This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

PACKHUNTER: Recovering Missing Packages for C/C++ Projects

Rongxin Wu, Zhiling Huang, Zige Tian, Chengpeng Wang, and Xiangyu Zhang

The reproducibility of software artifacts is a critical aspect of software development and application. However, current research indicates that a notable proportion of C/C++ projects encounter non-reproducibility issues stemming from build failures, primarily attributed to the absence of necessary packages. This paper introduces PACKHUNTER, a novel technique that automates the recovery of missing packages in C/C++ projects. By identifying missing files during the project's build process, PACKHUNTER can determine potentially missing packages and synthesize an installation script. Specifically, it simplifies C/C++ projects through program reduction to reduce build overhead and simulates the presence of missing files via mock build to ensure a successful build for probing missing files. Besides, PACKHUNTER leverages a sophisticated design to eliminate packages that do not contain the required missing files, effectively reducing the search space. Furthermore, PACKHUNTER introduces a greedy strategy to prioritize the packages, eventually recovering missing packages with few times of package enumeration. We have implemented PACKHUNTER as a tool and evaluated it on 30 real-world projects. The results demonstrate that PACKHUNTER can recover missing packages efficiently, achieving 26.59× speed up over the state-of-the-art approach. The effectiveness of PACKHUNTER highlights its potential to assist developers in building C/C++ artifacts and promote software reproducibility.

Index Terms—Package Management, Build System Maintenance, and Reproducibility.

1 INTRODUCTION

The reproducibility of software artifacts is one of the fundamental challenges for both industrial and academia [1]. An essential initial step in reproducing C/C++ projects is ensuring their buildability. However, a recent study [2] reports that less than 11% (65 out of 592) of C/C++ open source projects using Make or CMake from GitHub repositories can be built successfully. One primary cause of these build failures stems from the absence of the required packages, accounting for over 50% of the cases [2], [3], [4]. In this work, we refer to this problem as missing package errors.

The prevalence of missing package errors in C/C++ projects can be largely attributed to the absence of a cohesive method for managing package dependencies [5], [6]. Users resort to various ways to manage package dependencies, including project documents (e.g., `readme.md`), non-uniform scripts of package installation (e.g., `install.sh` and `prerequisite.sh`), and some package management tools (e.g., `vcpkg` [7] and `conan` [8]). Due to the lack of a widely-used standard of package management, developers would easily forget to specify the required packages, thus leading to failures when building in a new environment. When encountering missing package errors, users have to exhaustively search the missing dependency packages based on the error messages or report bugs to ask for guidance from developers, which often involves laborious manual effort when building real-world projects[1]. In this work, we concentrate on an automatic recovery of the missing packages for a given C/C++ project. More concretely, we aim to synthesize an installation script that can install all the missing packages and eventually facilitate a successful build of the project.

Unfortunately, it is a challenging task to solve the missing package recovery problem. On one hand, extracting required dependencies via statically analyzing build scripts [9], [10] is a possible solution. However, due to the complicated and diverse syntax of build commands, it is impossible to implement an omnipotent parser to extract dependencies from build commands precisely and completely, thus impeding the complete retrieval of required packages. On the other hand, dynamically capturing dependencies via monitoring the building process is another possible solution [11], [12], [13]. However, when encountering failures caused by missing packages, we must pause the building process and resume it only after successfully identifying and installing the necessary packages, which is similar to a manual trial process. Thus, this approach necessitates repetitive and iterative building, which is time-consuming and leads to efficiency concerns. Moreover, both techniques extract dependencies at the file level, requiring us to deduce the required packages from files by searching a large package repository (e.g., there are 66,475 packages in Ubuntu 22.04 repository [14]).

To efficiently recover missing packages, we propose a novel approach called PACKHUNTER, which originates from three key observations. First, only a small propor-

- *Rongxin Wu, Zhiling Huang, and Zige Tian are with School of Informatics, Xiamen University, China. E-mail: {wurongxin@xm.edu.cn, huangzhiling@stu.xmu.edu.cn, tianzige@stu.xmu.edu.cn}.*
- *Chengpeng Wang and Xiangyu Zhang are with Purdue University, USA. E-mail: {wang6590@purdue.edu, xyzhang@cs.purdue.edu}.*

Manuscript received June 12, 2024; revised September 30, 2024; accepted November 15, 2024.

1. A real-world project refers to a project that addresses practical, real-life problems and is typically implemented outside of a purely academic or theoretical context.

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

tion of source code, such as `#include` directives and `#define` directives, determines the dependencies of the project. Hence, it is not mandatory to perform a complete build for the dependency extraction. Second, the absence of symbols revealed in missing package errors, such as variables, functions, and header files, is primarily caused by the absence of specific files that should be provided by the required packages. These missing files can serve as important indicators for localizing the missing packages in the initial step. Moreover, not all the packages that include the missing files are desired by the project. If the file offered by a package does not offer any symbols used by the project, the package can be safely disregarded and not installed. Third, only when the missing files are covered by specific packages may the missing package error be resolved. By building the project after installing such packages, we can validate whether the packages are the desired ones.

Based on the above insights, we introduce PACKHUNTER with three stages. In the first stage, we employ the technique of program reduction and mocking files to facilitate a successful one-time build. During this stage, we collect all the missing files that reveal the root causes of missing package errors. In the second stage, we examine whether there exist the def-use relations between the symbols defined in a package and the ones used in the C/C++ project. This analysis enables us to effectively filter out the irrelevant packages. In the last stage, we enumerate the set of packages that covers the missing files and synthesize an installation script. Particularly, we introduce a prioritized enumeration where we maximize the number of missing files in the package selection. Owing to our designs, PACKHUNTER hits desired packages with few times of package enumeration in our pruned search space, promoting the efficiency of the installation script synthesis.

We have implemented our idea as a tool named PACKHUNTER and evaluated it using 30 popular and widely-used C/C++ open-source projects hosted in GitHub, which lack 4.67 packages on average. It is shown that PACKHUNTER finishes the overall analysis with 141.69 seconds averagely, achieving 26.59× speed up over the state-of-the-art approach. We also conduct an ablation study to demonstrate the importance of our technique designs. To sum up, the contribution of our work can be summarized as follows.

- Our research represents the pioneering effort in addressing the problem of missing package recovery specifically for C/C++ projects. By focusing on this aspect, our work contributes to enhancing the reproducibility of C/C++ software artifacts.
- To efficiently synthesize an installation script, we propose a novel approach called PACKHUNTER. This approach capitalizes on the identification of missing files to effectively reduce and explore the search space for missing packages during the synthesis process.
- We perform a comprehensive empirical evaluation to assess the effectiveness of our technical designs, showing the practical value of PACKHUNTER in promoting software reproducibility in real-world scenarios.

2 PACKHUNTER IN A NUTSHELL

In this section, we first introduce the motivation of this work (Section 2.1), and then point out the technical challenges (Section 2.2). Finally, we explain the key idea of PACKHUNTER (Section 2.3).

2.1 Empirical Study

To investigate the pervasiveness of missing package errors, we first collected 1,294 C/C++ projects from GitHub based on the following selection criteria. First, it should have achieved over 1,000 stars or forks which indicates its popularity. Second, it relies on certain build systems (e.g., `Make`, `CMake`, `Autoconf`, etc.) for building. Since it is nontrivial and time-consuming to build such a large number of projects, we decided to randomly select 80 of these projects to examine whether they suffered from build failures and missing package errors. To avoid introducing bias during the selection process, we labeled each of the 1,294 projects with an integer from 1 to 1,294. Using the sample function from Python random module [15], we randomly selected 80 projects to form the experimental subject set. It is worth noting that the sample function implements the Fisher-Yates shuffle algorithm [16], which ensures that each project has an equal probability of being chosen for the experimental subject set. Therefore, we did not introduce bias in the process of constructing the experimental subject set, and the experimental results can generally reflect the characteristics of the real-world projects.

For each selected project, we manually examine its build scripts and the instructions on how to build the project in the project documentation. Then, we set up a clean docker whose environment (e.g., OS kernel version, compiler, etc.) satisfies the requirements specified in project documents and build the project with the default configuration settings. If a project is built with failures, with the hints from the error log messages, we resort to the searching command of the system package manager (e.g., “`apt-file search`” in Ubuntu), the help from the project developers (e.g., submitting an issue report), and the online Q&A websites (e.g., `StackOverflow`) to repair the build failures manually. Based on the fixing solutions, we categorize the root causes of build failures.

Our preliminary study, which is publicly available online [17], shows that 58.75% (47 out of 80) projects suffer from build failures. We further investigate the root causes of the build failures and categorize them into four groups: missing package error (30 out of 47), build script error (7 out of 47), source code error (5 out of 47), and miscellaneous error (5 out of 47, which refers to the errors that cannot fall into the other three categories, such as incompatible compiler version and incorrect command options.). These results are consistent with the findings in the prior studies [2], [3], [4] that missing package error is the most dominant reason for the build failures.

To better understand the difficulties of resolving missing package errors, we examine the number of the missing packages that have been identified and installed during our manual repair process, as shown in Figure 1. Typically, the number of missing packages that require to be installed is

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

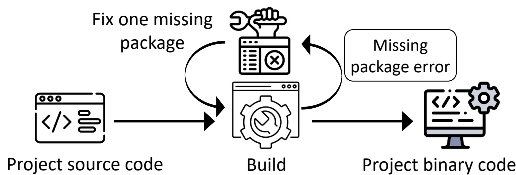

Fig. 1: The manual installation of missing packages

4.67 on average, ranging from 1 to 16. Resolving the missing package errors is non-trivial. This is because, even with the hints from error log messages, it still requires substantial effort to try numerous potential packages. Figure 2 provides an example simplified from the build failure in the open source project `guetzli` [18]. The error log message indicates that the header file "`png.h`" is required. By searching with the command "`apt-file search png.h`", there are four packages that have the potential to fix the error, but only the package `libpng-dev` is the correct one. Besides, if the name of a missing file is common, e.g., "`version.h`", "`config.h`", "`types.h`", etc., there can be hundreds of candidate packages containing the file with the same name. Even worse, the exploration of the search space expands exponentially when accounting for the permutations of different packages, thereby exacerbating the encountered challenges. In summary, manually recovering missing packages is challenging and labor-intensive.

2.2 Problems of State-of-the-Art Approaches

The key to automatically recovering the missing packages is to identify all the dependent packages required in the building process. Despite the tremendous research progress in extracting dependencies of C/C++ projects, existing methods are deficient in recovering missing packages in the scenario of resolving build failures.

One line of approaches, namely SCA (Software Composition Analysis) [9], [10], statically analyzes build scripts to extract third-party dependency libraries and is applicable to extracting missing packages. However, such static approaches are afflicted by two limitations. First, implementing a general parser to understand the syntax of build commands in various kinds of build scripts associated with different build systems is non-trivial and sometimes even impossible, thereby impeding the complete retrieval of required packages. For example, in scenarios where the build scripts incorporate commands with a syntax that is not predefined, the parser encounters difficulty in extracting the necessary dependencies from the commands. Second, in most cases, the dependencies derived from the parsing outcomes of SCA approaches primarily manifest at the file level, including header files, static/dynamic libraries, etc. Consequently, we still have to deduce the required packages from these files, thereby encountering the challenge of the large search space, as discussed in Section 2.1.

The other line of approaches, originally designed to detect build script errors [11], [12], [13], can capture dependency files via continuously monitoring the build process. Although such approaches can mitigate the generalizability issue in the static approaches, they suffer from low efficiency due to two main reasons. First, the nature of build failures necessitates the adoption of an iterative process: a build process halts whenever a package required by a build command is absent, only to be resumed upon the installation of the required package. Second, since build monitoring exclusively captures dependencies at the file level, it encounters the same challenge of inferring required packages from these files discussed in Section 2.1.

2.3 Key Idea

In this study, we also employ build monitoring to mitigate the generalizability issue associated with the implementation of build script parsers. To address the efficiency issues of build monitoring described in Section 2.2, we introduce two innovative designs.

First, instead of using an actual build which involves an iteratively suspending and resuming the process, we devise a "mock build" to accelerate the extraction of missing dependency files. Our idea is inspired by a previous study that introduced the notion of "virtual build" [13], which involves pruning the program while retaining the original dependencies in a one-time build. Although virtual build can accelerate the build process by skipping the compilation of unnecessary source code, it still cannot escape from the iterative suspending and resuming problem. Therefore, on top of virtual build, we mock the missing dependency files, such as the file `png.h` in Figure 2, so as to make the build proceed without the necessities of installing the required packages. It is important to highlight that the virtual build serves as the foundation of the mock build. Given that a virtual build exclusively retains preprocessor directives, such as `#include` directives and macro definitions, while excluding other program constructs like function bodies and global variables, there is no need for the mock files to furnish declarations or definitions for any symbols, including variables and functions. Thus, empty mocking files are adequate to ensure the successful completion of the build process.

Second, to reduce the expansive search space associated with inferring required packages from files, we propose a two-stage analysis, comprising package filtering and prioritization, to streamline the deduction of the required packages. Intuitively, the required packages should contain the missing dependency files and symbols. Thus, during the package filtering stage, we utilize the missing files identified in the mock build process to pinpoint a set of candidate packages that include these files. Meanwhile, we compare symbols used in the project code with those defined in the missing files provided by the candidate packages, which facilitates filtering irrelevant packages. During the package prioritization phase, we assign the packages covering more missing files with higher priorities, based on which we approach the task of selecting packages containing all missing files. With the greedy strategy, we are more likely to reach a set of packages that resolves the missing package errors with few times of package enumeration.

Roadmap. In the upcoming sections, we will first formalize the problem (Section 3) and delve into the technical intricacies of PackHunter (Section 4). The implementation and evaluation of PackHunter will demonstrate the effectiveness of our approach (Section 4.4 and Section 5).

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

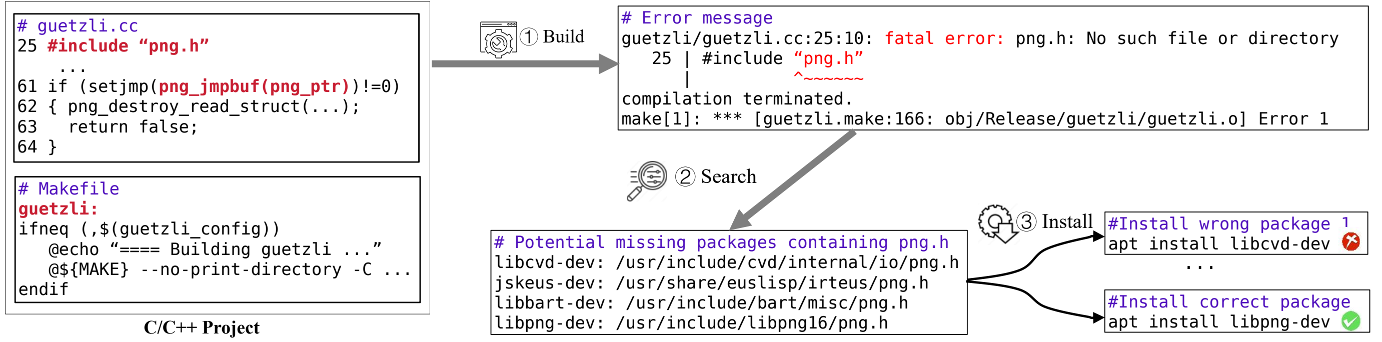

Fig. 2: A motivating example of recovering missing packages simplified from the open source project `guetzli`.

$$
\begin{aligned}
\text{Project} \quad \mathcal{P} &:= (\mathcal{F}_h,\ \mathcal{F}_s,\ \mathcal{F}_{bs}) \\
\text{Header File} \quad f_h &:= d\ f_h \mid c_s\ f_h \mid \varepsilon \\
\text{Source File} \quad f_s &:= d\ f_s \mid c_f\ f_s \mid \varepsilon \\
\text{Build Script} \quad f_{bs} &:= (\text{target} : (\text{pre})^*\ \text{recipe})^+ \\
\text{Preproc Directive} \quad d &:= \text{\#include str} \mid \text{\#define str str} \mid \cdots \\
\text{Func Def} \quad c_f &:= c_s\ \{(\text{stmt};)^*\} \\
\text{Func Signature} \quad c_s &:= \tau\ \ \text{fname}\ (\tau\ \ \text{para})^*
\end{aligned}
$$

Fig. 3: The program syntax

3 Problem Formulation

This section first formulates the program syntax (Section 3.1) and then introduces the missing package error (Section 3.2). Lastly, we offer the formal statement of the missing package recovery problem (Section 3.3).

3.1 Program Syntax

Following prior research [13], we formulate C/C++ projects with the syntax shown in Figure 3. In essence, a project encompasses header files, source files, and build scripts. Specifically, both the header and source files may incorporate various preprocessor directives, including those that involve the inclusion of specific header files and the definition of macros. Function signatures and definitions are presented in the header files and source files, respectively. Notably, a function body can either be empty or consist of multiple statements, such as assignments and function calls. Without the loss of generality, we assume that each function defined in the source file is declared in the corresponding header file. Finally, a build script is a collection of build target dependencies, each of which is associated with a set of instructions as the recipe.

3.2 Missing Package Error

In our work, we target a prevalent class of build errors, namely missing package errors. Formally, we formulate it as follows.

Definition 1. (Missing Package Error) Let $\mathcal{P}$ be a C/C++ project. A missing package error in $\mathcal{P}$ is the absence of a specific package required for the successful build of a specific target t within $\mathcal{P}$.

Basically, a missing package error is manifested as the lack of preprocessor directives, functions, and files, which are expected to be offered by the missing package. When a build process yields a missing package error, its error message can indicate the root cause, guiding the developers to localize the missing package for a fix.

Example 1. As depicted in Figure 2, an error is encountered while building the target `guetzli`, indicating that the file `png.h` is not found. Thus, the source files that include `png.h` as a header file, such as `guetzli.cc`, cannot utilize the symbols defined within that file. For instance, it becomes invalid to invoke the function `png_jmpbuf` in the source file `guetzli.cc` at Line 61. This absence of the package providing `png.h` leads to a missing package error.

As shown in Example 1, the essential cause of a missing package error is the absence of a specific file. Specifically, the symbols defined in the missing file, such as preprocessor directives and functions, become inaccessible to the current C/C++ project. Existing efforts have indicated that missing package errors are widespread in real-world C/C++ projects. In particular, it has been reported that over 50% of build failures of C/C++ projects are caused by the missing packages [2], [3], [4]. The difficulty of identifying and installing missing packages presents a major hurdle for programmers aiming to create an executable artifact of their own systems.

3.3 Problem Statement

To mitigate the burden of programmers in building C/C++ projects, we target the problem of automatically recovering missing packages in this paper. Before we state our problem explicitly, we first introduce two important preliminaries, namely *package installation strategy* and *installation script*.

Definition 2. (Package Installation Strategy) A package installation strategy I is a function that maps a package to its installation commands. Particularly, $dom(I)$ is named as the package base, indicating a finite set of packages that can be potentially used.

As defined in Definition 2, the package installation strategy I is responsible for specifying a finite set of packages as the package base, i.e., $dom(I)$, along with providing guidance on their installation. For instance, we can focus on packages that are maintained by a Linux package manager,

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

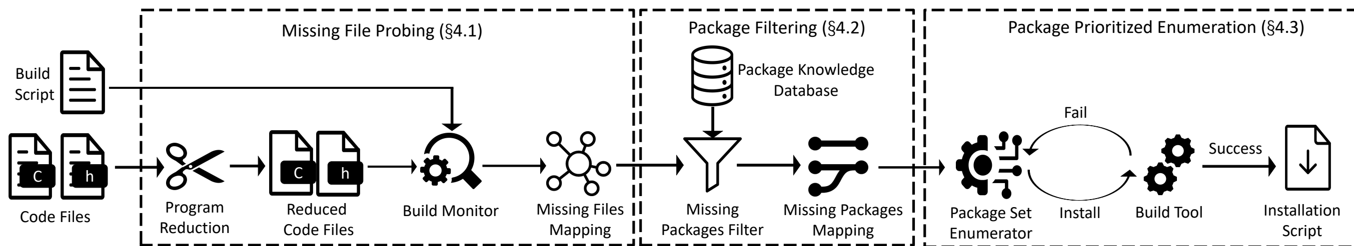

Fig. 4: The workflow of PACKHUNTER

as the installation commands for such packages are readily available. With the guidance of the package installation strategy, programmers can localize missing packages and install them by executing the corresponding installation commands. Essentially, programmers need to specify an *installation script*, which is defined as follows, to recover missing packages.

Definition 3. (Installation Script) Given a C/C++ project $\mathcal{P}$, an installation script f_{ps} is a sequence of commands installing packages such that $\mathcal{P}$ can be successfully built after the execution of f_{ps}.

Example 2. Consider the missing package error in Figure 2. An installation script should contain the installation command of the package `libpng-dev`. After running the installation script, the missing package, i.e., `libpng-dev`, can supply the missing file `png.h`, which resolves the missing package error in the project.

As illustrated at the end of Section 3.2, manually specifying an installation script can be a labor-intensive task, particularly when dealing with a large package base and multiple missing packages. To alleviate the laborious effort involved, we formulate ***missing package recovery problem*** as follows and attempt to propose an effective and efficient solution, which would greatly reduce the burden on individuals engaged in the development of C/C++ projects when confronted with missing packages.

> Given a C/C++ project $\mathcal{P}$ and a package installation strategy I, synthesize an installation script f_{ps} automatically.

Solving the missing package recovery problem is nontrivial. The challenges mainly arise from two aspects. First, extracting dependency files via monitoring the repetitive and iterative building process, i.e., pausing the build for a failure and resuming it upon a trial of package installation, is time-consuming, leading to efficiency concerns. Second, deducing missing packages from dependency files requires to search from a large package repository and the large space of candidate packages would exacerbate the efficiency issues. In Section 4, we will demonstrate how to address the above challenges and illustrate the details of our approach.

4 APPROACH

In this section, we introduce our approach, named PACKHUNTER, to resolve the missing package recovery problem. The overall workflow of PACKHUNTER is illustrated in Figure 4, consisting of three key stages: missing file probing (Section 4.1), package filtering (Section 4.2), and package prioritized enumeration (Section 4.3).

- First, the missing file probing involves simplifying C/C++ projects through program reduction to facilitate efficient identification of missing files during a one-time build (Section 4.1). This allows us to quickly locate the root causes of missing package errors.
- Second, we utilize the def-use relations between the symbols in the packages and the C/C++ projects to filter irrelevant packages (Section 4.2), which significantly reduces the search space for the subsequent installation script synthesis.
- Third, we prioritize the sets of packages that can resolve all the missing files in the enumeration and validate them by building the project after installing the packages (Section 4.3). A successful build indicates that the desired installation script has been generated.

In the following subsections, we provide a detailed explanation of each of these stages.

4.1 Missing File Probing

As illustrated in Section 3.2, the occurrence of missing package errors can be attributed to the absence of specific files that are offered by required but missing packages. This observation suggests that the missing files can serve as valuable indicators of the missing packages, allowing us to filter out the irrelevant package candidates effectively. Motivated by this insight, we propose probing missing files as our first step.

It is evident that probing missing files necessitates building the project for dynamic monitoring [11], [19], which enables us to gather file I/O information. However, we have to notice that a single build process can result in considerable overhead, especially considering the need to repeatedly try different packages and rebuild the project. To achieve efficient missing file probing, we propose two effective strategies, namely program reduction and mock build, which aim to minimize the overhead associated with a single build and reduce the number of build iterations, respectively. The formulation of these strategies is presented in Algorithm 1.

- ***Program Reduction.*** Inspired by prior study [13], gathering file I/O information does not necessarily require an actual build of the whole project. Instead, by trimming a project via the program reduction proposed in [13], we can obtain the same file I/O information as the one derived from the original project. We formulate it as the function `reduceProgram` at Line 2 in Algorithm 1. Note that the

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

Algorithm 1: Probing Missing Files

```
Input: P := (F_h, F_s, F_bs): A C/C++ project;
Output: M_f: A mapping from targets to missing file sets;
1 /* Remove statements and non-main functions. */
2 P' ← reduceProgram(P);
3 T ← getTarget(P');
4 foreach t ∈ T do
5     /* Create and gather missing files */
6     M_f(t) ← mockBuild(t);
7 return M_f;
```

successful build of a target only depends on the existence of its main function, so we eliminate non-main functions from the project and remove all the statements from the body of each main function. In particular, all the preprocessor directives remain in the project. As demonstrated in the previous study [13], such program reduction does not change the file IO information during the build, while the build time can significantly decrease. In this way, we can accelerate the missing file probing by reducing the overhead of a single build.

- ***Mock Build.*** To avoid multiple iterations of project builds, we adopt a mocking strategy to permit the build process to continue in the absence of necessary files. Specifically, when building each target t, we dynamically monitor the file IO operations and create empty files to mock the files that need to be accessed but missing. The mock build process is formulated by the function `mockBuild` at Line 6 in Algorithm 1. After the mock build, we can establish the mapping from a build target to a set of missing files, which serves as an ingredient for discovering missing packages in the subsequent stages (Section 4.2 and Section 4.3). Importantly, the mock build demands building the project only one time. Any build errors caused by missing files are skipped, avoiding unnecessary rebuild of the project with extra overhead.

It is important to note that simply creating an empty file for the missing one is sufficient to ensure a successful build. By leveraging program reduction, all the symbols imported from the packages are effectively removed, resulting in the decoupling of the targeted project from the files in its dependent packages. As a result, program reduction not only reduces the overhead associated with a single build but also enables the monitoring of file I/O information through a single round of mock build.

Example 3. In Figure 2, the file `png.h` is missing in the build of the target `guetzli`. Based on the mock build, we can monitor the access operation upon the missing file and create a corresponding empty file for `png.h`. The non-main functions and the statements of main functions are removed in the program reduction, such as the ones in the file `guetzli.cc`. With the assistance of program reduction, the mock build can successfully collect all the missing files efficiently, like the file `png.h`, in only one time of the build.

Algorithm 2: Filtering Packages

```
Input: P := (F_h, F_s, F_bs): A C/C++ project;
       I: Package installation strategy;
       M_f: A mapping from targets to missing file sets;
Output: M_p: A mapping from missing files to package sets;
1  M_p ← [f ↦ ∅ | f ∈ ⋃_{f' ∈ dom(M_f)} M_f(f')];
2  foreach t ∈ dom(M_f) do
3      foreach f' ∈ M_f(t) do
4          S'_p ← getPotentialPkg(f');
5          F̂ ← getAccessedFiles(t) ∩ F_h ∩ F_s;
6          foreach f ∈ F̂ do
7              /* Collect used symbols */
8              S_use ← getUsedSym(f);
9              foreach pkg ∈ S'_p do
10                 /* Collect defined symbols */
11                 S_def ← getDefinedSym(f', pkg);
12                 /* Compare defined/used symbols */
13                 if S_def ∩ S_use ≠ ∅ then
14                     M_p(f') ← M_p(f') ∪ {pkg};
15 return M_p;
```

4.2 Package Filtering

Although missing file probing can effectively identify all missing files in a project, we still need to address another challenge: A single missing file can be provided by multiple different packages. This phenomenon can significantly inflate the search space when synthesizing an installation script. To filter out irrelevant packages, we rely on an important insight that the symbol def-use relation enables us to filter infeasible packages in a light-weighted fashion. Specifically, if a package pkg offers a missing file but the file does not define any symbols that are used in the source/header files within the target C/C++ project, the file offered by the package pkg can be decoupled from the project, indicating that the package pkg cannot support necessary ingredients for the build process.

Based on the above insights, we propose to filter irrelevant packages, as formalized in Algorithm 2. Initially, it takes three inputs: a C/C++ project $\mathcal{P}$, a package installation strategy I, and a mapping M_f that associates targets with sets of missing files obtained from Algorithm 1. When dealing with each missing file f', the function `getPotentialPkg` at Line 4 retrieves all the packages that provide a file with the same name as f'. Based on the file I/O information during a mock build, we collect all the accessed source/header files in the project $\mathcal{P}$ at Line 5 and extract all the used symbols in the accessed files with a parsing-based static analysis, forming the set S_{use} at Line 8. Meanwhile, we consider each package pkg that provides the file f' and gather all the symbols defined in the corresponding file in a similar fashion, forming the set S_{def} at Line 11. Only when S_{use} and S_{def} are not disjoint can the package pkg be considered for use in the C/C++ project (Line 13–14). Ultimately, we obtain a mapping M_p from missing files to sets of packages, which defines the search space for the synthesis of the installation script, which will be demonstrated in Section 4.3. Owing to package filtering design, we can safely discard infeasible packages, effectively

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

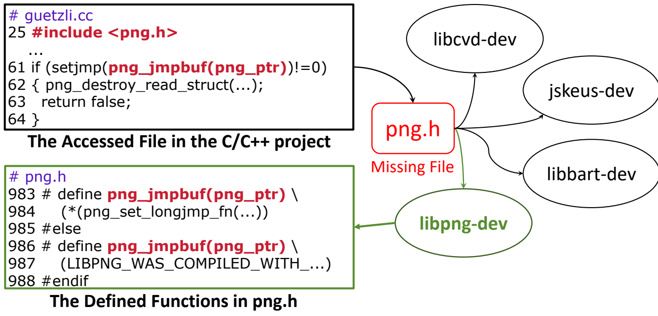

Fig. 5: An example of package filtering

reducing the search space for the synthesis process.

Lastly, it is important to note that there is no need to perform additional mock builds to obtain the accessed source/header files at Line 5 in Algorithm 2. When probing for missing files in Algorithm 1, we can simultaneously collect the information about the accessed files through an on-the-fly mock build. In the paper, we have intentionally separated the process of gathering missing files from the demonstration to simplify the presentation.

Example 4. In Figure 5, the file `png.h` is missing and can be provided by multiple packages. Upon analyzing the contents of the offered `png.h` files and the accessed files within the C/C++ project, we can discover that the `png.h` file in `libpng-dev` defines the symbols utilized by the C/C++ project, such as the function `png_jmpbuf`. Hence, the package `libpng-dev` is a potential missing package. Regarding other packages, they do not provide any symbols used by the project. Consequently, we can narrow down the list of missing packages for subsequent installation script synthesis.

4.3 Package Prioritized Enumeration

Benefiting from the previous two stages demonstrated in Section 4.1 and Section 4.2, we are able to establish a search space of missing packages for the targeted C/C++ project. In essence, we need to select a set of packages from the search space that can cover all the missing files identified during the mock build process. To further accelerate the enumeration of missing packages, we propose the package prioritization during the enumeration, which is based on two intuitions. First, if a missing file f' is provided by a unique package pkg, we have to install pkg in our installation script. Otherwise, the file f' would still be missing after the execution of the installation script. Second, if a package can offer multiple missing files, it is very likely to be the necessary one for the C/C++ project. Therefore, we conduct the prioritized enumeration to validate whether the enumerated packages facilitate the success of a build.

We formalize our idea in Algorithm 3. Technically, it begins by identifying the packages that uniquely offer specific missing files (Line 2-6). The remaining missing files, represented by the set S_f at Line 6, can be covered by more than one package. Consequently, we generate all possible choices of packages that can cover all the files in S_f using the `getCoverPkgSet` function and form the set Γ at Line 8.

Algorithm 3: Synthesizing Installation Script via Package Prioritized Enumeration

Input: $\mathcal{P} := (\mathcal{F}_h,\ \mathcal{F}_s, \mathcal{F}_{bs})$: A C/C++ project;
I: Package installation strategy;
M_p: A mapping from missing files to package sets;
Output: f_{ps}: An installation script;

```
1  f_ps ← ∅; S_f ← dom(M_p);
2  foreach f' ∈ dom(M_p) do
3      /* Select the unique package offering f' */
4      if |M_p(f')| = 1 then
5          f_ps ← f_ps ∪ {I(pkg) | pkg ∈ M_p(f')};
6          S_f ← S_f \ {f'};
7  /* Compute package sets covering missing files */
8  Γ ← getCoverPkgSet(S_f, M_p);
9  foreach S_p ∈ Γ do
10     w(S_p) ← des_sort([|M_p^{-1}(pkg) ∩ S_f| | pkg ∈ S_p]);
11 while Γ is not empty do
12     /* Peek a package set according to the descending order upon w. */
13     S_p ← peek(Γ, w); Γ ← Γ \ {S_p};
14     f_ps ← f_ps ∪ {I(pkg) | pkg ∈ S_p};
15     if installAndBuild(f_ps, P) is success then
16         return f_ps;
17 return f_ps;
```

Then we compute the weights of each package set by counting the missing files covered by each package and sorting them descendingly, which is formulated at Lines 9 and 10. Based on the computed weights, we enumerate the package sets in Γ by peeking a package set that contains packages covering as many missing files as possible. Furthermore, we synthesize the installation script f_{ps} based on the package installation strategy I (Line 14) and build the target C/C++ project $\mathcal{P}$ to determine whether the installation script f_{ps} successfully resolves the missing packages. If the build is successful, we directly return f_{ps} as one feasible solution for our problem. Otherwise, we continue to examine other package sets in Γ until a successful build is achieved. To ensure the integrity of subsequent iterations, we uninstall the newly installed packages if the installation script fails to support a successful build.

Note that the build at Line 15 represents the actual build rather than the mock build. In the mock build mentioned in Section 4.1, the project only contains the main functions with empty bodies after reduction, which can not validate the available symbols that should be provided by the packages. Considering the time overhead of performing an actual build, our design of the package filtering in Section 4.2 and the package prioritized enumeration in Algorithm 3 can significantly reduce the time cost of synthesizing the installation script. This is because the desired packages are more likely to be selected early in the enumeration process, thereby limiting the number of actual builds required.

Example 5. As shown in Figure 6, the straight arrows indicate the mapping from missing files to potential missing packages obtained in Algorithm 2. Note that the missing file `libfreetype.so` is only offered by

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

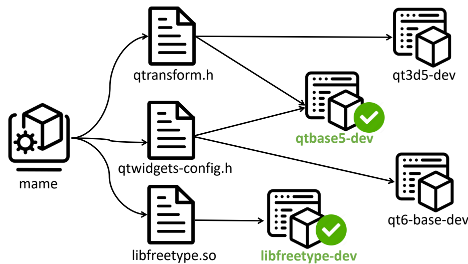

Fig. 6: A simplified example of synthesizing an installation script with package prioritized enumeration for `mame`

TABLE 1: Basic Information of Projects. **#Files, #Lines, and #Pkgs** represent the number of source files, lines of code, and the number of the missing packages. **Build** represents the time of clean build.

Project	#Files	#Lines (KLoC)	#Pkgs	Build (sec)	Build System
dump1090	3	2.78	1	0.80	Make
guetzli	26	2.02	1	4.14	Make
shairport	35	6.29	5	0.49	Make
coturn	85	46.07	8	4.88	Autoconf
clib	92	17.57	1	0.37	Make
tig	92	39.46	1	0.90	Make
paho-mqtt	102	54.22	1	5.92	Make
glfw	103	88.40	4	2.14	CMake
xmr-stak	144	36.11	3	32.47	CMake
tmux	191	75.19	2	1.82	Autoconf
box2d	240	99.89	5	2.98	CMake
workflow	245	41.57	1	4.30	CMake
Bonzomatic	260	203.86	7	3.14	CMake
raylib	293	335.77	5	10.17	CMake
TheAlgorithms/C	431	35.18	2	0.55	CMake
SFML	544	157.20	9	14.14	CMake
g2o	572	67.78	1	79.29	CMake
LearnOpenGL	579	157.18	8	5.13	CMake
xmrig	788	243.61	3	26.91	CMake
ZLMediaKit	1,031	176.85	1	35.82	CMake
minetest	1,036	309.71	16	203.56	CMake
libgit2	1,115	309.03	1	3.26	CMake
OpenRCT2	1,207	721.72	12	42.36	CMake
osrm-backend	1,680	332.87	6	104.33	CMake
postgres	2,342	1,393.94	2	64.17	Autoconf
stk-code	2,388	814.13	12	32.68	CMake
pcl	2,577	740.34	3	606.12	CMake
poco	2,898	944.03	1	80.02	CMake
vowpal_wabbit	4,783	1,005.60	3	88.81	CMake
mame	19,039	9,781.75	15	767.50	Make
Average			**4.67**	**74.31**	

the package `libfreetype-dev`. Hence, the installation script should contain the installation command of the package `libfreetype-dev`. Meanwhile, the missing files `qtransform.h` and `qtwidgets-config.h` can be offered by the package `qtbase5-dev`, so we enumerate the packages {`qtbase5-dev`} with the higher priority than {`qt3d5-dev`, `qt6-base-dev`}. Therefore, we pick the two packages `libfreetype-dev` and `qtbase5-dev` at the beginning of the prioritized enumeration. After validating them with an actual build, we can successfully synthesize an installation script that installs `libfreetype-dev` and `qtbase5-dev`.

4.4 Implementation

We have implemented our approach PACKHUNTER as a tool, which is designed to recover missing packages for C/C++ projects. Our implementation primarily focuses on APT [20], the package management system in Debian-based Linux distributions like Ubuntu. To establish our package base, we downloaded all 66,475 packages managed by APT from its official website. Benefiting from our approach, the process of manually identifying missing packages is significantly simplified, eliminating the need for laborious and time-consuming efforts.

To facilitate the detection of missing files, we have implemented a program reduction technique inspired by VIRTUALBUILD [13]. However, unlike VIRTUALBUILD, which relies on LD_PRELOAD for file IO monitoring, we utilize ptrace for more precise file IO monitoring and combine it with the Berkeley Packet Filter (BPF) to capture only the relevant system calls. This implementation design ensures our monitoring requirements are met while minimizing the performance overhead associated with surveillance.

To filter irrelevant packages, we utilize the ELF (Executable and Linkable Format) analysis tool READELF [21] and the parsing-based static analysis tool srcML [22] for symbol extraction. Specifically, during the preprocessing phase, for all packages, we leverage READELF to examine their static and dynamic library files, and utilize SRCML to analyze their header files. This process allows us to extract the variables and function names defined by these packages, with the extracted data stored in JSON files that occupy less than 30MB of disk space. For symbol analysis in the target C/C++ project, we adopt a demand-driven approach, where only the header and source files accessed during the mock build are analyzed using SRCML to identify the external variables and functions used, resulting in improved efficiency in the step of package filtering, as it means that we do not need to analyze all files within the project.

5 EVALUATION

We evaluate the effectiveness of PACKHUNTER by investigating the following research questions:

- **RQ1**: How effectively and efficiently does PACKHUNTER recover missing packages for real-world C/C++ projects?
- **RQ2**: How does PACKHUNTER compare against the considered baselines that could be used for recovering missing packages?
- **RQ3**: How important are package filtering and package prioritized enumeration in PACKHUNTER?

5.1 Experimental Setup

Dataset. We choose the projects with missing package errors that are investigated in our empirical study (Section 2.1). These projects are managed in different building systems, such as `Make`, `CMake` and `Autoconf`. Table 1 shows the basic information of these projects. We initially confirm through manual verification that these projects would fail to build due to missing essential packages when being compiled and built directly. The absence of the packages enables us to evaluate the effectiveness of PACKHUNTER in the current build environment.

Baselines and Ablations. To answer **RQ1** and **RQ2**, we evaluate PACKHUNTER on our dataset and compare it with CCSCANNER [9]. Specifically, CCSCANNER originally targets the identification of dependencies in C/C++ projects by statically analyzing build scripts and the software bill of

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

TABLE 2: The results of PACKHUNTER and baselines. Speedup◁ represents the speedup of **PackHunter** over **CCScanner**. Speedup⋆ represents the speedup of **PackHunter** over **TryInstall**. **NA** indicates that CCSCANNER fails to recover missing packages for specific subjects.

Project	PackHunter					CCScanner		TryInstall	
	Probe (sec)	Filter (sec)	Enumerate (sec)	Total (sec)	Total/Build	Total (sec)	Speedup◁ (×)	Total (sec)	Speedup⋆ (×)
dump1090	0.14	0.03	3.62	**3.79**	4.71	NA	NA	8.66	2.29
guetzli	0.84	0.18	7.59	**8.62**	2.08	11.67	1.35	179.28	20.80
shairport	0.28	0.06	16.03	**16.38**	33.57	NA	NA	863.57	52.72
coturn	6.08	0.55	17.64	**24.28**	4.97	130.61	5.38	1,180.31	48.62
clib	0.39	0.15	8.96	**9.50**	25.89	2,410.52	253.73	13.93	1.47
tig	1.04	0.83	2.51	**4.38**	4.85	335.76	76.70	8.55	1.95
paho-mqtt	3.62	0.86	10.32	**14.80**	2.50	97.39	6.58	474.48	32.05
glfw	7.74	0.50	19.44	**27.68**	12.92	306.38	11.07	112.39	4.06
xmr-stak	3.86	0.68	52.96	**57.50**	1.77	NA	NA	416.06	7.24
tmux	4.12	0.01	5.92	**10.04**	5.51	NA	NA	929.16	92.50
box2d	8.24	1.68	7.32	**17.24**	5.79	NA	NA	73.67	4.27
workflow	2.14	1.10	9.69	**12.92**	3.01	207.11	16.02	70.49	5.45
Bonzomatic	7.89	1.08	12.91	**21.89**	6.97	NA	NA	132.56	6.06
raylib	11.86	1.56	26.12	**39.54**	3.89	NA	NA	243.20	6.15
TheAlgorithms/C	14.52	1.38	6.36	**22.26**	40.47	383.62	17.23	60.74	2.73
SFML	2.24	0.30	22.63	**25.17**	1.78	351.56	13.97	398.96	15.85
g2o	32.03	1.68	82.23	**115.94**	1.46	182.05	1.57	748.02	6.45
LearnOpenGL	23.33	1.04	24.34	**48.71**	9.51	NA	NA	295.41	6.06
xmrig	7.69	2.58	35.97	**46.24**	1.72	186.47	4.03	179.31	3.88
ZLMediaKit	11.11	1.32	39.59	**52.02**	1.45	179.53	3.45	226.89	4.36
minetest	10.18	2.49	276.31	**288.98**	1.42	NA	NA	6,188.24	21.41
libgit2	13.61	11.40	8.33	**33.34**	10.23	358.10	10.74	304.57	9.14
OpenRCT2	35.00	28.02	67.21	**130.23**	3.07	5,977.07	45.90	663.05	5.09
osrm-backend	3.57	1.88	276.68	**282.13**	2.70	NA	NA	1,739.44	6.17
postgres	22.26	0.17	70.89	**93.32**	1.45	NA	NA	166.70	1.79
stk-code	18.24	7.07	50.26	**75.57**	2.31	NA	NA	1,786.43	23.64
pcl	94.92	2.56	654.55	**752.03**	1.24	2,188.31	2.91	1,694.69	2.25
poco	14.26	22.72	87.41	**124.40**	1.55	584.43	4.70	412.86	3.32
vowpal_wabbit	58.74	6.25	134.47	**199.45**	2.25	199.99	1.00	744.35	3.73
mame	858.09	2.69	831.47	**1,692.25**	2.20	3,780.95	2.23	21,775.44	12.87
Average	**42.60**	**3.43**	**95.66**	**141.69**	**6.78**	**992.86**	**26.59**	**1403.05**	**13.81**

materials. We adapt CCSCANNER to achieve the identification of potential missing packages and synthesize an installation script via enumeration. Also, we adapt existing build monitoring methodologies [11], [12], [13] to detect missing files. Subsequently, we adhere to developers' convention of searching for packages, i.e., deriving missing packages from missing files using the "apt-file search" command, which forms a baseline approach termed TRYINSTALL. Concretely, we enumerate all the packages offering the missing files to fix each missing package error and iterate such a process until all the missing package errors are fixed. To answer **RQ3**, we remove the design of package filtering, which forms an ablation named PACKHUNTER-NOPF, and do not prioritize the packages in the enumeration, which induces an ablation named PACKHUNTER-NOPPE. The comparison with the above baselines and ablations can effectively demonstrate the superiority of PACKHUNTER and quantify the benefit of each technique design.

Environment. Each group of experiments is conducted on a computer running Ubuntu 22.04 LTS system, equipped with an Intel(R) Xeon(R) Gold 6230R CPU @ 2.10GHz forty-core processor and 512GB of physical memory.

5.2 Effectiveness and Efficiency

Setting and Metrics: To quantify the effectiveness and efficiency of PACKHUNTER, we evaluate its performance on the experimental subjects to determine whether it can successfully recover missing packages. Additionally, we measure the total time cost of PACKHUNTER, as well as the time overhead associated with different stages of the process. To illustrate the additional overhead incurred by PACKHUNTER, we calculate the ratio of its total time cost to that of a successful clean build.

As shown by the column **PackHunter** in Table 2, PACKHUNTER successfully synthesizes the installation scripts for all the C/C++ projects to recover the missing packages. The time cost ranges from 3.79 seconds to 1,692.25 seconds, and the average time cost is 141.69 seconds. The average ratio of the time cost of PACKHUNTER over the time of the actual build upon the corresponding C/C++ projects is only 6.78, indicating the low overhead of our approach in assisting the build process in real-world scenarios. In particular, the project `mame` has around 10 MLoC, and its actual build is quite costly, consuming 767.50 seconds in total. What's even worse is that the number of missing packages of the project `mame` reaches 15, which makes it quite challenging to fix missing package errors manually. It is worth noting that all the projects except for the project `mame` can be processed within 13 minutes. The high efficiency demonstrates the practical value of PACKHUNTER in analyzing real-world C/C++ projects, especially the ones on a large scale.

In the sub-columns of **PackHunter** of Table 2, we display the time cost of each stage of PACKHUNTER. First,

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

the average time cost of the missing file probing is 42.60 seconds, while the overheads can vary greatly among different projects. When a project contains more lines and build targets, the missing file probing would take more time to process each file and build the targets accordingly. However, the average ratio of the time cost of missing file probing over the time of the actual build is only 0.57, indicating that our program reduction and mock build can significantly reduce the overhead of the missing file probing. Second, the average cost of package filtering is only 3.43 seconds. The number of missing files, packages offering the missing files, and the accessed header/source files in the projects can affect the overhead in this stage as such files determine the scope of parsing-based static analysis. Remarkably, PACKHUNTER can finish the package filtering in only a few seconds for most of the projects. Third, the package prioritized enumeration takes the most significant proportion of time cost among the three stages, consuming 95.66 seconds on average. As demonstrated in Section 4.3, it has to validate the enumerated packages with an actual build, which introduces significant time overhead. Fortunately, our designs of the package filtering and package prioritized enumeration enable PACKHUNTER to find the missing packages quickly.

> **Answer to RQ1:** PACKHUNTER successfully recovers missing packages for all experimental subjects, with an average time cost of 141.69 seconds, introducing only 5.78 times more overhead than a clean build.

5.3 Comparisons with Baselines

Setting and Metrics: Similar to Section 5.2, we investigate whether the two baselines, namely CCSCANNER and TRYINSTALL, can recover missing packages successfully. To demonstrate the superiority of our approach in terms of efficiency, we also measure the time costs of the two baselines and further compute the speedups of PACKHUNTER compared to the two baselines.

As demonstrated by the column **CCScanner** in Table 2, CCSCANNER successfully recovers the missing packages for only 18 projects, with the time cost ranging from 11.67 to 5,977.07 seconds, averaging 26.59 times that of PACKHUNTER. Notably, for the project `clib`, which only has one missing package, CCSCANNER spends 2,410.52 seconds recovering the package. The reason is that CCSCANNER only identifies partial files and package abbreviations needed by the project, such as `libm.so` and "`ssl`". However, there are dozens of packages that could provide `libm.so`, and numerous packages include "`ssl`" in their names. The inability of the static analysis to gather complete information for filtering candidate packages implies that it has to traverse a vast space of potential packages in an attempt to fix the project's missing packages. Additionally, CCSCANNER fails to repair 12 other projects. The failure can be attributed to the difficulty in accurately parsing all build scripts with static analysis, given their complexity and flexibility.

For example, as shown in the makefile in Listing 1 for project `dump1090`, CCSCANNER cannot identify any information related to the missing package `librtlsdr-dev` from Line 1 to Line 2, thus it can never succeed in recovering the missing packages for the project. The inability to identify even a single missing package can lead to the failure of the project's compilation and build process.

```
# Makefile for dump1090
1 CFLAGS?=-O2 -g -Wall -W $(shell pkg-config --cflags librtlsdr)
2 LDLIBS+=$(shell pkg-config --libs librtlsdr) -lpthread -lm
3 CC?=gcc
4 %.o: %.c
5     $(CC) $(CFLAGS) -c $<
```

Listing 1: A failure case of CCSCANNER

As outlined in the column **TryInstall** in Table 2, TRYINSTALL can recover the missing packages for all projects, demonstrating the effectiveness of starting with the project's missing files to guide the recovery of missing packages. However, the time cost of TRYINSTALL is far from satisfactory. It is 13.81 times slower than PACKHUNTER on average. Particularly, it takes 21,775.44 seconds to recover missing packages for the largest project `mame`. The key reason for its low efficiency is that TRYINSTALL cannot obtain all missing files through a single mock build. The time overhead incurred from each missing package error can be aggregated. Besides, the lack of package filtering makes TRYINSTALL validate each potential package with more attempts than PACKHUNTER.

> **Answer to RQ2:** CCSCANNER and TRYINSTALL successfully recover 18 and 30 out of 30 experimental subjects, respectively, incurring average time overheads of 25.59 and 12.81 times more than PACKHUNTER.

5.4 Ablation Study

Setting and Metrics: We conduct the ablation study to evaluate the two ablations, namely PACKHUNTER-NOPF and PACKHUNTER-NOPPE, and measure their time costs of recovering missing packages upon the experimental subjects. Particularly, we count the numbers of missing package candidates before and after the package filtering in the ablation study. Considering potentially huge time cost of analyzing large projects, we set ten hours as the time budget for each project.

As shown in Table 3, PACKHUNTER-NOPF fails to generate the installation script for six projects within the time budget due to the incredibly large search space. On average, PACKHUNTER-NOPF has to consider 94.43 packages as candidates in the package prioritized enumeration, while PACKHUNTER only needs to take 8.10 packages into account. Benefiting from the pruned search space, PACKHUNTER achieves a 2.52× speedup on average over PACKHUNTER-NOPF. For the largest project, i.e., the project `mame`, PACKHUNTER saves 1424.97 (= 3117.22-1692.25) seconds compared with PACKHUNTER-NOPF, as it only needs to select 28 packages instead of 45 ones.

Owing to our package filtering design, the missing files in 26 out of 30 projects can be uniquely offered by specific packages, and thus, we can deterministically recover the packages without any enumeration. For the rest four projects, including `minetest`, `OpenRCT2`, `stk-code`, and `mame`, we conduct the comparison between PACKHUNTER and PACKHUNTER-NOPPE as shown in Figure 7. Basically, PACKHUNTER-NOPPE analyzes the projects 2.11

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

TABLE 3: The statistics of PACKHUNTER-NOPF

Project	PackHunter		PackHunter-NoPF		
	#Pkgs	Total (sec)	#Pkgs	Total (sec)	Speedup (×)
dump1090	1	3.79	1	6.57	1.73
guetzli	1	8.62	8	32.14	3.73
shairport	5	16.38	364	36.18	2.21
coturn	8	24.28	208	43.42	1.79
clib	1	9.50	79	21.14	2.23
tig	1	4.38	3	8.27	1.89
paho-mqtt	1	14.80	186	27.92	1.89
glfw	4	27.68	21	55.65	2.01
xmr-stak	3	57.50	18	176.56	3.07
tmux	2	10.04	174	14.38	1.43
box2d	5	17.24	12	39.09	2.27
workflow	1	12.92	198	17.87	1.38
Bonzomatic	7	21.89	24	75.74	3.46
raylib	5	39.54	37	NA	NA
TheAlgorithms/C	2	22.26	4	30.73	1.38
SFML	9	25.17	101	NA	NA
g2o	1	115.94	219	271.43	2.34
LearnOpenGL	8	48.71	27	191.92	3.94
xmrig	3	46.24	23	101.05	2.19
ZLMediaKit	1	52.02	72	412.65	7.93
minetest	23	288.98	40	NA	NA
libgit2	1	33.34	134	42.16	1.26
OpenRCT2	90	130.23	393	541.01	4.15
osrm-backend	6	282.13	57	NA	NA
postgres	2	93.32	8	187.88	2.01
stk-code	17	75.57	62	NA	NA
pcl	3	752.03	233	NA	NA
poco	1	124.40	84	205.45	1.65
vowpal_wabbit	3	199.45	8	526.55	2.64
mame	28	1692.25	45	3117.22	1.84
Average	**8.10**	**141.69**	**94.43**	**257.62**	**2.52**

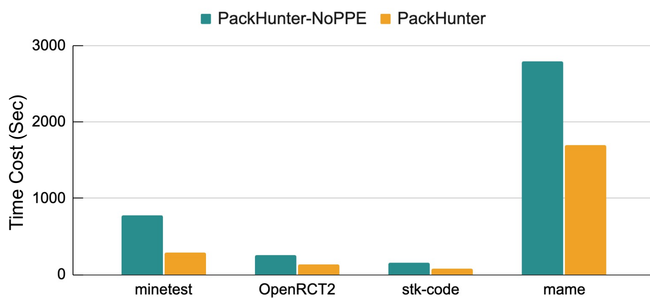

Fig. 7: The statistics of PACKHUNTER-NOPPE

times slower than PACKHUNTER due to the lack of package prioritization. It should be noted that the above four projects still demonstrate that the package prioritized enumeration improves the efficiency in several extreme cases where several missing files can be offered by multiple packages.

> **Answer to RQ3:** The package filtering and package prioritized enumeration can significantly improve the efficiency of PACKHUNTER, achieving 2.52× and 2.11× speedups on average, respectively.

5.5 Threats to Validity

The threats to the validity of our work include the following internal and external ones. The biggest threat to internal validity is that the time overhead of PACKHUNTER depends on the environment. The absence of specific packages in a C/C++ project may vary across different installation environments, as each environment may have different sets of installed packages. During our evaluation, we compare PACKHUNTER with the baselines and ablations in the same environment, which allows us to showcase the superiority of PACKHUNTER through the observed speedup. Another factor that can threaten the internal validity is the network status, which can affect the overhead of package downloading and consequently influence the time cost of the package filtering. The most significant threat to external validity is that the time overhead of PACKHUNTER is affected by the packages that cover the same number of missing files during the package enumeration. In such cases, employing a prioritization strategy still results in multiple choices, necessitating the use of random selection to make decisions. In our evaluation, we run PACKHUNTER ten times to compute the average time cost, which can mitigate the effect of randomness.

5.6 Discussion

Factors Impacting Performance. To further understand the factors impacting the performance of PACKHUNTER, we investigate the time cost of each step. As shown in Table 2, the step "Package Prioritized Enumeration" takes up the majority of analysis time, accounting for 67.5% of total time. Essentially, the time cost of this step is determined by two factors as shown in Algorithm 3. The first one is the time cost of one actual build, since PACKHUNTER requires at least one successful build to verify the solution. Specifically, for the projects whose time cost of the clean build is high, such as `mame`, `pcl`, and `minetest`, they require more analysis time than the other projects. The second one is the times of package set enumeration (Line 11-16 in Algorithm 3). Due to the design of "Package Filtering", in 26 out of 30 projects, we can deterministically recover the packages without enumeration. For the rest four projects that still require multiple enumerations, we observe that the times of package enumerations are not linearly related to the number of candidate packages. This is because our prioritized enumeration strategy is not influenced by the candidate packages that only contain a few missing files. Moreover, even with our prioritization strategy, we still encounter the cases where the candidate packages cover the same number of missing files. For these cases, we decide to use random selection strategy to make decisions, since it is simple and effective. We also compared the random selection strategy with a popularity-based selection for four projects that required prioritized enumeration: `minetest`, `OpenRCT2`, `stk-code`, and `mame`. The final iteration counts for the two methods were 1.4, 1.7, 2.3, and 2.2 versus 2, 1, 2, and 2, respectively, showing no significant difference in performance between the two strategies.

Open Source Platforms and Repositories. GitHub is currently the most popular platform for open-source projects, widely recognized within the C/C++ community. Many previous studies [9], [23] have used projects hosted on GitHub as their datasets. In terms of package managers, apt is the most commonly used in open-source projects [9]. Consequently, our experiments are based on GitHub and apt. However, our approach is not dependent on any specific platform or package manager. The symbol extraction and source code analysis we perform on packages are universally applicable to all header files, library files, and

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

source files. Therefore, our method is equally suitable for other open-source platforms and package managers, such as Sourceforge and vcpkg.

Limitations and Future Work. Firstly, the developers of a C/C++ project may define specific macros in the build script to check if a dependent package has been installed. Simply mocking the missing files will not allow the mock build to bypass the missing package error and complete the build in a single round. To address this issue, we could combine PACKHUNTER with the baseline TRYINSTALL to create a hybrid approach. This hybrid approach can handle such challenging missing package errors by first utilizing TRYINSTALL and then resolving other missing packages using PACKHUNTER. Secondly, the enumeration process may be time-consuming when dealing with a large number of packages after filtering irrelevant packages. For example, analyzing the project `mame` requires the enumeration process upon 28 packages, which introduces significant overhead when the actual build of the project is costly. One way to enhance efficiency further is to leverage more sophisticated static analysis for performing irrelevant package filtering. Specifically, we could pose a stronger constraint on the missing files provided by packages. That is, the missing files should offer all the symbols used by the accessed header/source files but not be defined in other files included within the project. The enhanced irrelevant package filtering can potentially prune more search space for the package prioritized enumeration than PACKHUNTER. We plan to extend PACKHUNTER in the two aspects above to enhance its applicability and efficiency.

6 RELATED WORK

Missing Package Recovery. Some studies have explored the missing package recovery problems for facilitating the build process of a given project in the past decade. Typically, DOCKERIZEME [24] infers the dependent packages by analyzing the import instructions in Python code snippets and generates a Dockerfile for the corresponding environment to avoid import errors. V2 [25] is an improvement of DOCKERIZEME, addressing the issue of code snippets becoming outdated due to configuration drift. While the aforementioned studies focus on code snippets, others tackle practical challenges such as complex dependency specifications. READPYE [26], PYEGO [27], and PYCRE [28] build their own knowledge graphs of the Python ecosystem, extracting syntax features, module imports, and more from Python source code to automatically infer Python-compatible runtime environments. However, unlike Python's PyPI, the C/C++ ecosystem lacks a central repository, making it challenging to construct a comprehensive knowledge graph. Furthermore, in C/C++ projects, obtaining dependencies in third-party libraries involves multiple concerns, including header file inclusion, library file linking, and conditional compilation, which cannot be addressed solely through statically analyzing source code. Our work is the first attempt to address missing package issues in C/C++ projects.

Build Script Error Detection. There is extensive literature on detecting build script errors, typically falling into three categories. The first type statically analyzes build scripts to detect dependency errors. For instance, Gunter [29] proposed to utilize Petri nets to model dependency relationships in build scripts to check their correctness. SYMAKE [30] detects bad code smell, such as cyclic and duplicated dependencies, using symbolic dependency graphs. However, due to the unsoundness of extracting dependencies, such analyses suffer from the issues of low precision and recall. The second type detects build errors by dynamically analyzing the build scripts. MKCHECK [12], for example, employs a fuzz-testing-like approach to track system calls during the build process and then triggers incremental builds for each file to validate the correctness of the dependencies. FABRICATE [31] and MEMOIZE [32] utilize build monitor to automatically resolve unspecified dependency relationships. TUP [33], IBM CLEARCASE [34], and VESTA [35] employ runtime validation to detect dependency issues. These approaches would incur significant overhead for large-scale projects. The third type combines both static and dynamic methods to detect build script errors, yielding more satisfactory results. Bezemer et al. [19] combine declared dependencies with actual dependencies during the build process to reveal missing dependencies in make build systems, while VERIBUILD [11] merges them into a unified dependency graph to detect missing and redundant dependencies. VIRTUALBUILD [13] extends the concept of unified dependency graph and uses program reduction to accelerate the detection. Although they are promising in reducing the overhead, a complete analysis requires a successful build. Our work aims to resolve missing package errors, which is a different problem from build script error detection. Following the spirit of the hybrid approaches for dependency error detection, we adopt the virtual build [13] to achieve a smooth mock build even in the absence of packages. Also, static analysis facilitates search space pruning significantly, which reduces the overall overhead further.

Automatic Build Repair. The build failures can be attributed to many factors. Some studies have investigated various root causes and proposed different automatic repair approaches. For example, BUILDMEDIC [36] summarizes three repair strategies, including version updates, dependency removal, and repository addition, which can address build failures related to Maven dependencies in Java projects. HIREBUILD [37] generates automatic repair patches for build scripts by analyzing the similarity of build logs and incorporating specific repair patterns from the scripts. HOBUFF [38] extracts error information from build logs of Gradle or Maven build systems and performs fault localization through lightweight data flow analysis using current information in the logs. Other studies address dependency-related build failures in Python projects. PYDFIX [1] detects and fixes the irreproducibility of Python builds caused by dependency errors by analyzing build logs. It primarily addresses issues arising from version specification of open-source dependency packages hosted in centralized repositories like PYPI. LOOCO [39] optimizes library version constraints to fix build failures caused by dependency conflicts in Python projects. These studies mainly focus on build failures caused by version constraints of packages. Different from them, PACKHUNTER can be regarded as an automatic build repair technique targeting missing package recovery.

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

7 Conclusion

We present PackHunter, an automated tool that recovers missing packages in C/C++ projects. PackHunter employs a comprehensive approach to address this problem, including probing missing files through a one-time mock build, filtering irrelevant packages, and conducting a package prioritized enumeration. Our evaluation demonstrates the high efficiency and effectiveness of PackHunter. To facilitate the future research, we make our tool and dataset [40] publicly available.

Acknowledgement

We thank anonymous reviewers for their insightful comments. This work is supported by the Natural Science Foundation of China (62272400) and the research grants from Huawei. Rongxin Wu works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University. Cheng Wang is the corresponding author.

References

[1] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing Dependency Errors for Python Build Reproducibility," in *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, C. Cadar and X. Zhang, Eds. ACM, 2021, pp. 439–451. [Online]. Available: https://doi.org/10.1145/3460319.3464797

[2] D. Fonović and T. G. Grbac, "A Quantitative Study of C/C++ FOSS Software Buildability," in *Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications*, 2022, pp. 1–10.

[3] N. Kerzazi, F. Khomh, and B. Adams, "Why Do Automated Builds Break? An Empirical Study," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 41–50. [Online]. Available: https://doi.org/10.1109/ICSME.2014.26

[4] H. Seo, C. Sadowski, S. G. Elbaum, E. Aftandilian, and R. W. Bowdidge, "Programmers' Build Errors: A Case Study (at Google)," in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 724–734. [Online]. Available: https://doi.org/10.1145/2568225.2568255

[5] D. Wu, L. Chen, Y. Zhou, and B. Xu, "How do developers use C++ libraries? An empirical study," in *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015*, H. Xu, Ed. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2015, pp. 260–265. [Online]. Available: https://doi.org/10.18293/SEKE2015-9

[6] A. Miranda and J. Pimentel, "On the Use of Package Managers by the C++ Open-Source Community," in *Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018*, H. M. Haddad, R. L. Wainwright, and R. Chbeir, Eds. ACM, 2018, pp. 1483–1491. [Online]. Available: https://doi.org/10.1145/3167132.3167290

[7] Microsoft, "vcpkg," 2023, accessed on 28/12/2023. [Online]. Available: https://vcpkg.io/en/

[8] JFrog, "Conan," 2023, accessed on 28/12/2023. [Online]. Available: https://conan.io/

[9] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards Understanding Third-party Library Dependency in C/C++ Ecosystem," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 106:1–106:12. [Online]. Available: https://doi.org/10.1145/3551349.3560432

[10] J. Long, "dependency-check – File Type Analyzers," 2023, accessed on 28/12/2023. [Online]. Available: https://jeremylong.github.io/DependencyCheck/analyzers/index.html

[11] G. Fan, C. Wang, R. Wu, X. Xiao, Q. Shi, and C. Zhang, "Escaping Dependency Hell: Finding Build Dependency Errors with the Unified Dependency Graph," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 463–474. [Online]. Available: https://doi.org/10.1145/3395363.3397388

[12] N. Licker and A. Rice, "Detecting Incorrect Build Rules," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 1234–1244. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00125

[13] R. Wu, M. Chen, C. Wang, G. Fan, J. Qiu, and C. Zhang, "Accelerating Build Dependency Error Detection via Virtual Build," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 5:1–5:12. [Online]. Available: https://doi.org/10.1145/3551349.3556930

[14] U. Packages, "All packages [Ubuntu 22.04 - Jammy]," 2023, accessed on 28/12/2023. [Online]. Available: https://packages.ubuntu.com/jammy

[15] Python Software Foundation, "Python random module," 2023, accessed on 28/12/2023. [Online]. Available: https://docs.python.org/3/library/random.html

[16] R. Durstenfeld, "Algorithm 235: random permutation," *Communications of the ACM*, vol. 7, no. 7, p. 420, 1964.

[17] PackHunter, "Dataset of Empirical Study," 2024. [Online]. Available: https://github.com/PackHunter-dataset/Empircal_Study

[18] googke, "guetzli," 2024. [Online]. Available: https://github.com/google/guetzli

[19] C. Bezemer, S. McIntosh, B. Adams, D. M. Germán, and A. E. Hassan, "An empirical study of unspecified dependencies in make-based build systems," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3117–3148, 2017. [Online]. Available: https://doi.org/10.1007/s10664-017-9510-8

[20] D. P. Contributors, "The Debian package management tools: APT," 2023, accessed on 28/12/2023. [Online]. Available: https://www.debian.org/doc/manuals/debian-faq/pkgtools.en.html

[21] Free Software Foundation Inc., "Readelf," 2023, accessed on 28/12/2023. [Online]. Available: https://sourceware.org/binutils/docs/binutils/readelf.html

[22] Michael L. Collard, Jonathan I. Maletic, "srcML," 2023, accessed on 28/12/2023. [Online]. Available: https://www.srcml.org/

[23] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying Open-Source License Violation and 1-day Security Risk at Large Scale," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds. ACM, 2017, pp. 2169–2185. [Online]. Available: https://doi.org/10.1145/3133956.3134048

[24] E. Horton and C. Parnin, "DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 328–338. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00047

[25] ——, "V2: Fast Detection of Configuration Drift in Python," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 477–488. [Online]. Available: https://doi.org/10.1109/ASE.2019.00052

[26] W. Cheng, W. Hu, and X. Ma, "Revisiting Knowledge-Based Inference of Python Runtime Environments: A Realistic and Adaptive Approach," *IEEE Trans. Software Eng.*, vol. 50, no. 2, pp. 258–279, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2023.3346474

[27] H. Ye, W. Chen, W. Dou, G. Wu, and J. Wei, "Knowledge-Based Environment Dependency Inference for Python Programs," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1245–1256. [Online]. Available: https://doi.org/10.1145/3510003.3510127

[28] W. Cheng, X. Zhu, and W. Hu, "Conflict-aware Inference of Python Compatible Runtime Environments with Domain Knowledge Graph," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 451–461. [Online]. Available: https://doi.org/10.1145/3510003.3510078

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2024.3506629

[29] C. A. Gunter, "Abstracting Dependencies between Software Configuration Items," in *Proceedings of the Fourth ACM SIGSOFT Symposium on Foundations of Software Engineering, San Francisco, California, USA, October 16-18, 1996*, D. Garlan, Ed. ACM, 1996, pp. 167–178. [Online]. Available: https://doi.org/10.1145/239098.239129

[30] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "SYMake: A Build Code Analysis and Refactoring Tool for Makefiles," in *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, M. Goedicke, T. Menzies, and M. Saeki, Eds. ACM, 2012, pp. 366–369. [Online]. Available: https://doi.org/10.1145/2351676.2351749

[31] B. Technology., "fabricate," 2022, accessed on 28/12/2023. [Online]. Available: https://github.com/brushtechnology/fabricate

[32] B. McCloskey, "Memoize," 2022, accessed on 28/12/2023. [Online]. Available: https://github.com/kgaughan/memoize.py

[33] M. Shal, "Build System Rules and Algorithms," *Published online (2009). Retrieved July*, vol. 18, p. 2013, 2009. [Online]. Available: http://gittup.org/tup/build_system_rules_and_algorithms.pdf

[34] I. B. M. C. (IBM), "IBM Rational Clearcase," 2020, accessed on 28/12/2023. [Online]. Available: https://www.ibm.com/us-en/marketplace/rational-clearcase

[35] VestaSys, "Vesta Configuration Management System," 2020, accessed on 28/12/2023. [Online]. Available: http://www.vestasys.org/

[36] C. Macho, S. McIntosh, and M. Pinzger, "Automatically Repairing Dependency-Related Build Breakage," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 106–117. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330201

[37] F. Hassan and X. Wang, "HireBuild: An Automatic Approach to History-Driven Repair of Build Scripts," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 1078–1089. [Online]. Available: https://doi.org/10.1145/3180155.3180181

[38] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-Driven Build Failure Fixing: How Far Are We?" in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 43–54. [Online]. Available: https://doi.org/10.1145/3293882.3330578

[39] H. Wang, S. Liu, L. Zhang, and C. Xu, "Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 198–210. [Online]. Available: https://doi.org/10.1145/3611643.3616264

[40] PackHunter, "PackHunter," 2024. [Online]. Available: https://github.com/L00000719/PackHunter

Rongxin Wu received the PhD degree from HKUST, in 2017. He is currently an associate professor at the Department of Computer Science and Technology, School of Informatics, Xiamen University. His research interests include program analysis, software security, and mining software repository. His research work has been regularly published in top conferences and journals in the research communities of program languages and software engineering, including POPL, PLDI, ATC, ICSE, FSE, ISSTA, ASE, and TSE and so on. He has served as a reviewer in reputable international journals and a program committee member in several international conferences (FSE'25, ISSTA'25, SANER'25, FSE'24, ISSTA'24, ASE'23, SANER'23, and ASE 2021 and so on). He is a two-time recipient of the ACM SIGSOFT Distinguished Paper Award. More information about him can be found at: https://wurongxin1987.github.io/wurongxin.xmu.edu.cn/

Zhiling Huang is a post-graduate student at the Department of Computer Science and Technology, School of Informatics, Xiamen University. He received his Bachelor's degree in Engineering from Xiamen University in 2022. His current research focuses on software engineering, particularly on C/C++ compilation processes and build systems.

Zige Tian is a post-graduate student at the Department of Computer Science and Technology, School of Informatics, Xiamen University. She received her Bachelor's degree in Engineering from Xiamen University in 2023. Her current research area is software engineering, with research interests including build error detection and dependency management.

Chengpeng Wang is a post-doctoral research fellow at the Computer Science Department of Purdue University. His research mainly focuses on the use of program analysis, especially static analysis, to improve software reliability and performance. He is also interested in the intersection of machine learning techniques, such as Large Language Models, and symbolic analysis techniques, with the aim of establishing neuro-symbolic program analysis. His contributions to the field have been recognized through publications in esteemed conferences and journals on programming languages, software engineering, and systems. Notably, he has been awarded the SIGPLAN Distinguished Paper Award (2022) and the ASPLOS Best Paper Award (2024). He earned his Ph.D. from the Hong Kong University of Science and Technology in 2023. Before that, he completed his bachelor's and master's degrees at Tsinghua University in 2016 and 2019, respectively.

Xiangyu Zhang is a professor specializing in AI security, software analysis and cyber forensics. His work involves developing techniques to detect bugs, including security vulnerabilities, in traditional software systems as well as AI models and systems, and to diagnose runtime failures. He has served as the Principal Investigator (PI) for numerous projects funded by organizations such as DARPA, IARPA, ONR, NSF, AirForce, and industry. Many of the techniques developed by his team have successfully transitioned into practical applications. His research outcome has been published on top venues in the areas of Security, AI, Software Engineering, and Programming Languages, and recognized by various distinguished paper awards including the prestigious ACM Distinguished Dissertation Awards. He has mentored over 30 PhD students and post-docs, with fifteen securing academic positions in various universities. Many of them have been honored with NSF Career Awards or comparable recognitions.

Authorized licensed use limited to: Purdue University. Downloaded on December 02,2024 at 21:09:18 UTC from IEEE Xplore. Restrictions apply.
© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.