63] designed tools to detect dependency conflicts for Maven and PyPI ecosystems. These tools focused on the diamond dependency problem, which detects conflicts among different dependencies. They assume each dependency itself is correct, whereas DepOwl detects bugs within dependencies.

VIII. CONCLUSION

In this paper, we find CFailures are caused by using incompatible library versions, which are hard to be prevented by the existing research works or industrial DMSs. To fill this gap, we design and implement DepOwl, a practical tool to prevent CFailures by avoiding incompatible versions. DepOwl can detect unknown DepBugs in the software repository shipped with Ubuntu-19.10, and prevent CFailures in real-world issues collected from StackOverflow. However, DepOwl still has limitations in practice. With limited helps from library developers (release binaries with debug symbols) and application developers (provide one required library version), DepOwl could achieve higher accuracy. As a result, applications could be both flexible for library evolutions and reliable for CFailures.

REFERENCES

[1] A. Brito, L. Xavier, A. Hora, and M. T. Valente, “Apidiff: Detecting api breaking changes,” in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), March 2018, pp. 507–511.
[2] A. Ponomarenko and V. Rubanov, “Backward compatibility of software interfaces: Steps towards automatic verification,” Programming and Computer Software, vol. 38, no. 5, pp. 257–267, Sep 2012. [Online]. Available: https://doi.org/10.1134/S0361176812050052.
[3] W. Wu, Y. Gouhèneuc, G. Antoniol, and M. Kim, “Aura: a hybrid approach to identify framework evolution,” in 2010 ACM/IEEE 32nd International Conference on Software Engineering, vol. 1, May 2010, pp. 325–334.
[4] S. Meng, X. Wang, L. Zhang, and H. Mei, “A history-based matching approach to identification of framework evolution,” in Proceedings of the 34th International Conference on Software Engineering, ser. ICSE 12. IEEE Press, 2012, pp. 353–363.
[5] G. Mezzetti, A. Moller, and M. T. Torp, “Type regression testing to detect breaking changes in node.js libraries,” in 32nd European Conference on Object-Oriented Programming (ECOOP) 2018, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Millstein, Ed., vol. 109. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 7:1–7:24. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9212.
[47] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, “Api change and fault proneness: A threat to the success of android apps,” in Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 477–487. [Online]. Available: https://doi.org/10.1145/2491411.2491428

[48] L. Xavier, A. Brito, A. Hora, and M. T. Valente, “Historical and impact analysis of api breaking changes: A large-scale study,” in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Feb 2017, pp. 138–147.

[49] H. Xia, Y. Zhang, Y. Zhou, X. Chen, Y. Wang, X. Zhang, S. Cui, G. Hong, X. Zhang, M. Yang, and Z. Yang, “How android developers handle evolution-induced api compatibility issues: A large-scale study,” in Proceedings of the 42th IEEE/ACM International Conference on Software Engineering, ser. ICSE, 2020.

[50] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, “How the apache community upgrades dependencies: an evolutionary study,” Empirical Software Engineering, vol. 20, no. 5, pp. 1275–1317, Oct 2015. [Online]. Available: https://doi.org/10.1007/s10664-014-9325-9

[51] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, “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, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 109–120. [Online]. Available: https://doi.org/10.1145/2950290.2950325

[52] A. Decan and T. Mens, “What do package dependencies tell us about semantic versioning?” IEEE Transactions on Software Engineering, pp. 1–1, 2019.

[53] A. Decan, T. Mens, and P. Grosjean, “An empirical comparison of dependency network evolution in seven software packaging ecosystems,” Empirical Software Engineering, vol. 24, no. 1, pp. 381–416, Feb 2019. [Online]. Available: https://doi.org/10.1007/s10664-017-9589-y

[54] K. Jezek and J. Dietrich, “On the use of static analysis to safeguard recursive dependency resolution,” in 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, Aug 2014, pp. 166–173.

[55] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, “Structure and evolution of package dependency networks,” in Proceedings of the 14th International Conference on Mining Software Repositories, ser. MSR 17. IEEE Press, 2017, pp. 102–112. [Online]. Available: https://doi.org/10.1109/MSR.2017.55

[56] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, “Do developers update their library dependencies?” Empirical Software Engineering, vol. 23, no. 1, pp. 384–417, Feb 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[57] S. Mirhosseini and C. Parnin, “Can automated pull requests encourage software developers to upgrade out-of-date dependencies?” in Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ser. ASE 2017. IEEE Press, 2017, pp. 84–94.

[58] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, “Vulnerable open source dependencies: Counting those that matter,” in Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ser. ESEM 18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 319–330. [Online]. Available: https://doi.org/10.1145/3239235.3268920

[59] Z. Xian, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, “Automated third-party library detection for android applications: Are we there yet?” in Proceedings of the 35th ACM/IEEE International Conference on Automated Software Engineering, ser. ASE, 2019.

[60] Y. Wang, M. Wen, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, “Watchman: Monitoring dependency conflicts for python library ecosystem,” in Proceedings of the 42th IEEE/ACM International Conference on Software Engineering, ser. ICSE, 2020.

[61] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. González-Barahona, “An empirical analysis of technical lag in npm package dependencies,” in New Opportunities for Software Reuse, R. Capilla, B. Gallina, and C. Cetina, Eds. Cham: Springer International Publishing, 2018, pp. 95–110.