INSIGHT: Exploring Cross-Ecosystem Vulnerability Impacts

Meiqiu Xu
Northeastern University
Shenyang, China
xumeiqiu@outlook.com

Ying Wang*
Northeastern University, and The Hong Kong University of Science and Technology
Shenyang, Hong Kong, China
wangying@swc.neu.edu.cn

Shing-Chi Cheung
The Hong Kong University of Science and Technology
Hong Kong, China
scc@cse.ust.hk

Hai Yu
Northeastern University
Shenyang, China
yuhai@mail.neu.edu.cn

Zhiliang Zhu
National Frontiers Science Center for Industrial Intelligence and Systems Optimization, and Key Laboratory of Data Analytics and Optimization for Smart Industry, Northeastern University
Shenyang, China
ZHUZhiLiang_NEU@163.com

ABSTRACT

Vulnerabilities, referred to as CLV issues, are induced by cross-language invocations of vulnerable libraries. Such issues greatly increase the attack surface of Python/Java projects due to their pervasive use of C libraries. Existing Python/Java build tools in PyPI and Maven ecosystems fail to report the dependency on vulnerable libraries written in other languages such as C. CLV issues are easily missed by developers. In this paper, we conduct the first empirical study on the status quo of CLV issues in PyPI and Maven ecosystems. It is found that 82,951 projects in these ecosystems are directly or indirectly dependent on libraries compiled from the C project versions that are identified to be vulnerable in CVE reports. Our study arouses the awareness of CLV issues in popular ecosystems and presents related analysis results.

The study also leads to the development of the first automated mechanism, INSIGHT, which provides a turn-key solution to the identification of CLV issues in PyPI and Maven projects based on published CVE reports of vulnerable C projects. INSIGHT automatically identifies if a PyPI or Maven project is using a C library compiled from vulnerable C project versions in published CVE reports. It also deduces the vulnerable APIs involved by analyzing the usage of various foreign function interfaces such as *CFFI* and *JNI* in the concerned PyPI or Maven project. INSIGHT achieves a high detection rate of 88.4% on a popular CLV issue benchmark. Contributing to the open-source community, we report 226 CLV issues detected in the actively maintained PyPI and Maven projects that are directly dependent on vulnerable C library versions. Our reports are well received and appreciated by developers with queries on the availability of INSIGHT. 127 reported issues (56.2%) were quickly confirmed by developers and 74.8% of them were fixed/under fixing by popular projects, such as `Mongodb` [40] and `Eclipse/Sumo` [19].

KEYWORDS

Software Ecosystem; Multilingual Program Analysis

ACM Reference Format:
Meiqiu Xu, Ying Wang, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2022. INSIGHT: Exploring Cross-Ecosystem Vulnerability Impacts. In *Proceedings of 37th IEEE/ACM International Conference on Automated Software Engineering, Rochester, MI, USA, October 10–14, 2022 (ASE '22),* 13 pages.
https://doi.org/10.1145/3551349.3556921

*Ying Wang is corresponding author.

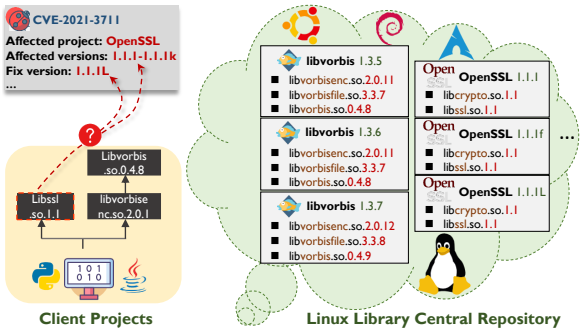

Figure 1: Challenges of finding source code provenances of libraries

1 INTRODUCTION

Foreign function interfaces (FFIs) like *CFFI* [4] and *JNI* [36] provide powerful bridging mechanisms that allow a project implementing in one language to access libraries written in another language. The use of FFIs by Python and Java projects in the PyPI and Maven ecosystems is common. Our investigation shows that, as of March 1 2022, 46.7% and 32.1% of projects in PyPI and Maven ecosystems directly or transitively use C libraries. However, existing library managers and dependency management tools (e.g., `Maven` [39], `Pip` [43] and `DependencyBot` [18]) in these ecosystems fail to warn developers against the use of vulnerable libraries written in another language. As a result, Python and Java developers can easily miss the security risks induced by the use of vulnerable C libraries, as demonstrated by the real-life example in Section 2.2. In this paper, we refer to the vulnerability issues induced by cross-language invocations as *CLV issues*.

To understand the status quo of cross-ecosystem vulnerabilities, we conduct an empirical study on the CLV issues in the PyPI and Maven ecosystems induced by vulnerable C libraries. We select PyPI, Maven and C because of their popularity. The results of our study would be of relevancy to many developers. It finds that 24,605 projects in PyPI and Maven ecosystems are directly or indirectly

dependent on libraries compiled from the C project versions that are identified to be vulnerable in CVE reports.

Our study also leads to the development of an automated tool, Insight, that complements existing dependency management tools to identify the projects suffering from CLV issues. The development of Insight needs to resolve two challenges.

(1) Gaps between CVE reports and the build scripts of PyPI or Maven projects. CVE reports identify vulnerable C project versions whereas PyPI or Maven project build scripts identify the C library versions used. However, C libraries and their corresponding C projects have different naming and versioning schemes. In practice, a C project version is often compiled into multiple published C library versions. However, CVE (Common Vulnerabilities and Exposures) reports only indicate the C project versions that contain the vulnerabilities. Non-trivial analysis effort is required to track down the vulnerable C library versions from CVE reports.

Figure 1 presents an illustrative scenario of the version number differences. `OpenSSL` is a C project compiled into two C libraries `libcrypto.so` and `libssl.so`. It is reported by *CVE-2021-3711* that the `OpenSSL` project versions 1.1.1-1.1.1k are vulnerable. The vulnerability is reported to be fixed in `OpenSSL` project version 1.1.1L. However, the version number of the `libssl.so` library evolves separately from that of its OpenSSL project. It is also true for the `libvorbis` project and its libraries. C developers may not necessarily use a separate version number for a library after code updates or fixes. Indeed, `libssl.so.1.1` in `OpenSSL` 1.1.1L contains the fix for `libssl.so.1.1` in `OpenSSL` 1.1.1-1.1.1k. As discussed in Section 2, nearly 80% of C libraries violate the semantic versioning rules in Linux community. The version number of libraries, therefore, cannot reliably identify their project version. As a result, it is challenging to determine if the `libssl.so.1.1` library used by a client project originates from a vulnerable `OpenSSL` version.

(2) Varieties of FFIs. FFIs with different syntax rules and mechanisms are supported in PyPI and Maven ecosystems for interfacing Java/Python code with the APIs in C libraries. For instance, three types of FFIs are commonly used for bridging between Python and C, including *CFFI* [4], *Cython* [15] and *Ctypes* [7]. Two types of FFIs, namely, *JNI* [36] and *JNA* [35] are often adopted to bind Java and C code. There is no automated analysis technique to accurately extract the concerned APIs invoked by the various types of FFIs, which support different syntax rules and mechanisms.

Approach. To address the above challenges, we develop an automated tool, Insight, which can continually monitor the impacts of vulnerable C libraries on their client projects in PyPI and Maven ecosystems and report CLV issues detected to the project developers. To determine if a C library can introduce vulnerabilities, Insight identifies the C project version where the library originates. The identification is achieved in two steps. First, Insight constructs a large-scale knowledge repository indexing the metadata of all C project versions with their known compiled libraries published in the mainstream Linux distributions. Second, Insight determines a C library's identity by matching the similarity of the library's feature against those in our knowledge repository. The similarity matching is performed by leveraging a state-of-the-art technique LibRARIAN [58]. If the C library is compiled from vulnerable C project versions identified by CVE reports, Insight further analyzes the accessibility of corresponding vulnerable APIs via parsing cross-language invocations, and generates diagnosis info for the CLV issues. Various parsing patterns are proposed to analyze the C library APIs invoked in five types of FFIs, namely, *CFFI*, *Cython*, *Cytpes*, *JNI* and *JNA*.

Evaluation and Study Results. To evaluate Insight, we collected 69 real CLV issues that can be exposed by the projects' bundled tests as a ground truth dataset. Insight achieves a detection rate of 88.4%. By applying Insight to millions of projects in PyPI and Maven ecosystems on the snapshots of March 1, 2022, we further conducted a large-scale empirical study to explore the status quo of cross-ecosystem vulnerability impacts. We identified 82,951 multilingual projects directly or transitively using vulnerable C libraries. Among them, 21,326 projects (25.7%) can access vulnerable C-APIs. Such situation may be grossly underestimated since Insight only traced to project versions for 75.2% of C libraries and located a limited number of vulnerable APIs. To validate the usefulness of Insight, we report 226 CLV issues to respective project developers with detailed diagnosis information. Eventually, 127 reported issues (56.2%) were confirmed by developers and 95 of them (74.8%) had been fixed/under fixing by well-known projects, such as `Mongodb` [40], `Libgit2` [37] and `Eclipse/Sumo` [19].

Contributions. To summarize, the contributions of this paper are:

- **CLV issue detection technique.** Observing an important limitation of existing build tool support, we propose an automated mechanism, Insight, that is able to detect and diagnose CLV issues in multilingual projects based on the published CVE reports of C projects. The mechanism is scalable to very large ecosystems.
- **Quantitative analysis of CLV issues.** Leveraging Insight, we conduct the first empirical study on the CLV issues arising from the use of vulnerable C libraries in two large ecosystems: PyPI and Maven. The study provides quantitative analysis of the pervasiveness of CLV issues and the means of their propagation. Such CLV issues greatly increase the attack surface of projects. The analysis indicates that many CLV issues are propagated widely in the ecosystems by the pivotal projects, which provide popular third-party libraries. The results provide useful information for the ecosystems to curb the issues.
- **A reproduction package and successful detection of new CLV issues.** We implement Insight and provide a reproduction package at Insight's website (**http://insight-clv-detection.com/**) for future research, which includes: (1) an available Insight tool, (2) a benchmark dataset containing detected CLV issues, and (3) large-scale vulnerability and library dependency metadata on recent ecosystem snapshots. As an effort to effectively reduce the attack surface in PyPI and Maven ecosystems, we reported 226 CLV issues arising from the direct dependency on vulnerable C libraries by pivotal projects such as `Mongodb` and `Eclipse/Sumo`. 127 (56.2%) reported issues were rapidly confirmed and 95 of them (74.8%) have been fixed or were under fixing. The fixes result in reduction of CLV issues in 49,118 projects.

2 BACKGROUND

2.1 C Library Releases and Their Naming Rules

Typically, a C project version is complied into multiple C libraries in the form of `*.so` files, and publish them to the Linux library central repository. As of March 1, 2022, three mainstream Linux

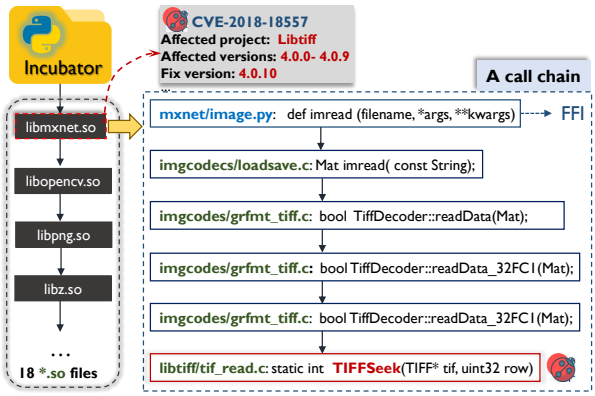

Figure 2: An illustrative example of issue #14623 [28] in `Incubator`

distributions, including *Ubuntu* [50], *Debian* [16] and *Arch Linux* [1], host 485,021 versions of 165,347 C projects, involving 829,242 C libraries. With the access to such rich facilities, client projects in an ecosystem can leverage `*.so` files compiled from various C project versions as third-party libraries according to their requirements.

Linux community recommends a set of semantic versioning rules for C libraries (i.e., `*.so` files). Specifically, a fully-qualified soname consists of a prefix "*lib*", a library name, a phrase "*.so*", followed up by a version number. Official documentation [23] encourages C developers to adopt the form of "*Major.Minor.Patch*" to specify the unique version numbers for libraries, based on versioning rules.

However, our investigation in the library central repositories of three Linux distributions indicated that developers often violated the semantic versioning rules in C library releases. We find that among the 451,233 C libraries in *Ubuntu* distribution, 364,420 of them (80.8%) were published with non-standard version numbers, in the form of "*Major*" or "*Major.Minor*". These libraries did not increment the version numbers for stating API changes or bug fixes during evolution. Similarly, in *Debian* and *Arch Linux* distributions, nearly 80.0% of C libraries are published with non-standard version numbers (*Debian*: 511,825/636,132 = 80.5%; *Arch Linux*: 25,646/30,824 = 83.2%). As a result, such version numbers of `*.so` files cannot be taken as the identity of C libraries.

2.2 A Motivating Example

Existing build tools of PyPI and Maven fail to report the dependency on vulnerable C libraries. Developers in these ecosystems can easily miss CLV issues induced by such libraries. They need to diagnose the root causes when CLV issues are noted. Due to the different version naming schemes between C projects and their compiled libraries, such diagnosis is time-consuming. Figure 2 gives an illustrative example. Developers filed an issue report #14623 [28] in project `Incubator`, in which they expressed that, the Python project introduced a critical vulnerability *CVE-2018-18557* [12] in a C library `libmxnet.so`. Since the *soname* ignores a version number, they figured out the project version (`Libtiff` 4.0.9) where this library originates, based on `Incubator`'s development documentations. `Incubator` is a deep learning framework designed for both efficiency and flexibility, achieving over 20,000 stars on GitHub and 329,000 monthly downloads. Due to its considerable impacts on the ecosystem, this issue attracts seven developers to participate in the discussions. To avoid such vulnerability issues from C libraries, they decided to keep an eye on the C projects' release notes in the future, since `Incubator` highly depends on 18 `*.so` files.

Three years later, a downstream project using an old version of `Incubator`, triggered this vulnerability *CVE-2018-18557* issue. The developer posted a call chain that can access to the vulnerable C-API `TIFFSeek()` in report #14623, to arouse other `Incubator` users' attentions to validate if they were affected by such a vulnerability.

Similar situations can be found in many issue reports. In the CLV issue report #167 [30] of Java project `Etlflow`, an experienced developer commented that: "*An automated tool for detecting vulnerability affects from C dependencies is required to help us perform the exhaustive analyses.*" This motivates us to develop a mechanism to help developers combat CLV issues and conduct a study on the status quo of these issues in popular ecosystems.

3 INSIGHT APPROACH

To locate the PyPI and Maven projects that are dependent on vulnerable C libraries, we developed a tool, INSIGHT, to capture CLV issues from CVE reports. We deploy it at PyPI and Maven ecosystems to continually detect and diagnose CLV issues induced by libraries compiled from vulnerable C projects.

Figure 3 gives an overview of INSIGHT. We refer to the PyPI and Maven projects that directly depend on C libraries as *client projects*. INSIGHT comprises three stages: C library extraction, C project version tracing, and vulnerable C-API analysis. INSIGHT identifies the C libraries used by a client project from its archive in the ecosystem. To trace an extracted C library to its originated C project version, INSIGHT constructs a large-scale knowledge repository indexing the main features inherent to C library versions published in three mainstream Linux distributions. Leveraging a recent tool LIBRARAIN [58], INSIGHT determines the project version of a given C library, by iteratively comparing it with the known version instances in our knowledge repository based on similarity matching. To boost automation, INSIGHT collects the metadata of vulnerability reports directly from the official security advisory websites to construct a large-scale vulnerability database. INSIGHT checks for each extracted C library whether its parent C project version contains vulnerabilities. Finally, the parsing patterns that we design for various types of FFIs, INSIGHT performs cross-language program analysis to determine if the client project code can access the concerned vulnerable C-APIs.

3.1 C Library Extraction

Project developers in PyPI and Maven ecosystems are required to publish project archives at central repositories. The archives include binaries, configuration files and all necessary dependencies such as copies of the C libraries used. INSIGHT can therefore determine if a client project depends on C libraries by extracting `*.so` files, which are copies of C libraries, from its published archive. The extraction works in two steps:

- ***Step 1: File exploration.*** INSIGHT takes the archives (`*.whl` packages for PyPI; `*.Jar` or `*.aar` packages for Maven) of client projects as inputs and leverages tools WHEEL UNPACK [53] and ZIPFILE [54] to explore the files in these archives. Since an archive

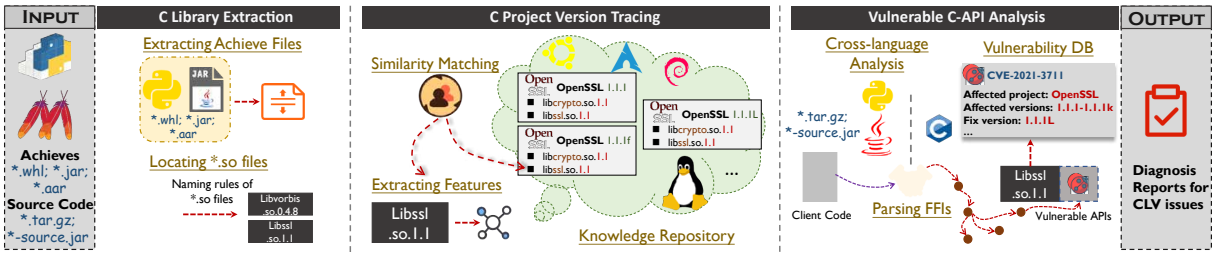

Figure 3: The overall architecture of INSIGHT

may contain `*.Jar` and `*.aar` packages, INSIGHT iteratively expands these packages for file exploration until no further packages can be found.

- ***Step 2: C library extraction.*** Based on the C library naming rules, INSIGHT uses the regex "`lib.+.so(.[0-9]+){0,3}`" to extract C libraries (e.g., *libcrypto.so.1.1*) from the explored files.

3.2 C Project Version Tracing

To decide if an extracted `*.so` file can introduce vulnerabilities into its associated client project, INSIGHT traces the C project version from which the file was compiled. To achieve that, INSIGHT incorporates a state-of-the-art technique, LIBRARIAN [58], which proposes to determine the parent project of a C library embedded in an Android APK by matching its features with those of the libraries compiled from the project. To adopt the technique for CLV issue detection in PyPI and Maven ecosystems, INSIGHT constructs a large-scale knowledge repository by collecting the metadata of millions of C libraries with their known versions published in the mainstream Linux distributions. The construction of the knowledge repository took over six months. The repository is over 3.2 TB in size, indexing 485,021 C project versions and 829,242 C libraries as of 1 March 2022 (see statistics in Table 1). For a given `*.so` file, INSIGHT decides its identity by matching its feature similarity with those in the knowledge repository. We explain the procedures taken to adopt LIBRARIAN for tracing the parent project versions of extracted C libraries in INSIGHT below.

- ***Extracting characterization features of C library versions.*** C libraries are packaged as *.so files of ELF (Executable and Link-able Format) binaries. Each file contains a symbol table with externally visible identifiers such as function names, global symbols, local symbols, and imported symbols. Following LIBRARIAN, INSIGHT extracts five features from each *.so file to characterize its library version, by parsing ELF metadata: (a) *Exported Globals*: Visible variables that can be accessed externally; (b) *Imported Globals*: Variables from other libraries that are used in this library; (c) *Exported APIs*: Visible APIs that can be called by other libraries; (d) *Imported APIs*: APIs from other libraries that are used in this library; and (e) *Dependencies*: Library dependencies that are loaded by the ELF object. The five features represent the code elements of a library that would be expected to change based on a versioning scheme during evolution, which can distinguish different libraries and their versions. The five features are chosen because of their stability across platforms, regardless of the underlying architecture or compilation environments. Like LIBRARIAN [58], we do not characterize C libraries using their code structure features (e.g., control-flow features) as they can be volatile and subject to change between compilations and across architectures.
- ***Constructing knowledge repository for C libraries.*** INSIGHT constructs a large-scale knowledge repository by collecting the metadata of all `*.so` library files compiled by C project versions published in three mainstream Linux distributions, including *Ubuntu*, *Debian* and *Arch Linux*. By crawling the information of C projects in these distributions, INSIGHT indexes collected C libraries to their corresponding project versions. Each C library version is characterized by a vector of the five aforementioned features. We represent the metadata of a C project version in a 3-tuple <*P*, *Lib*, *Src*>, where
  - *P* denotes the project name with a version number;
  - *Lib* is a collection of C libraries compiled from the project version *P*. Each C library is a 2-tuple <*soname*, *FV*>, where *soname* corresponds to the name of its `*.so` file, and *FV* denotes a vector recording the features extracted from `*.so` file.
  - *Src* denotes the source code of the project version *P*.

  INSIGHT monitors the evolution of Linux library central repositories and incrementally updates the knowledge repository when a new distribution version is published.
- ***Determining project version by feature similarity.*** The large-scale knowledge repository enables INSIGHT to determine the project version where a C library originates, based on similarity matching. It works in two steps:
  - *Step 1: Reducing the number of C libraries for comparison.* For each `*.so` file extracted from a client project, INSIGHT searches for the C libraries matched with its *soname* in the constructed knowledge repository, using the regex `soname([0-9]*)`. As discussed in Section 2, since the version number of a `*.so` file may remain unaltered after API changes or bug fixes, the search may return multiple `*.so` files. INSIGHT considers the returned files as candidates for feature similarity comparison.
  - *Step 2: Similarity comparison.* INSIGHT extracts the feature vector FV_1 from the `*.so` file under analysis, and iteratively compares it with those of the returned libraries FV_2 in our knowledge repository. The comparison adopts the similarity score metric bin^2sim proposed by LIBRARIAN [58]:

$$bin^2sim(FV_1, FV_2) = \frac{|FV_1 \cap FV_2|}{|FV_1 \cup FV_2|} \in [0, 1] \quad (1)$$

  bin^2sim allows INSIGHT to account for addition or removal of features between different libraries and versions. The similarity score is a real number between 0 and 1, with a score of 1 indicating identical features, a score of 0 indicating no shared features between the two libraries, and a fractional

value indicating a partial match. INSIGHT considers the `*.so` file successfully matched a collected library version in the knowledge repository if the bin^2sim is above 0.85, which is the threshold adopted by LIBRARIAN. INSIGHT determines the matching project version based on the highest bin^2sim value, in the cases that more than one comparison achieve a similarity score above the threshold.

3.3 Vulnerable C-API Analysis

After tracing the C project version of an extracted `*.so` file, INSIGHT can determine if it contains vulnerabilities based on the disclosed vulnerability reports. Furthermore, INSIGHT reports whether the Python/Java client code can access the vulnerable APIs in C libraries by analyzing the cross-language invocations.

- ***Constructing Vulnerability Database.*** INSIGHT constructs a large-scale vulnerability database by collecting the metadata of vulnerability reports from three official security advisory websites: *Ubuntu Security* [51], *Debian Security* [17], and *Arch Linux Security* [2]. For each CVE report, it identifies the vulnerability's *unique identifier*, *affected project*, *affected project versions*, *fix versions* and *severity level*. In addition, INSIGHT identifies the GitHub commits or GitLab commits provided in vulnerability reports for patching the corresponding CVEs. Like existing studies [74, 75, 93], it further identifies the revised APIs in relevant commits as the vulnerable APIs that induced the security issues. INSIGHT continually monitors the security advisory websites and updates the vulnerability database.
- ***Analyzing Cross-language Invocations.*** Based on the constructed vulnerability database, for each C library extracted from a client project, INSIGHT checks whether its parent C project version contains a CVE. If so, INSIGHT further analyzes whether the client code can access the concerned vulnerable APIs. INSIGHT takes the source code of client project and C libraries as inputs and generates diagnosis reports for CLV issues. The reports point out the call chains from entry C-APIs to the concerned vulnerable functions from C libraries to facilitate developers' diagnosis.
  To perform the analysis, INSIGHT parses the FFIs that support cross-language invocations to locate the entry APIs of C libraries used by client projects. However, each type of FFIs has its own syntax rules and mechanism to interface with the APIs in C libraries. To address the challenge, we design parsing patterns for five popular FFI types by studying its official FFI documentation. To allow the parsing scaling up to the large number of projects in an ecosystem, each pattern is so designed that it can be automated by scripts. Figure 4 shows the five FFI parsing patterns. Each parsing pattern identifies the concerned FFI configurations on the Python/Java side, according to the FFI syntax rules. For example, *Ctypes* uses function `cdll.LoadLibrary()` or `CDLL()` to create instance objects of C libraries in `*.py` file. Leveraging a C static analysis tool on program control flows, CFLOW [5], INSIGHT obtains the function call graphs within C libraries rooting at the entry C-APIs declared in FFIs. It reports a CLV issue with the diagnostic information of a function calling sequence in C libraries to reach the concerned vulnerable C-API.

Table 1: Statistics of INSIGHT's knowledge repository for C libraries

Linux Distribution	*#C Project*	*#C Project Versions*	*#C Libraries*
Ubuntu	121,507	273,813	451,233
Debian	111,122	269,950	636,132
Arch Linux	12,264	12,297	30,824
Sum	165,347	485,021	829,242

4 EMPIRICAL STUDY AND EVALUATION

In this section, we present our empirical study and evaluation. Specifically, we aim to answer the following four research questions.

- **RQ1 (Effectiveness of INSIGHT)**: ***How effective is INSIGHT in detecting CLV issues?*** To answer RQ1, we construct a benchmark dataset containing 69 real CLV issues for validating whether INSIGHT can capture these issues in their corresponding buggy project versions.
- **RQ2 (Pervasiveness of Use of C Libraries)**: ***What is the scale of projects that depend on C libraries in the PyPI and Maven ecosystems?*** To answer RQ2, we gather the statistics of projects that directly or transitively depend on C libraries on recent snapshots of PyPI and Maven ecosystems.
- **RQ3 (Cross-ecosystem Vulnerability Impacts)**: ***To what extent are vulnerable C libraries used by projects in the PyPI and Maven ecosystems?*** To answer RQ3, we apply INSIGHT to real-world projects to explore the status quo of cross-ecosystem vulnerability impacts.
- **RQ4 (Usefulness of INSIGHT)**: ***Is the diagnosis information provided by INSIGHT useful to developers for dealing with CLV issues? To what extent do practitioners concern the CLV issues?*** To answer RQ4, we report the detected CLV issues together with diagnosis information to respective developers. We evaluate INSIGHT's usefulness based on developers' feedback.
- **RQ5 (Fixing Strategies and Challenges)**: ***What are common practices for fixing CLV issues? What are challenges of figuring out a fix?*** To answer RQ5, by digging into developers' feedback on our reported CLV issues, we distill their common fixing strategies and challenges to be addressed.

4.1 Data Collection

INSIGHT's detection capability relies on its construction of *knowledge repository for C libraries* and *vulnerability database*. At the time of carrying out our experiments, we collected metadata from the snapshots of three mainstream Linux distributions: *Ubuntu* [50], *Debian* [16] and *Arch Linux* [1], on March 1, 2022.

4.1.1 ***INSIGHT's Knowledge Repository for C Libraries.*** Table 1 summarizes the statistics of INSIGHT's knowledge repository. It comprises of 485,021 C project versions released in three Linux distributions, involving 829,242 C libraries. By collecting the metadata of C projects provided by respective maintainers, INSIGHT maps the C libraries to their corresponding project versions. More importantly, for each C library, it constructs a unique feature vector as the identification of its precise version.

4.1.2 ***INSIGHT's Vulnerability Database.*** Table 2 depicts the statistics of INSIGHT's vulnerability database. Our vulnerability database consists of 45,473 vulnerability reports collected from three official security advisory websites: *Ubuntu Security* [51], *Debian*

Programming Language	*FFI Type*	*Parsing Pattern*	*Illustrative Example*
Python-C	*CFFI*	(1) Identify the `*.py` file declared as a parameter assigned to the CFFI attribute `cff_modules` of function `setup()` in the script `setup.py`; (2) Locate the entry C-APIs declared in the function `cdef()` from the above identified `*.py` file	*# xmlstarlet/setup.py* setup(… *cffi_modules* = ["xmlstarlet_build.py: FFIBUILDER"], …) *# xmlstarlet/xmlstarlet_build.py* FFIBUILDER.cdef(‘‘‘ int *selMain*(int *argc*, char * * *argv*); int *trMain*(int *argc*, char * * *argv*); ’’’)
	Cython	Identify the entry C-APIs and C header file `*.h` in the Cython configuration file `*.pxd`	*# cython-hidapi/chid.pxd* cdef extern from "hidapi.h": int *hid_get_manufacturer_string*(*hid_device**, *wchar_t **, *size_t*); int *hid_get_product_string*(*hid_device**, *wchar_t **, *size_t*);
	Ctypes	(1) Identify the instance objects of C libraries created by *Ctypes* function `cdll.LoadLibrary()` or `CDLL()` in the script `*.py` that imports *Ctypes* module; (2) Locate the entry C-APIs invoked through the above identified instance object	*#PublicFiles/usb_device.py* from *ctypes* import * USB2XXXLib = *cdll.LoadLibrary*("./lib/linux/ARMv7/*libUSB2XXX.so*") def DEV_GetDeviceInfo(*DevHandle*, *pDevInfo*, *pFunctionStr*): return USB2XXXLib.*DEV_GetDeviceInfo*(*DevHandle*, *pDevInfo*, *pFunctionStr*)

Java-C	*JNI*	(1) Identify the native methods in `*.java` file;
		(2) Locate the entry C-APIs wrapped by the JNI functions whose name containing the above native method name in the JNI implementation file `*.c`
			#zstd/Zstd.java
			public class Zstd{
			public static native long *decompressUnsafe*(long *dst*, long *dstSize*, long *src*, long *srcSize*);}
			# native/jni_zstd.c
			JNIEXPORT jlong JNICALL *Java_com_github_zstd_decompressUnsafe*(JNIEnv * *env*){
			return *ZSTD_decompress*((void *)(intptr_t) *dst_buf_ptr*, (size_t) *dst_size*); }

	JNA	(1) Identify an interface that extends the interface `com.sun.jna.Library`, in which a C library is loaded via the function `Native.loadLibrary()`;
		(2) Locate the entry C-APIs declared in the above identified interface
			# gethostname4j/Hostname.java
			private interface *UnixCLibrary* extends *Library* {
			UnixCLibrary INSTANCE = *Native.loadLibrary*("c", *UnixCLibrary.class*);
			public int *gethostname*(byte[] *hostname*, int *bufferSize*);
			}

Figure 4: INSIGHT's parsing patterns for various types of FFIs

Table 2: Statistics of INSIGHT's vulnerability Database

Vulnerability DB	*# CVE Vulnerability Reports*					*# CVEs with Vulnerable APIs*
	Critical	*High*	*Medium*	*Low*	*Sum*	
Ubuntu Security	2,530	12,917	16,317	1,830	33,594	16,750
Debian Security	553	2,130	2,523	255	5,461	2,255
Arch Linux Security	897	5,871	6,303	607	13,678	7,902
Sum†	3,119	18,536	21,291	2,527	45,473	21,384

†The duplicated vulnerabilities in three databases are filtered out.

Table 3: Statistics of 69 real CLV issues in our benchmark

PyPI	*Maven*	*#Vulnerable Libraries*
27 projects; **32** CLV issues	**32** projects; **37** CLV issues	**39**

†Some projects induce more than one CLV issues.

Security [17], and *Arch Linux Security* [2]. In total, we collected vulnerable APIs corresponding to 21,384 vulnerability reports.

The vulnerability metadata collection took four months for three authors of this paper who have over two years vulnerability analysis experience to implement and test. We adopted an effective tool CTAGS [6] to precisely extract the signatures of vulnerable APIs from code commits. More importantly, the three authors conducted daily manual validation for the vulnerability info.

4.2 RQ1: Effectiveness of INSIGHT

Study Methodology. To evaluate INSIGHT's capability in CLV issue detection, we constructed a high-quality benchmark consisting of real CLV issues because there is no related benchmark publicly available. Each CLV issue is a 5-tuple clv = <P_b, Vul, Cp, vr, A>, where P_b is the buggy version of a PyPI or Maven project; Vul denotes a CVE identifier disclosed in the vulnerable C project Cp, vr is the vulnerable version range of Cp affected by Vul, and A is a collection of the concerned vulnerable APIs in Cp. Note that the information of Cp, vr and A can be identified in Vul's CVE report. We collected the benchmark dataset in two steps:

- First, we located a set of projects on GitHub that use Python or Java as primary language, and their commit log messages in code repositories described the CVE identifier of a C project (e.g., *CVE-2016-4658*). We found 341 such project versions. We manually checked the these commits and only kept the project versions that satisfy: (1) their commit log messages explicitly state that the revisions of build scripts aimed to remedy the concerned CVEs; (2) bundling test files (for triggering CLV issues); (3) achieving more than ten *Stars* (popularity). Finally, 232 project versions with CVE-patching commits were retained. We considered these project versions before merging the commits as buggy versions potentially inducing CLV issues.
- Second, we executed the tests bundled with the buggy project versions to validate if they can actually invoke the relevant vulnerable APIs. To this end, we manually inserted log statements into each vulnerable API in C libraries and then checked the outcomes after running tests. Finally, we obtained 69 validated CLV issues where the concerned vulnerable C-APIs can be triggered by the tests bundled with the buggy project versions.

The benchmark construction took over one month for two authors to implement. Table 3 shows the statistics of our benchmark. The collected 69 real CLV issues cover 27 and 32 projects in PyPI and Maven ecosystems, respectively, involving 39 vulnerable C libraries. In the following, we run INSIGHT on these buggy project versions to validate whether it can detect the CLV issues. Specifically, if one of the identified C project versions satisfies the vulnerable version range vr of Cp affected by vulnerability Vul, we consider INSIGHT correctly determining the concerned C library's identity. For a CLV issue involving multiple vulnerable APIs, we consider INSIGHT correctly capturing the issue if it reports one of them.

Results. Table 4 shows our experiment results. INSIGHT reports 61 CLV issues in the 59 benchmark projects. All the reported CLV issues are true positives, where the vulnerable libraries and APIs are correctly identified. In other words, INSIGHT accurately reports 88.4% of the 69 CLV issues in the benchmark. We manually investigated the eight cases that were missed by INSIGHT, and divided them into two categories:

Table 4: Effectiveness of INSIGHT on CLV issue detection

Types \ Stage	Detection Rate	Stage 1	Stage 2	Stage 3
CLV Issues in PyPI Projects	29/32 = 90.6%	32/32	30/32	29/30
CLV Issues in Maven Projects	32/37 = 86.5%	37/37	32/37	30/30
Detection Rate = 61/69 = 88.4%				

***Stage 1**: C Library Extraction; **Stage 2**: C Project Version Tracing; **Stage 3**: Vulnerable C-API Analysis*

- *INSIGHT failed to determine the identities of C libraries.* For seven of the missing cases, INSIGHT obtained low similarity scores (bin^2sim value was below LIBRARIAN' threshold 0.85) when matching the features of C libraries with those of known version instances in our knowledge repository. In the two missing cases of PyPI projects, INSIGHT could not determine the identities of C libraries, because the corresponding C project maintainers only published the vulnerability fix versions to the Linux library repository. Since there is no metadata of the concerned vulnerable versions, our tool failed to determine the library identities via feature matching. For instance, vulnerability *CVE-2018-15686* [11] affects project version `systemd` 227 [49], while only `systemd` 229-249 can be found in *Unbuntu* library repository. In the PyPI project `Ble-driver` 0.16.2 [3] with *CVE-2018-15686*, INSIGHT reported that library `libudev.so` was compiled from project `systemd`, but failed to identify the project version.
  In the five missing cases of Maven projects, by checking the feature lists extracted from C libraries, we found that these client projects mixed their FFI declarations with the referenced C code and recompiled them into `*.so` files. Such recompilation affects our feature matching results, leading to low bin^2sim values. For instance, the Maven project `Emodb` 0.3.1 [20] with *CVE-2019-17543* [14], inserted their *JNI* code into the C library `liblz4.so`. We observed that the original exported functions in the feature list were replaced with the *JNI* functions after the recompilation. As a result, INSIGHT obtained a bin^2sim of 0.46.
- *INSIGHT failed to analyze the accessibility of vulnerable APIs based on parsing FFIs.* The remaining missing case concerns the PyPI project `Swiglpk` [48] that uses *SWIG* FFI [47]. Since the current implementation of INSIGHT does not support the related parsing pattern, it fails to capture the vulnerable C-API accessed by the project. INSIGHT is able to capture the vulnerable C-APIs in 68 out of 69 (98.6%) projects in our benchmark using the parsing patterns designed for *JNI*, *JNA*, *CFFI*, *Cython* and *Ctypes*. This also demonstrates the effectiveness of our parsing patterns.

***Answer to RQ1:** INSIGHT accurately identifies 61 out of 69 (88.4%) CLV issues collected in our benchmark, achieving high detection rates for both PyPI and Maven projects.*

4.3 RQ2: Pervasive Dependency on C Libraries

Study Methodology. For a large-scale empirical study, we collected all the projects' metadata in PyPI and Maven ecosystems using the open-source discovery service *Libraries.io* [38] on its snapshot of March 1, 2022. Table 5 shows the statistics of our collected subjects. We deployed INSIGHT to iteratively analyze 9,090,665 project versions in PyPI and Maven ecosystems, to investigate if their released archives (e.g., `*.Jar`, `*.aar`, `*.whl`) bundled `*.so` files (i.e., directly depending on C libraries). More importantly, our metadata from *Libraries.io* also maps the dependency relationships between project versions, which enables us to explore whether C libraries are introduced into other projects via transitive dependencies.

Table 5: Statistics of subjects in the PyPI and Maven ecosystems

Ecosystem	#Projects	#Project Versions	#Dependency Relationships
PyPI	417,416	3,383,391	12,382,505
Maven	356,718	5,707,274	28,945,967

Table 6: Empirical results of RQ2

Results	PyPI	Maven
# Multilingual Projects	**195,045**/417,416 46.7%	**114,330**/356,718 32.1%
# Projects Directly Depending on C Libraries	**11,402**/195,045 5.8%	**10,227**/114,330 8.9%
# Projects Transitively Depending on C Libraries	**183,643**/195,045 94.2%	**104,103**/114,330 91.1%
# Multilingual Project Versions	**1,383,554**/3,383,391 40.9%	**2,037,098**/5,707,274 35.7%
# Project Versions Directly Depending on C Libraries	**96,505**/1,383,554 7.0%	**110,063**/2,037,098 5.4%
# Project Versions Transitively Depending on C Libraries	**1,287,049**/1,383,554 93.0%	**1,927,035**/2,037,098 94.6%

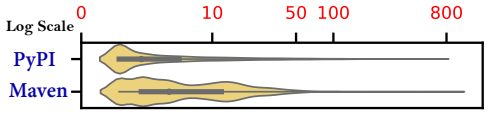

Figure 5: Number of C libraries in each multilingual project version

Results. Table 6 summarizes the statistics of our investigation results. With a longitudinal analysis of 774,134 projects, we identified 195,045 (46.7%) and 114,330 (32.1%) projects that directly or transitively depend on C libraries in the PyPI and Maven ecosystems, respectively, involving 3,420,652 project versions. In fact, 93.0% of multilingual projects transitively use C libraries through a small fraction of pivotal projects that bundle `*.so` files. For example, we observed that a popular project `Pillow` [42] leverages C libraries to provide efficient image processing capabilities. Due to its powerful features, in PyPI, `Pillow` is depended by 46,268 downstream projects, achieving over ten millions of downloads per month. Similarly, project `RocksDB` [45] is developed by *Facebook Database Team* for storing data on flash drives based on C libraries. At present, in Maven, there are 16,649 downstream projects using `RocksDB` to implement Flash and RAM storage. Such pivotal projects introduce C libraries into a significant number of projects in the PyPI and Maven ecosystems via dependency chains, which should arouse practitioners' awareness.

By further inspecting the multilingual projects, we found that in total, there are 87,668 C libraries being introduced into these two ecosystems, as of March 1, 2022. The distribution plot in Figure 5 presents the number of C libraries in each project version. Our results show that the number of dependent C libraries varies from 1 to 790 (i.e., 8.7±23.3). In both PyPI and Maven ecosystems, the projects of machine learning modeling and computer vision toolkits, etc., typically depend on many C libraries to enhance their performances. For example, a popular Python project `PyAV` 9.1.1 [44] bundles 93 `*.so` files. These C libraries originate from

Table 7: Empirical results of RQ3

Results	PyPI	Maven
Projects Directly Depending on Vunlerable C Libraries	**1,556**/11,402 (13.6%)	**1,305**/10,227 (12.8%)
Projects Transitively Depending on Vulnerable C Libraries	**59,118**/183,643 (32.2%)	**20,972**/104,103 (20.1%)
Projects Directly Invoke Vulnerable APIs from C Libraries	**43**/1,556 (2.8%)	**41**/1,305 (3.1%)
Projects Transitively Invoke Vulnerable APIs from C Libraries	**14,213**/59,118 (24.0%)	**7,029**/20,972 (33.5%)
Project Versions Directly Depending on Vulnerable C Libraries	**15,942**/96,505 (16.5%)	**17,312**/110,063 (15.7%)
Project Versions Transitively Depending on Vulnerable C libraries	**381,129**/1,287,049 (29.6%)	**417,269**/1,927,035 (21.7%)
Project Versions Directly Invoke Vulnerable APIs from C Libraries	**662**/15,942 (4.2%)	**870**/17,312 (5.0%)
Project Versions Transitively Invoke Vulnerable APIs from C Libraries	**72,814**/381,129 (19.1%)	**101,113**/417,269 (24.2%)

a leading multimedia framework `FFmpeg` [21] to help decode, encode and transcode audios and videos. A Java project `Scalismo` 4.0.0 [46] used for statistical shape modeling, depends on 203 C libraries, which originate from 41 C projects, including the high performance data management project `Hdf5` [24] and the image processing project `vtk` [52]. Such heavy use of C libraries can increase the projects' attack surface through cross-language invocations.

Answer to RQ2: *Based on a recent snapshot of PyPI and Maven ecosystems, 11,402 (5.8%) PyPI and 10,227 (8.9%) Maven projects directly depend on C libraries. The direct dependencies cause 183,643 (94.2.0%) and 104,103 (91.1%) projects transitively depending on C libraries, respectively, involving 3,420,652 project versions. Most projects tend to introduce multiple C libraries, increasing their attack surface through cross-language invocations.*
Implication: *A small number of popular projects can introduce C libraries to a large number of PyPI and Maven projects through dependency chains.*

4.4 RQ3: Cross-ecosystem vulnerability impacts

Study Methodology. For the projects depending on C libraries identified in RQ2, we applied Insight to diagnosing if they (1) depend on vulnerable C libraries; (2) invoke vulnerable APIs from C libraries. To understand if developers were unaware of such CLV issues, we quantitatively analyzed the time taken by the developers to upgrade a vulnerable C library version in their projects after the fixing release of the library version.

Results. Insight successfully traced the project versions for 65,949 out of 87,668 C libraries (75.2%) that were introduced into PyPI and Maven ecosystems. For 5,162 out of 9,352 identified vulnerable C libraries (55.2%), our vulnerability database collected the vulnerable APIs from their corresponding CVE reports. Based on the above captured vulnerable C libraries and APIs, we summarized the statistics of our investigation results in Table 7. In these two ecosystems, we observed that 33,254 project versions directly depend on vulnerable C libraries. In total, they were transitively used by 798,398 versions of their downstream projects. It is noteworthy that, Insight identified 43 PyPI projects and 41 Maven projects (involving 1,532 project versions) that can directly invoke vulnerable APIs from C libraries. These project versions transitively affect 173,927 versions of downstream projects. The figures may underestimate the real situation since only a limited number of vulnerable APIs were explicitly identified from the CVE reports.

Figure 6(a) plots the distribution of vulnerability counts introduced in each project version. It is noticeable that the vast majority

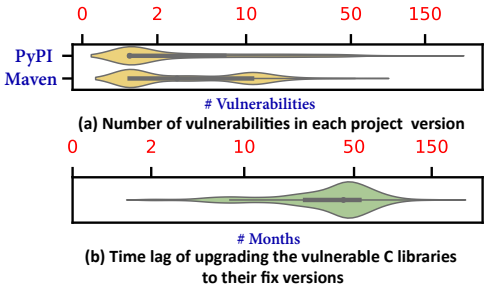

Figure 6: Investigation results of RQ3 (log scale)

of projects contain more than one vulnerability. The number of vulnerabilities in these projects varies from 1 to 151 (i.e., 6.9 ± 11.0). For example, project `pdftopng` [41] depends on four vulnerable C libraries, involving 4 critical-level and 9 high-level CVEs. By parsing the FFIs in `pdftopng`, Insight points out 15 call chains that can access 12 vulnerable APIs.

Among the 2,861 projects directly dependent on vulnerable C libraries, 2,603 of them (91.0%) have not yet remediated the vulnerabilities in their latest versions. We further inspected the 2,218 active projects disseminating new releases in the past year, to analyze the time lag in adopting the fixes of vulnerable C library versions. Figure 6(b) presents the above investigation results (only focusing on vulnerabilities with critical and high severity levels). We observed that 75.0% of C libraries have already remediated the vulnerabilities for more than two years. However, project developers in PyPI and Maven communities probably were unaware of these problems and still depend on the vulnerable C library versions.

Answer to RQ3: *In the PyPI and Java ecosystems, we identified 82,951 multilingual projects (26.8%) directly or transitively using vulnerable C libraries. Among them, 21,326 projects (involving 175,459 versions) can invoke vulnerable APIs from C libraries.*
Implication: *The cross-ecosystem vulnerability impacts may be greatly underestimated since Insight identified project versions for 75.2% of C libraries and located a limited number of the involved CVE reports. Future research may focus on improving the effectiveness of security analyzers to help developers capture the CLV issues.*

4.5 RQ4: Usefulness of Insight

Study Methodology. To further validate the usefulness of diagnosis information generated by Insight for CLV issues, we report them to respective developers. We focused on two types of CLV issues in the PyPI and Maven ecosystems: (1) *API-level CLV issues*: the projects whose latest versions could invoke vulnerable APIs from C libraries; (2) *Library-level CLV issues*: the pivotal projects whose latest versions directly depend on vulnerable C libraries. Specifically, we considered the projects that are directly or transitively used by more than 500 downstream projects as the pivotal ones. We concern the library-level CLV issues in the condition that their corresponding vulnerable APIs cannot be explicitly identified in the CVE reports. In such cases, the pivotal projects using vulnerable C libraries can bring potential security risks to large amounts of downstream projects. We filtered out 178 CLV issues whose associated projects had no maintenance records in the last two years and submitted 226 reports to the active projects' issue trackers.

Ethical Considerations. To avoid spamming the open-source community and developers, we only concerned the pivotal projects with

CLV issues and thoroughly validated the diagnosis info generated by INSIGHT. All the 226 issue reports were submitted in compliance with the projects' contributing guidelines and licenses.

Results. Table 8 summarizes the statistics of our CLV issue reports. 127 reports (56.2%) were quickly confirmed by developers, and 95 confirmed reports (78.4%) have been fixed or are being fixed based on our suggestions. For the remaining 32 confirmed issues (25.2%), developers tried to upgrade vulnerable C libraries but encountered challenges during fixing process. 11 library-level issue reports (4.9%) were rejected because the vulnerable functionalities in C libraries were not actually used by client code after developers' validation. The other reports are still pending likely due to the less active maintenance of the projects.

Feedback on reported CLV issues. The high confirmation rates in the two ecosystems suggest that developers are deeply concerned about the CLV issues induced in their projects and tend to timely resolve the issues. We discuss a few selected cases below.

- *API-level CLV Issues.* INSIGHT reported 41 issues to the projects whose latest versions could invoke the vulnerable C-APIs, achieving a 100% of confirmation rate. 39 (95.1%) were fixed/under fixing by developers. It is noteworthy that, for all the API-level CLV issue reports, we received developers' quick confirmation within two days. The detailed diagnosis information generated by INSIGHT indeed aroused their awareness of the call chains reaching to vulnerable C-APIs. For example, in #199 [32] of `Xmlstarlet`, we pointed out four call chains starting from the FFI declarations in Python client code to the vulnerable C-APIs disclosed in four CVE reports (i.e., *CVE-2019-11068* [13], *CVE-2017-5029* [10], *CVE-2016-2073* [8] and *CVE-2016-3627* [9]). Developers were previously unaware of the risk and commented "*Thanks for the thorough report! I will prepare a new release with a fix this week.*"
- *Library-level CLV Issues.* INSIGHT reported 185 issues to pivotal PyPI and Maven projects whose latest version depend on vulnerable C libraries. Since the involved CVE reports do not explicitly point out the vulnerable APIs, such vulnerabilities can lead to an uncertain attack surface in the ecosystems. 46.5% of these influential projects confirmed our CLV issue reports, developers expressed that they should take actions to avoid bringing potential risks to large amounts of downstream projects in the ecosystems. For example, in #135 [27], the project `cython-hidapi` that implements *Cython* FFI for interfacing Python with C, quickly upgraded their C libraries to fix the CLV issues. Since they had over 70,000 downstream users, the developers took our report seriously and commented that "*This report makes sense. We should fix the issue and publish a new wheel package.*"

The responsive CLV issue reports cover many well-known projects, such as `Mongodb` [40], `Libgit2` [37] and `Eclipse/Sumo` [19], etc. INSIGHT's generated reports arouse the awareness of 95 projects to remediate CLV issues. Their corresponding fix versions could be incorporated by 49,118 downstream projects, significantly reducing the attack surface in PyPI and Maven ecosystems.

Feedback on INSIGHT. Developers also showed great interest in the INSIGHT tool. For example, we reported an issue `#171` [31] to `ParallelSSH`, which aroused project developers' attention. Since `ParallelSSH` provides C-APIs for Python clients to build SSH2 (*Secure Shell*) protocol, they take vulnerabilities seriously. One developer supported our suggestions and left a comment:

"*The diagnosis info is absolutely useful to us! How did you generate the call chains? I would be interested in automated notifications of vulnerabilities in the C libraries we use. Since Python build tools cannot figure out C dependencies for security checking, we rely on people to report any C vulnerabilities.* " Developers' feedback suggests that INSIGHT is useful in warning developers against CLV issues.

> ***Answer to RQ4:*** *The feedbacks indicate that detecting the vulnerability impacts across ecosystems is indeed important to and welcomed by real world developers. The diagnosis information provided by INSIGHT is also useful to help developers diagnose CLV issues in practice.*
> ***Implication:*** *Continuously monitoring the vulnerability impacts across ecosystems is of great values for building healthy open-source communities. Future research may focus on generalizing the technical framework of INSIGHT to other pairwise software ecosystems to help capture CLV issues.*

4.6 RQ5: Fixing Strategies and Challenges

Study Methodology. We performed an in-depth analysis on: (1) the patches of 42 fixed CLV issues; (2) clear fixing plans and relevant discussions in the 38 confirmed CLV issues. We followed an open coding procedure [78], a widely used approach for qualitative research, to categorize the fixing strategies and challenges discussed in our reported issues. Two authors of this paper, who have over three years Java and Python development experience, independently analyzed the fixes and developers' discussions on the issue reports. In order to adjust the taxonomy, they gathered to compare and discuss their results. The conflicts of labeling were discussed during meetings and resolved by the third author.

Results. We observed two fixing strategies from the issue reports.

Fixing Strategy 1: *Upgrading the C direct dependencies that directly or transitively introduce the vulnerabilities (55/80).* This strategy was adopted by 55 reports (68.8%) to resolve CLV issues. For example, in #118 [26] of `Stratega`, the client project directly depended on a vulnerable library `libfreetype.so`, which was originated from C project `Freetype` 2.6.1 containing four CVEs. Following our suggestions, developers incorporated the `libfreetype.so` from the CVE fix version `Freetype` 2.9.1.

Fixing Strategy 2: *Upgrading the target OS of client project (25/80).* We observed that 25 projects (31.3%) adopted this strategy to fix CLV issues, since their referenced vulnerable `*.so` files are automatically bundled by Linux OS during the building process. In such cases, project developers have to upgrade their target OS versions to introduce the vulnerability fix versions of C libraries. For example, in #15 [29] of `Archi`, the client project transitively used a vulnerable C library `liblz4.so` which was bundled by OS `Ubuntu` 16.04. Developers resolved this issue by upgrading its target OS to `Ubuntu` 18.04 to introduce the CVE fix version of `liblz4.so`.

Regarding the fixing challenges, we found clues in 43 CLV issue reports and summarized them as follow:

Challenge 1: Developers have to resolve the dependency conflict issues induced by upgrading dependencies/target OS (10/43). Upgrading libraries/target OS can fix CLV issues but may introduce new dependencies that violate the dependency constrains specified by other C libraries/the build environment. For example, in #6 [34],

Table 8: Statistics of 226 CLV issues reported by INSIGHT

♠ Fixed issues; ♣ Issues that were under fixing; ★ Confirmed issues but facing challenges during fixing process; ■ Rejected issues; ▲ Pending issues.
More detailed information of issue reports is provided on our homepage (**http://insight-clv-detection.com/**).

Types	*Issue Reports*
PyPI *API-level*	xmlstarlet#199♠; capstone-get#1864♠; chiavdf#108♠; depthai#543♠; digital-rf#40♠; electrumsv-secp256kl#5♠; mikecore#21♠; pdftopng#125♠; pyrsb#12♠; python-gdcm#13♠; healpy#758♠; psycopg-binary#262♠; triton#489♠; libsonata#189♠; ligo-skymap#12♣; mpegcoder#4♣; pyfast#155♣; pymssql#753♣; pyopencap#41♣; pytraj#1607♣; pyhdf#54★;
PyPI *Library-level*	hidapi#135♠; rasterio#81♠; opencv-python#646♠; python-casacore#229♠; python-mbedtls#59♠; eclipse-sumo#10456♠; python-gdcm#13♣; healpy#758♣; django-social#1♣; shape-detection#1♣; archi#15♠; Stratega#118♠; robobag#1♣; jdla#1♣; docbarcodes#38♣; ssh2-python#171♣; arbor#1874♣; zelos#140♣; crypto-msg-parser#1♣; fastjet#80★; klayout#1046★; skia-python#175★; smt-switch#299♣; sqlcipher3-binary#12♣; urh#973♣; kongalib#1♣; miller-rabin#1♣; pygit2#1136♣; mmedit#830♣; redlibssh2#1♣; regina#1♣; lpk#62♣; gudhi#599♣; regionmask#347♣; atlite#232★; geocube#100★; rioxarrary#503★; terracotta#261★; mapchete#455★; pytraffic#14183★; sourcery-cli#216★; kivy#7851★; lavavu#99★; medaka#364★; pymeshlab#205★; mip#263★; pysam#1097■; pro-gan-pth#65■; wbia-pyhesaff#9■; wbia-vtool#16■; gphoto2#139■; mixstream#35■; pysurvive#269▲; asmc-preparedecoding#11▲; carla#5289▲; curlyboi#5▲; cyvcf2#239▲; daisykit#12▲; disparity-interpolation#6▲; ecell4-base#495▲; extractcode-libarchive#21▲; imread#40▲; tuplex#90▲; wasmer-compiler-llvm#623▲; mpdaf#25▲; mvdtool#105▲; nd2#48▲; ndsi#77▲; pyapr#54▲; pybbi#21▲; pycreds#81▲; pygroupsig#92▲; pyinvariant#8▲; pylibtermkey#1▲; taxa-sdk#9▲; pysseract#17▲; pytoulbar2#18▲; pyuwsgi#2424▲; pywf#28▲; qif#4▲; quickfix-ssl#390▲; pyzswagcl#79▲; deep-learning-plus#1▲; vp-suite#33▲; object-detector#2▲; manga-scraper#4▲;
Maven *API-level*	JVips#120♠; idml#34♠; pipes#3♠; sparkey#52♠; yosegi#177♠; blacklite#24♠; terse#139♠; isarn#26♠; spark-utils#23♠; rasterframes#583♠; glow#508♠; servicecomb#753♠; bzffmpegcmd#4♠; pdf2htmlEX#59♠; coral#1♠; Indigo#719♣; apng#106♣; cognitive-services#1475♣; liberium#7♣; scalismo#6★;
Maven *Library-level*	libpag#298♠; OpenMLDB#1723♠; BigDL#4517♠; realm#803♠; zippelin#4354♠; finagle#927♣; laurelin#102♣; incubator#83♣; ignite#9974♣; mongo-java-driver#917♣; timeshape#94♣; APE#55♣; etlflow#167♣; smart#511♣; clicktap#14♣; JPPF#39♣; caoyx-rpc#13♣; enos#20♣; wasp#9♣; Orchid#413♣; LibRawFX#2♣; 180protocol#50■; pulsar#15124■; huaweicloud#73▲; djl#1561■; incubator#82♣; alive-detect#2♣; hyperdriver#264★; netflow#82★; druid#12462♣; nebula#78♣; ffsampledsp#14★; FloorPlan#65★; azure-sdk#1090★; recommenders#1701★; jctp#3★; paparazzi#417★; trino#12082★; h5jan#11★; J2V8#581★; FastAsyncWorldEdit#1701★; snowflake#426★; mongo-spark#68★; amazon-codeguru#1★; overwatch#352■; parquet4s#263■; fun-io#21▲; logisland#605▲; embulk#84▲; thrift#16▲; gamioo#1▲; wasp#10▲; aiven#105▲; parquet#8▲; fm-common#41▲; armor#48▲; topleet#1▲; AndroidDocumentScanner#42▲; ebiznext#653▲; featran#619▲; rumble#1192▲; damavis#11▲; octavo#25▲; bitcoin#4263▲; nix-java#80▲; konduit#517▲; ssj#40▲; AudioMixer#8▲; libxmfunsdk#2▲; NIM#3▲; artcodes#4▲; Frankenstein#8▲; RxGalleryFinal#315▲; spectrum#1842▲; qtjambi#11▲; unidbg#423▲; kafka-connect#13▲; tessera#1446▲; Habanero#10▲; LTSample#1▲; MMVideoSDK#1▲; langx#35▲; MMSPlayer#1▲; ClassRoomSDK#1▲; ijkrtspdemo#5▲; oss-allegro#31▲; rovio#26▲; web3-event#36▲; Takin#121▲; hawkbit#78▲; chameleon#9▲; cordova#1▲; moip#144▲; iot#16▲; cf4j#22▲; FFmpegTools#1▲; opensrp#863▲; HeartSaVioR#38▲; fresco#2671▲;

project `scalismo` bundled 203 C libraries, which made it quite difficult to deploy. Library `libjhdf5.so` originating from `Hdf5` 1.8.10, contains 14 CVEs whose vulnerable APIs can be accessed by the Java client code. Project developers tried to upgrade `libjhdf5.so`, but its CVE fix version introduced new dependencies whose specified version constraints were incompatible with the existing ones. `Scalismo` team opened a discussion for addressing the induced dependency conflicts.

Challenge 2: Developers cannot easily figure out which version of C direct dependency/target OS can transitively introduce the vulnerability fixes (20/43). For example, in #54 [33] of `Pyhdf`, the client project transitively depend on a vulnerable C library `libjpeg.so` through the direct dependency `libcpython.so`. Specifically, `libjpeg.so` was compiled from the C project `libjpeg-turbo` 1.3.0 containing two CVEs, whose vulnerable APIs can be invoked by the Python client code. To resolve this issue, developers tried out different versions of `libcpython.so`, but could not figure out a patch that transitively introduced the CVE fix version of `libcpython.so`.

Challenge 3: Developers have to request the Linux community to remediate the vulnerabilities (13/43). In the cases that projects' compatible target OS versions have not bundled the CVE fix versions of C libraries, developers have to request the Linux distribution to remediate such vulnerable C libraries. For example, in #1046 [25] of `Klayout`, the project automatically introduced a vulnerable C library through its target OS `CentOS7` during building process. Developers failed to identify a compatible OS version having incorporated the vulnerability fixes. As a result, they forwarded our issue report to the Linux community (see report [22]), seeking for assistance to remediate the vulnerable libraries.

> ***Answer to RQ5:*** *We observed that the vulnerable C libraries can be introduced as a client project' direct dependencies/transitive dependencies, or even automatically bundled by the target OS during the building process. According to the different forms of introducing C libraries, project developers determine their fixing strategies accordingly.*
> ***Implication:*** *Future research may focus on addressing the three fixing challenges to design automated approaches for repairing CLV issues.*

5 THREATS TO VALIDITY

Internal Validity. The internal threat relates to the identification of C libraries . We leverage an existing technique LIBRATIAN to extract main features inherent to C library versions for matching the similarity with those in our knowledge repository. However, these features may lead to misidentifications for consecutive versions. The consecutive versions with minor revisions may not change, add, or remove exported symbols. To mitigate these threats, we significantly reduced the similarity matching scope in our knowledge repository for post analysis to a few candidate versions.

Conclusion Validity. In our study, we focus on the C libraries targeting Linux OS in the form of `*.so` files, because the community provides centralized library repositories for mainstream Linux distributions and continuous services for collecting vulnerability reports of C libraries. The above metadata enables the implementation of our automated tool, but poses threats to the conclusion validity. Future studies may improve our knowledge repository and feature extraction approaches for adapting to C libraries targeting Windows OS (in the form of `*.dll` files).

6 RELATED WORK

Multilingual program analysis. Khomh et al. [56, 57] defined a catalog of design smells related to multilingual projects and provide empirical evidence on their severity and impact on software quality. Tan et al. [90] empirically studied on a large collection of Python projects to understand the interaction inefficiencies between Python code and native libraries written in C/C++/Fortran. Bai et al. [60] proposed a bi-directional dynamic taint tracking method, BRIDGETAINT, to detect cross-language privacy leaks and code injection attacks in hybrid apps using JavaScript bridges. Lee et al. [77] proposed an approach to analyze the *JNI* invocations between Java and C, in order to extract semantic summaries from multilingual programs. Li et al. [79] developed a tool NATIDROID, to facilitate cross-language analysis on the Android system, and extract the permission-API protection mappings from the native libraries on Android apps. The study most relevant to our work is done by Almanee et al. [58]. They propose a technique LIBRARIAN, which leverages different features extracted from C libraries to determine their project versions. Our INSIGHT incorporates LIBRARIAN to trace to the identities of C libraries and provides a turn-key solution to diagnose CLV issues. To adopt the LIBRARIAN for CLV issue detection in PyPI and Maven ecosystems, INSIGHT constructs a large-scale knowledge repository by collecting the metadata of millions of C libraries with their known versions.

Software ecosystems. Decan et al. [67, 69] conducted a quantitative empirical analysis of the similarities and differences between the evolution of package dependency networks for seven software ecosystems, including *Cargo*, *CPAN*, etc. Approaches [61, 62] pointed out the fragility of software ecosystems and gave insights on the challenges software developers face. They examined the *Eclipse*, *CPAN* and *npm* ecosystems, focusing on what practices cause API breakages. In work [55, 55, 76], researchers conducted empirical investigations with millions of *npm* packages and suggested that *npm* developers should be careful about the selection of packages and how to keep them updated. Kula et al. [76, 91] suggested developers to be aware of how sensitive their third-party dependencies were to critical changes in the software ecosystem. Jafari et al. [73] conducted surveys with practitioners, to identify and quantify seven dependency smells with varying degrees of popularity and investigate why smells are introduced. Wittern et al. [92] investigated the evolution of *npm* using metrics such as dependencies between packages, download count, and usage count in JavaScript applications. Shariffdeen et al. [89] empirically studied the bug fix issues in the Linux ecosystem, and proposed a patch backporting technique transfering patches from the mainline version of Linux into older stable versions. To the best of our knowledge, our work is the first attempt to study the vulnerability impacts across multiple software ecosystems.

Vulnerable dependencies. In approaches [70] and [85], researchers empirically studied the evolution and decay of vulnerabilities in source code, and found that due to the code reuse and third-party libraries, most vulnerabilities are recurring. Cox et al. [66] introduced metrics to qualify the "dependency freshness" of software projects, to understand the relations between outdated dependencies and vulnerabilities based on an industry benchmark. Studies [65, 68, 71, 96] investigated millions of *npm* packages to analyze how and when these vulnerabilities were discovered and fixed, and to what extent they affected other packages. Zimmermann [96] pointed out a small proportion of individual vulnerable packages could impact large parts of the entire ecosystem. Whereas approaches [59, 63, 64, 72, 80–82, 86, 95] studied the evolution and impacts of vulnerable packages in specific communities, including Firefox, and Docker Images, Android, etc. Recent studies [83, 84, 87, 88, 94] were proposed to address the over-estimation problem for reporting the vulnerable dependencies. They provided fine-grained assessment approaches using static or dynamic analysis to determine whether the located vulnerable code would be reachable. In this paper, we conducted a thorough study to investigate whether the Python or Java code can invoke vulnerable APIs in C libraries, by parsing diversified FFIs.

7 CONCLUSION AND FUTURE WORK

In this paper, we conducted a large-scale empirical study to understand the status quo of vulnerability impacts across ecosystems. The study also leads to the development of the first automated tool, Insight, which provides a turn-key solution to help practitioners capture the CLV issues. Our study arouses the awareness of vulnerable C libraries in popular ecosystems and presents related analysis results. In the future, we plan to generalize the technical framework of Insight to other software ecosystems and involve various programming languages into our empirical study.

ACKNOWLEDGMENTS

The authors express thanks to the anonymous reviewers for their constructive comments. The work is supported by the National Natural Science Foundation of China (Grant Nos. 62141210, 61932021, 61902056), the Hong Kong RGC/GRF grant 16207120, MSRA grant, ITF grant (MHP/055/19, PiH/255/21), Research Grants Council (RGC) Research Impact Fund under Grant R5034-18, Shenyang Young and Middle-aged Talent Support Program (Grant No. ZX20200272), Open Fund of State Key Lab. for Novel Software Technology, Nanjing University (KFKT2021B01), and 111 Project (B16009).

REFERENCES

[1] 2022. Arch Linux. https://archlinux.org/. (2022). Accessed: 2022-03-01.
[2] 2022. Arch Linux Security. https://security.archlinux.org/issues/all. (2022). Accessed: 2022-03-01.
[3] 2022. Ble-driver. https://github.com/NordicSemiconductor/pc-ble-driver-py. (2022). Accessed: 2022-03-01.
[4] 2022. CFFI. https://cffi.readthedocs.io/en/latest/. (2022). Accessed: 2022-03-01.
[5] 2022. CFlow, is part of the GNU project, which analyzes a collection of C source files and prints the call chains between functions. https://savannah.gnu.org/projects/cflow/. (2022). Accessed: 2022-03-01.
[6] 2022. Ctags tool. http://ctags.sourceforge.net/. (2022). Accessed: 2022-03-01.
[7] 2022. Ctypes. https://docs.python.org/3/library/ctypes.html. (2022). Accessed: 2022-03-01.
[8] 2022. CVE-2016-2073. https://nvd.nist.gov/vuln/detail/CVE-2016-2073. (2022). Accessed: 2022-03-01.
[9] 2022. CVE-2016-3627. https://nvd.nist.gov/vuln/detail/CVE-2016-3627. (2022). Accessed: 2022-03-01.
[10] 2022. CVE-2017-5029. https://nvd.nist.gov/vuln/detail/CVE-2017-5029. (2022). Accessed: 2022-03-01.
[11] 2022. CVE-2018-15686. https://nvd.nist.gov/vuln/detail/CVE-2018-15686. (2022). Accessed: 2022-03-01.
[12] 2022. CVE-2018-18557. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-18557. (2022). Accessed: 2022-03-01.
[13] 2022. CVE-2019-11068. https://nvd.nist.gov/vuln/detail/CVE-2019-11068. (2022). Accessed: 2022-03-01.
[14] 2022. CVE-2019-17543. https://nvd.nist.gov/vuln/detail/CVE-2019-17543. (2022). Accessed: 2022-03-01.
[15] 2022. Cython. https://cython.readthedocs.io/en/latest/src/tutorial/clibraries.html. (2022). Accessed: 2022-03-01.
[16] 2022. Debian. https://www.debian.org/. (2022). Accessed: 2022-03-01.
[17] 2022. Debian Security. https://lists.debian.org/debian-security-announce/. (2022). Accessed: 2022-03-01.
[18] 2022. Dependabot. https://github.com/dependabot. (2022). Accessed: 2022-03-01.
[19] 2022. Eclipse/Sumo. https://www.eclipse.org/sumo/. (2022). Accessed: 2022-03-01.
[20] 2022. Emodb. https://github.com/bazaarvoice/emodb. (2022). Accessed: 2022-03-01.
[21] 2022. FFmpeg. https://ffmpeg.org/. (2022). Accessed: 2022-03-01.
[22] 2022. The forwarded issue report in Linux community. https://github.com/pypa/manylinux/issues/1302. (2022). Accessed: 2022-03-01.
[23] 2022. Guidelines for naming C libraries. https://tldp.org/HOWTO/Program-Library-HOWTO/shared-libraries.html#AEN46. (2022). Accessed: 2022-03-01.
[24] 2022. Hdf5. https://www.hdfgroup.org/solutions/hdf5. (2022). Accessed: 2022-03-01.
[25] 2022. Issue#1046 of KLayout. https://github.com/KLayout/klayout/issues/1046. (2022). Accessed: 2022-03-01.
[26] 2022. Issue#118 of Stratega. https://github.com/GAIGResearch/Stratega/issues/118. (2022). Accessed: 2022-03-01.
[27] 2022. Issue#135 of cython-hidapi. https://github.com/trezor/cython-hidapi/issues/135. (2022). Accessed: 2022-03-01.
[28] 2022. Issue#14623 of Incubator. https://github.com/apache/incubator-mxnet/pull/14623. (2022). Accessed: 2022-03-01.
[29] 2022. Issue#15 of Archi. https://github.com/whtsky/archi/issues/15. (2022). Accessed: 2022-03-01.
[30] 2022. Issue#167 of Etlflow. https://github.com/tharwaninitin/etlflow/issues/167. (2022). Accessed: 2022-03-01.
[31] 2022. Issue#171 of ParallelSSH. https://github.com/ParallelSSH/ssh2-python/issues/171. (2022). Accessed: 2022-03-01.

[32] 2022. Issue#199 of Xmlstarlet. https://github.com/dimitern/xmlstarlet/issues/199. (2022). Accessed: 2022-03-01.
[33] 2022. Issue#54 of Pyhdf. https://github.com/fhs/pyhdf/issues/54. (2022). Accessed: 2022-03-01.
[34] 2022. Issue#6 of Scalismo. https://github.com/unibas-gravis/scalismo-native/issues/6. (2022). Accessed: 2022-03-01.
[35] 2022. Java Native Access(JNA). http://java-native-access.github.io/jna/5.10.0/javadoc/overview-summary.html. (2022). Accessed: 2022-03-01.
[36] 2022. Java Native Interface(JNI). https://docs.oracle.com/en/java/javase/17/docs/specs/jni/index.html. (2022). Accessed: 2022-03-01.
[37] 2022. Libgit2. https://libgit2.org/. (2022). Accessed: 2022-03-01.
[38] 2022. Libraries.io. https://libraries.io/. (2022). Accessed: 2022-03-01.
[39] 2022. Maven. https://maven.apache.org/. (2022). Accessed: 2022-03-01.
[40] 2022. Mongodb. https://www.mongodb.com/. (2022). Accessed: 2022-03-01.
[41] 2022. Pdftopng. https://github.com/vinayak-mehta/pdftopng. (2022). Accessed: 2022-03-01.
[42] 2022. Pillow. https://github.com/python-pillow/Pillow. (2022). Accessed: 2022-03-01.
[43] 2022. Pip. https://packaging.python.org/en/latest/key_projects/#pip. (2022). Accessed: 2022-03-01.
[44] 2022. PyAV. https://github.com/PyAV-Org/PyAV/. (2022). Accessed: 2022-03-01.
[45] 2022. RocksDB. https://github.com/facebook/rocksdb. (2022). Accessed: 2022-03-01.
[46] 2022. Scalismo. https://github.com/unibas-gravis/scalismo-native. (2022). Accessed: 2022-03-01.
[47] 2022. SWIG. https://swig.org/. (2022). Accessed: 2022-03-01.
[48] 2022. Swiglpk. https://pypi.org/project/swiglpk/. (2022). Accessed: 2022-03-01.
[49] 2022. Systemd. https://systemd.io/. (2022). Accessed: 2022-03-01.
[50] 2022. Ubuntu. https://ubuntu.com/. (2022). Accessed: 2022-03-01.
[51] 2022. Ubuntu Security. https://ubuntu.com/security/cve. (2022). Accessed: 2022-03-01.
[52] 2022. Vtk. https://vtk.org/. (2022). Accessed: 2022-03-01.
[53] 2022. Wheel unpack. https://wheel.readthedocs.io/en/stable/reference/wheel_unpack.html. (2022). Accessed: 2022-03-01.
[54] 2022. Zipfile. https://docs.python.org/zh-cn/3/library/zipfile.html. (2022). Accessed: 2022-03-01.
[55] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 385–395.
[56] Mouna Abidi, Moses Openja, and Foutse Khomh. 2020. Multi-language design smells: A backstage perspective. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 615–618.
[57] Mouna Abidi, Md Saidur Rahman, Moses Openja, and Foutse Khomh. 2021. Are multi-language design smells fault-prone? an empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–56.
[58] Sumaya Almanee, Arda Ünal, Mathias Payer, and Joshua Garcia. 2021. Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1347–1359.
[59] Hamid Bagheri, Jianghao Wang, Jarod Aerts, Negar Ghorbani, and Sam Malek. 2021. Flair: efficient analysis of Android inter-component vulnerabilities in response to incremental changes. *Empirical Software Engineering* 26, 3 (2021), 1–37.
[60] Junyang Bai, Weiping Wang, Yan Qin, Shigeng Zhang, Jianxin Wang, and Yi Pan. 2018. BridgeTaint: a bi-directional dynamic taint tracking method for JavaScript bridges in android hybrid applications. *IEEE Transactions on Information Forensics and Security* 14, 3 (2018), 677–692.
[61] Christopher Bogart, Christian Kästner, and James Herbsleb. 2015. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE, 86–89.
[62] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.
[63] Mircea Cadariu, Eric Bouwers, Joost Visser, and Arie van Deursen. 2015. Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 516–519.
[64] Xiao Cheng, Haoyu Wang, Jiayi Hua, Guoai Xu, and Yulei Sui. 2021. DeepWukong: Statically detecting software vulnerabilities using deep graph neural network. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–33.
[65] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021), 1–28.
[66] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. 2015. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 109–118.
[67] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2–12.
[68] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 181–191.
[69] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.
[70] Massimiliano Di Penta, Luigi Cerulo, and Lerina Aversano. 2009. The life and death of statically detected vulnerabilities: An empirical study. *Information and Software Technology* 51, 10 (2009), 1469–1484.
[71] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. 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*. 2169–2185.
[72] Jun Gao, Li Li, Pingfan Kong, Tegawendé F Bissyandé, and Jacques Klein. 2019. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability* 70, 1 (2019), 212–230.
[73] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2021. Dependency smells in Javascript projects. *IEEE Transactions on Software Engineering* (2021).
[74] Jiyong Jang, Abeer Agrawal, and David Brumley. 2012. ReDeBug: finding unpatched code clones in entire os distributions. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 48–62.
[75] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 595–614.
[76] Raula Gaikovina Kula, Ali Ouni, Daniel M German, and Katsuro Inoue. 2017. On the impact of micro-packages: An empirical study of the npm javascript ecosystem. *arXiv preprint arXiv:1709.04638* (2017).
[77] Sungho Lee, Hyogun Lee, and Sukyoung Ryu. 2020. Broadening horizons of multilingual static analysis: semantic summary extraction from C code for JNI program analysis. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 127–137.
[78] Sarah Lewis. 2015. Qualitative inquiry and research design: Choosing among five approaches. *Health promotion practice* 16, 4 (2015), 473–475.
[79] Chaoran Li, Xiao Chen, Ruoxi Sun, Jason Xue, Sheng Wen, Muhammad Ejaz Ahmed, Seyit Camtepe, and Yang Xiang. 2021. NatiDroid: Cross-Language Android Permission Specification. *arXiv preprint arXiv:2111.08217* (2021).
[80] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. 2017. An empirical study on android-related vulnerabilities. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2–13.
[81] Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. 2011. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *International Symposium on Engineering Secure Software and Systems*. Springer, 195–208.
[82] Israel J. Mojica, Meiyappan Nagappan, Bram Adams, Thorsten Berger, Steffen Dienst, and Ahmed E. Hassan. 2016. On Ad Library Updates in Android Apps. *IEEE Software* 33, 2 (2016), 74–80.
[83] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–10.
[84] Ivan Pashchenko, H. Plate, Serena Elisa Ponta, Antonino Sabetta, and F. Massacci. 2020. Vuln4Real: A Methodology for Counting Actually Vulnerable Dependencies. *IEEE Transactions on Software Engineering* (2020), 1–1.
[85] Nam H Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. 2010. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 447–456.
[86] Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. 2019. Fixing of security vulnerabilities in open source projects: A case study of apache http server and apache tomcat. In *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 68–78.
[87] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 411–420.
[88] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 449–460.
[89] Ridwan Shariffdeen, Xiang Gao, Gregory J Duck, Shin Hwei Tan, Julia Lawall, and Abhik Roychoudhury. 2021. Automated patch backporting in Linux (experience paper). In *Proceedings of the 30th ACM SIGSOFT International Symposium on*

Software Testing and Analysis. 633–645.

[90] Jialiang Tan, Yu Chen, Zhenming Liu, Bin Ren, Shuaiwen Leon Song, Xipeng Shen, and Xu Liu. 2021. Toward efficient interactions between Python and native libraries. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1117–1128.

[91] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering*. 511–522.

[92] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*. 351–361.

[93] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2021. TRACER: Finding Patches for Open Source Software Vulnerabilities. *arXiv preprint arXiv:2112.02240* (2021).

[94] Rodrigo Elizalde Zapata, Raula Gaikovina Kula, Bodin Chinthanet, Takashi Ishio, Kenichi Matsumoto, and Akinori Ihara. 2018. Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm JavaScript packages. In *International Conference on Software Maintenance and Evolution (ICSME)*. 559–563.

[95] Ahmed Zerouali, Valerio Cosentino, Tom Mens, Gregorio Robles, and Jesus M Gonzalez-Barahona. 2019. On the impact of outdated and vulnerable javascript packages in docker images. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 619–623.

[96] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small world with high risks: A study of security threats in the npm ecosystem. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 995–1010.