LiScopeLens: An Open-Source License Incompatibility Analysis Tool Based on Scope Representation of License Terms

Ziang Liu[1†], Xin Liu[1†*], Yingli Zhang[1], Zihao Zhang[1], Song Li[2], Weina Niu[3], Qingguo Zhou[1], Rui Zhou[1*], Xiaokang Zhou[4]
[1] Lanzhou University, {liuza20, bird, 120220909270, zhzihao2023, zhouqg, zr}@lzu.edu.cn
[2] Zhejiang University, songl@zju.edu.cn
[3] University of Electronic Science and Technology of China, vinusniu@uestc.edu.cn
[4] Kansai University, zhou@kansai-u.ac.jp

***Abstract*—Open-source software has emerged as a pivotal force in the advancement of information technology. Robust open-source compliance governance is essential for the sustainable and healthy growth of both open-source software and its communities. License incompatibility analysis, in particular, represents a critical challenge hindering the progress of open-source software. Traditional methods of incompatibility analysis often fail to account for diverse usage scenarios or are tailored to a limited subset of scenarios. This limitation obstructing their ability to handle the intricate compatibility arising from varied programming language interactions, leading to a high false positives. Our study embarks from an examination of license exceptions, delving into the incompatibility analysis challenges through extensive empirical research on these exceptions. We discovered that the majority of exceptions are, in fact, detectable. Leveraging this empirical insight, our research further develops the license compatibility analysis model by introducing a new, refined legal terminology representation alongside a novel method for license compatibility reasoning. This approach begins with modeling different scenarios to represent license compatibility variably. Furthermore, based on these modeling outcomes, we have designed and implemented LiScopeLens, a tool capable of discerning dependency behaviors for granular compatibility assessment, starting with binary dependencies. Our experimental findings affirm that LiScopeLens proficiently determines the license compatibility status of open-source software across various usage scenarios, demonstrating its significant practical utility.**

***Index Terms*—Open Source Software, Open Source Compliance, License, Incompatibility Detection**

I. Introduction

Open-source compliance governance is crucial for the sustainable and healthy development of the open-source community, with license compliance being particularly critical. The growing size of open-source software, as shown by a 27% increase in projects hosted on GitHub in 2023 [1], and its crucial role in driving the development of information technology [2], [3], highlight the importance of license compliance for the long-term viability of the open-source community. Misunderstandings or incorrect applications of open-source licenses by enterprises or individuals can lead to legal risks [4]. These risks often arise from dependencies on third-party components with conflicting licenses or changes in their licensing terms. Numerous studies and real-world cases have documented the severe consequences of non-compliance, which range from simple disagreements to protracted legal battles [5], [6].

Software reliability traditionally denotes software's capability to run as anticipated under predetermined conditions for a specified duration. This field, encompassing risk management throughout the software lifecycle, faces new challenges due to the complexities of modern software supply chains [7]–[11]. Increasingly, open-source security incidents and compliance issues arise from the extensive incorporation of third-party dependencies and open-source licenses, posing threats to sustainable open-source development [12], [13]. As a reliability engineering component, open-source license compliance can enhance open-source software's resilience, underscoring the need for practical license compatibility tools. Recent research has thus aimed at creating accurate and efficient mechanisms to face these compliance challenges.

License compatibility checking includes two main steps: identifying the license used in a project [14], [15] and assessing its compatibility [16]. Historically, most research has focused on a limited set of widely-used licenses, relying on predefined compatibility outcomes [17], [18]. Recent advancements have automated the compatibility analysis by treating licenses as a collection of legal terms [19]. For instance, *LiDetector* has significantly improved the handling of custom licenses by automating the extraction of legal terms. Despite these advancements, the ability to fully capture the meanings of complex and ambiguous legal terms remains limited [20]. The compatibility of licenses can significantly vary based on the usage scenarios of the authorized works. Prior research has often been limited by assumptions within specific scenarios, such as specific software development or distribution models, which hinders the derivation of precise compatibility conclusions from the employed software components.

∗ Xin Liu and Rui Zhou are corresponding authors.
† These authors contributed equally in this work.

To solve the above problems, our research starts with an empirical study focusing on large-scale complex open-source software in terms of license compatibility. Specifically, we attempt to answer the following questions:

① What gaps and challenges exist between existing license compatibility tools and actual license compatibility detection?

② How can a license compatibility checking tool be adapted to complex scenarios, such as large open-source software with multi-programming language?

This paper draws inspiration from prevalent issues and methodologies in compatibility analysis. We have refined the existing compatibility analysis model to address the complexity of determining license compatibility across diverse scenarios. Our comprehensive approach encompasses the following components:

- **Scope Representation of License Terms (SRLT)**: This novel method models the **effective scope** of license terms, especially under ambiguous declarations. **SRLT** aids in the efficient management of various scenario-specific license exceptions.
- **Compatibility Reasoning Analysis**: Leveraging **SRLT** and a redefined concept of compatibility, our method automatically constructs a compatibility knowledge graph for any given SRLT set of a license. This graph serves as a valuable tool for subsequent detection in different scenarios.
- **Scenarios Sniffer and Detection Module**: The sniffer senses usage scenarios from projects and verifies whether they satisfy compatible conditions. Later, the detection Module evaluates the project's compatibility according to the compatibility knowledge graph and usage scenarios.

The primary contributions of this paper are as follows:

1. Our extensive empirical studies on license exceptions uncover a major challenge in license compatibility analysis: sufficient modeling of usage scenarios. We show that a detailed examination of license exceptions is possible and valuable.
2. Based on our findings, we introduce a new representation called "Scope Representation of License Terms" to capture legal terminology and reasoning about license compatibility. This method accurately models the complex conditions that determine license term applicability.
3. We have developed LiScopeLens, a tool to detect dependency behaviors and conduct thorough compatibility checks for C/C++ usage scenarios. This work has been made open-source[1].

II. Background

A. open-source License

An open-source license grants the rights to use, modify, and distribute software, along with specific obligations and restrictions. The Linux Foundation lists over 700 licenses, each uniquely identified by a with an unique Software Package Data Exchange (SPDX) identifier, which is a key component of the SPDX specification designed to help distinguish every license. In this paper, we refer to licenses by their SPDX identifiers.

Software licenses can be classified based on their location within a project:

- **Project License**: These licenses, defined by the authors or maintainers, are located in **LICENSE** files or equivalent documentation at the project's root.
- **File License**: Licenses within a project's code files.

Another classification is based on the method of declaration:

- **Embedded License**: The license text is directly embedded within the source code. Common examples include MIT and BSD-family licenses.
- **Referencing License**: This license type includes a link to the original license text, authorizing code use. A common example is the GPL license.

Regarding restriction levels, open-source licenses are categorized as either Permissive or Copyleft [21]. Copyleft licenses prevent the software from being proprietary, whereas Permissive licenses do not. Platforms such as tldrlegal[2], choosealicense[3], and Openeuler [22] represent licenses through accessible terms like **Commercial Use**, **Same License**, and **Patent Use**, thus lowering barriers to understanding for developers and facilitating automated analysis. However, the complexity of these terms limits the effectiveness of detailed automated analysis. In addition, the coexistence of multiple and custom licenses within projects also complicates their management.

B. Conflicts in open-source License

According to previous work, the compatibility of open source licenses is determined: if license l_a is compatible with license l_b, then l_b may be combined or relicensed without breaching the terms of l_a. If not, then l_i and l_j are deemed to be in conflict [23], [24]. Typically, Permissive licenses can be compatible with stricter ones [25], [26]. However, this is only sometimes the case. For example, although l_b may be more restrictive than l_a and confer no extra rights, specific licenses, such as GPL-2.0 and GPL-3.0, may still be incompatible. Identifying license conflicts requires substantial legal expertise to achieve reliable outcomes.

In enhancing security compliance within the open-source software supply chain, automated tools for detecting license compatibility have become increasingly necessary. Previous works have combined expert knowledge with license detection tools to determine compatibility [12], [27], but the license exceptions and the customization of licenses compromise their reliability. Recent developments [16], [20], [28] have leveraged structured license data to enhance these tools' analytical capabilities. However, the limited granularity of the constructed data and insufficient depth in exploring compatibility scenarios lead to these tools often missing the detection of potential conflicts that do not surface explicitly [29]–[31].

[1]https://gitee.com/openharmony-sig/compliance_license_compatibility/

[2]https://tldrlegal.com/

[3]https://choosealicense.com/

C. Motivating Examples

In this study, we examine the persistent challenges of compliance in large open-source environments. Current methods for analyzing software licenses fail to identify compatibility issues across various usage scenarios accurately. For example, is there a need for compatibility assessment across different projects? What interoperability methods make license compatibility assessments necessary when file-level and project-level licenses coexist? Following a systematic license analysis, we engaged with legal experts from the open-source community to address compatibility governance. This collaboration helped us identify examples demonstrating the difficulties in current compatibility analysis and why existing tools are not practically useful.

1) Exception Terms and Custom License: License incompatibility is complex due to varied legal rules and scenarios and because these provisions evolve over time. Users frequently alter licenses, either by appending conditions to restrictive ones or crafting new ones, to safeguard their rights and clarify legal uncertainties. Such modifications can significantly impact compatibility with other licenses. The presence of exception clauses and custom licenses makes it impossible to develop universally applicable compatibility tools based solely on empirical data. Previous studies, such as [16], [20], [28], focused on semi-structured license clause features but did not systematically address these exceptions and custom clauses.

2) Automated Reasoning for Compatibility: Due to exception clauses and custom licenses, dynamic compatibility reasoning is essential for automated analysis. The structured license description methodology can accurately determine compatibility between two licenses based on a set of characteristics encompassing rights, obligations, and constraints. Despite significant efforts in license compatibility research, achieving reliable automated reasoning from structured licenses remains challenging. In the industry, organizations like CreativeCommons[4], SPDX[5], and openeuler [22] have taken significant steps to structure license descriptions. However, a fully effective structured scheme for automated compatibility reasoning has not been established under the nuances of license compatibility. For example, limitation by structured granularity and reasoning rules, the compatibility detection tool by Xu et al. [20] fails to recognize conflicts caused by the cc-by-nd-4.0 license.

3) License Conditional Compatibility: License compatibility is inherently directional and may exhibit what we term "conditional compatibility." Directional compatibility often indicates potential conflicts between licenses, which usually arise in one license or can be mitigated under certain conditions. For instance, a project under Apache-2.0 can only incorporate LGPL-2.1-only licensed work through dynamic linking due to the former's patent licensing and termination clauses, which otherwise conflict with LGPL's commitment to software freedom. Additionally, the compatibility of a license may vary based on the project type. For example, C/C++ projects, which bundle different licenses in binary distributions, are more prone to compatibility issues than Python projects. Therefore, assessing license conflicts requires consideration of the specific usage context. Current literature and tools often provide an ambiguous definition of license compatibility, and it is necessary to address its conditional and directional nature systematically. The osdal-matrix's findings [32] illustrate directional compatibility differences but need more practical application for real-world assessment.

In conclusion, it is essential to comprehend the nuances of license compatibility across diverse usage scenarios to develop practical license compatibility testing tools.

III. Empirical Study

A. Reseach Questions

This empirical study aims to comprehend the limitation of prior knowledge of license compatibility and leverage license exceptions to explore the impact of usage scenarios. Most current detection tools rely, in particular, on preexisting knowledge of license compatibility. We seek to assess whether these preexisting conclusions adequately support practical scenario detections, pinpoint the license compatibilities needing manual intervention, and determine the number of license combinations identifiable via automated methods based on current understanding. To this end, we present the following research questions centered around license exceptions:

- **RQ1:** Are the conclusions drawn about license compatibility adequately usable in practice? Additionally, are there established mechanisms for managing exceptions to these licenses?
- **RQ2:** How common are exception clauses in the wider open-source community and large-scale open-source system projects?
- **RQ3:** What proportion of real-world license exception compatibility issues could theoretically be identified by automated methods?

B. Study Objects

For **RQ1**, we conducted a comparative analysis of existing resources, including Openeuler's compatibility table[6], LCVCM [18], and OSADL-matrix [32]. We evaluated their usability primarily based on their ability to account for conditional compatibility, such as the LGPL-2.1-only license under dynamic linking conditions. Additionally, we examined whether these resources consider various usage scenarios, including different programming languages and distribution methods.

In addressing **RQ2**, we engage with the OpenHarmony[7] project and a large dataset from Software Heritage [33] to survey and analyze the prevalence and types of exception licenses. We rely on Software Heritage for direct provision of scancode [34] results. For OpenHarmony, we utilize the

[4]https://wiki.creativecommons.org/wiki/License_RDF
[5]https://github.com/spdx/license-list-data
[6]https://compliance.openeuler.org/compatiableTable
[7]https://www.openharmony.cn/

scancode tool to conduct a detailed license analysis across its entire codebase and selected projects.

For **RQ3**, we focus on 66 specific exception licenses listed by the SPDX organization, exploring whether their exception conditions can be automatically detected. This task is performed in collaboration with legal experts and active members of the open-source community.

C. Methods and Results

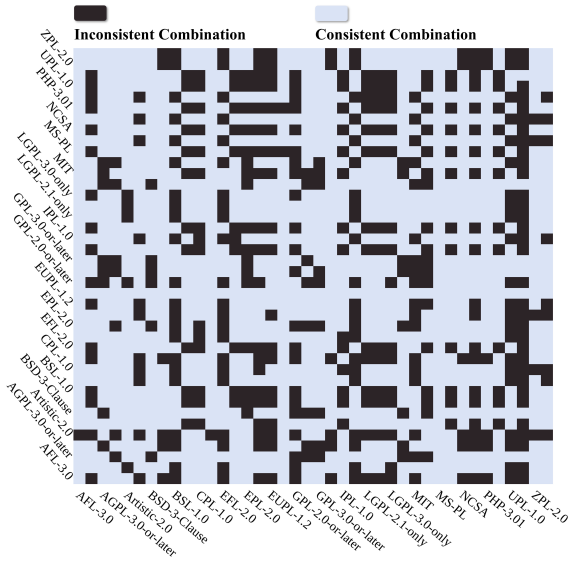

Fig. 1. Heatmap of inconsistent distribution of license compatibility.

1) RQ1: In our study, we evaluated the compatibility of 120 licenses and identified conflicts in 40 of them. The Fig 1 illustrates our findings: The horizontal and vertical axes in the figure represent licenses, with black blocks indicating inconsistent prior knowledge of license compatibility among the three research objects and light blocks indicating consistency. We discovered identified inconsistencies in 496 out of 14,280 possible license combinations. Notably, 4,450 combinations were excluded due to the lack of relevant prior knowledge on the part of a specific research subject.

In addition to these inconsistencies, the discussion on licenses with scenario-dependent compatibility needs to be revised. Conclusions drawn from narrowly defined scenarios frequently offer limited utility. For instance, advice for combinations like lcv-cm and osdal-matrix is vaguely summarized as 'Check dependency.' In contrast, while Openeular's investigation into 48 licenses, considering source code and library interoperability, yields specific compatibility conclusions, it overlooks a systematic examination of license exceptions.

2) RQ2: To improve our understanding of the license exceptions in real-world cases, we conducted a detailed analysis of large open-source systems like OpenHarmony. We used Scancode, a renowned license scanning tool, to track and evaluate the distribution of licenses, focusing on those with exception clauses and potential incompatibilities.

TABLE I
THE LICENSE EXCEPTION DISTRIBUTION STATISTICS

Data Sources	Licenses		Exceptions		Non-standard
	Total	Type	Total	Type	Exceptions
OpenHarmony	13819	135	190	20	196
Software Heritage	7503060	1573	66800	130	302044

This study analyzed 972 repositories within the OpenHarmony open-source community, focusing on the distribution of open-source licenses. The findings, detailed in Table I, reveal the frequency of standard licenses and exceptions. In addition, our analysis extends to non-standard license exceptions, identifying their occurrence in the dataset. In OpenHarmony's repository, standard license exceptions constitute 1.37% of all licensed documents, a figure that drops to 0.89% in the broader software heritage dataset. When accounting for non-standard exceptions, these numbers increase to 2.79% and 4.92%, respectively. This variation reflects the differing license restrictions and the scope of repositories analyzed.

During the extended development and maintenance phases of large systems, the complexity of system components can increase, particularly as they adapt to new requirements and technologies. Despite the low frequency, license exceptions pose significant challenges. For example, in systems developed with C/C++ or similar languages, the compiled binaries, linked from various source codes, inherit the licensing constraints of these sources. Tools that check compatibility may likely flag or report conflicts, such as the coexistence of LGPL-2.1-only and Apache-2.0 licenses, unless the tool overlooks the file license. Developers then manually verify each of these conflicts, which is labor-intensive.

Exploring **RQ1** and **RQ2**, we discovered that compatibility assessments typically neglect license exceptions, causing inconsistent findings across identical license combinations and reducing their applicability in practical scenarios. Further research indicates that despite the relative rarity of license exceptions, the occurrence of license exceptions introduces substantial governance difficulties, significantly hindering efforts to enforce license compliance.

3) RQ3: License exceptions typically arise from specific changes in the rights and obligations granted by a license under certain conditions and are crucial in the final step of license compatibility analysis. Identifying these exceptions and their trigger conditions enables the development of tools that effectively meet real-world needs. Among the 66 license exceptions listed by SPDX, we collaborated with legal experts to analyze their clauses. Our goal was to ascertain the feasibility of automatically detecting the conditions for these license exceptions. The criterion for our analysis was whether the trigger condition in license exception terms could be extracted solely from the work. Our findings indicate that 64 out of the 66 exceptions have identifiable conditions despite some being vaguely defined. Interestingly, understanding these vague conditions requires little legal knowledge.

TABLE II
COMPREHENSIVE COMPILATION AND CATEGORIZATION OF LICENSE EXCEPTIONS

Exception Type	Licenses SPDX ID	Effective Conditions	Base License
Exception Relicense	*CLISP-exception-2.0*	Packages not part of CLISP	GPL-2.0
	GCC-exception-2.0, GCC-exception-2.0-note, GNOME-examples-exception, eCos-exception-2.0, GCC-exception-3.1	Specified file	GPL-2.0
	Autoconf-exception-macro, Autoconf-exception-2.0, Autoconf-exception-3.0, Bison-exception-1.24	Program Output	GPL-2.0
	Font-exception-2.0	Font	GPL-2.0
	FLTK-exception	Subclassed from FLTK widgets	LGPL-2.0
	389-exception, GPL-3.0-interface-exception, freertos-exception-2.0, u-boot-exception-2.0	Approved Interfaces	Various
	PS-or-PDF-font-exception-20170817	Postscript	AGPL-3.0
	Bison-exception-2.2	Combined work	GPL-2.0
	Bootloader-exception, LZMA-exception, mif-exception, GPL-3.0-linking-source-exception, Classpath-exception-2.0, GNU-compiler-exception, gnu-javamail-exception, GNAT-exception	Dynamic/Static linking	GPL-2.0
	Swift-exception	Compile/embed	Apache-2.0
	freertos-exception-2.0	FreeRTOS communication	GPL-2.0
Exception Compatible	*cryptsetup-OpenSSL-exception, x11vnc-openssl-exception*	OpenSSL	GPL-2.0
	GStreamer-exception-2008, GStreamer-exception-2005	GStreamer plugins	GPL-2.0
	Gmsh-exception	Specified combined code	GPL-2.0

In our analysis, we classified license exceptions into four categories, acknowledging that these categories are not strictly separate; exceptions may serve multiple functions simultaneously:

1. **Exception Relicense**: This category involves changes to the rights, obligations, or restrictions under specific conditions. These changes may affect one or several clauses of the original license.
2. **Exception Remediating**: These exceptions address remedial actions following significant license violations, aiming to prevent disputes from escalating into legal actions.
3. **Exception Compatible**: The licensed work can be compatible with a particular type of work, a specific license, or a named project in this type of exception.
4. **Exception Refine**: Focuses on clarifying the original license to mitigate compliance risks due to ambiguous terms.

In addition, our analysis extends beyond classification to a detailed examination of 35 excluded clauses and their respective exception conditions, presented in Table II. **Base License** refers to the original license affected by these exceptions. The **Effective Conditions** column shows the trigger conditions of the exception. Critical conditions include **Specified File**, activating the Exception for specific file interactions; *Program Output*, for exceptions triggered by program output; *Approved Interfaces*, for interaction via specified interfaces; and *Dynamic and Static Link*, for exceptions applied during dynamic or static linking.

In answering **RQ3**, we found that key information for identifying compatibility issues, such as clauses affecting compatibility under certain conditions, is mainly found in source code, configuration files, or the status of objects during compatibility checks. Coupled with our **RQ2** findings, this underscores a significant problem: License exceptions and license authorization work in different usage methods will directly affect the compatibility of the license. Unfortunately, previous research did not fully consider these problems.

IV. RESEARCH METHOD

A. Notations

This paper presents a formal framework to describe the structure and implications of software licenses, defining key notations as show in Table III.

B. Problem Statements

According to **RQ2** and **RQ3**, licenses exhibit varying compatibility based on different usage scenarios. The customization and exceptions within licenses make using a license as the smallest unit for compatibility analysis impractical. Thus, license compatibility tools must be capable of automatically inferring compatibility with other licenses under various usage scenarios by analyzing a collection of license terms characteristics. A knowledge base that includes scenario-specific compatibility findings is crucial for developing tools that adapt

TABLE III
SUMMARY OF NOTATIONS

Symbol	Description
t_k	A single license term.
l_a	A license represented by a set of terms $\{t_1^a, t_2^a, \ldots, t_n^a\}$.
T_{can}	Category of authorization terms.
T_{not}	Category of prohibition terms.
T_{must}	Category of obligation terms.
$T_{special}$	Category of special terms introduced in this study.
$t_{k\in}T_{can}$	Expression indicating t_k is an authorization term.
(l_a,l_b)	Direct, unconditional compatibility between l_a and l_b.
$\langle l_a, l_b, S_a\rangle$	Conditional compatibility based on satisfying S_a.
$G_p = \langle V, E\rangle$	G_p represents the project, nodes V may include source code and dependency libraries, and edges E represent the interoperability of nodes.

to various usage scenarios. Such detailed analysis not only aids in current compatibility assessments but also prepares the framework to accommodate new licenses and exceptions.

To address the above issues, we propose a new definition of compatibility that more accurately reflects various scenarios. We have refined our understanding of license compatibility, conflict, and condition compatibility as follows:

Definition 1: License l_a is unconditionally compatible with l_b if no conflict exists between their terms.

Remark 1: Traditional compatibility definitions confuse the conflict between license terms and compatibility in specific usage scenarios, meaning one-way compatible licenses often have conflicts in terms. At the same time, they may be compatible with specific usage scenarios. Considering actual usage or distribution, any one-way compatible licenses can lead to compliance problems. It is easy to prove that the unconditional compatibility relationship defined in this article has no direction, which means that the license for unconditional compatibility does not have any legal conflicts in any scenario.

Definition 2: License l_a is conditionally compatible with l_b if l_a conflicts with l_b but the conflict can be avoided under specific conditions.

Remark 2: According to conditional compatibility, all conflicting licenses must have a defined, non-empty resolution scope to avoid actual conflict, rendering them conditionally compatible. If multiple clauses conflict, conditional compatibility is achievable only if there is a non-empty intersection among all resolution scopes. In such cases, the derived conditions represent the overlapping scope. Ideally, the differences in license terms for various usage scenarios should be indicated on the directed edges of conditional compatibility, ensuring that conclusions about license term compatibility are applicable in practical scenarios.

Definition 3: If License l_a is incompatible with License l_b, then conflicts between l_a and l_b cannot be avoided for work authorized under l_a.

Remark 3: Under this definition, a set of licenses is incompatible if conflicts arise in all interactions among them. Conversely, any condition allowing for compatibility suggests potential compatibility within the set.

Finally, to analyze license compatibility in complex scenarios, the process can be defined as follows:

- **Input**: A code repository P, including its file directory structure and dependency graph $G_p = \langle V, E\rangle$.
- **Output**: A conflict graph G_I illustrates the relationships between incompatible licenses in the repository.

C. Approach Overview

The empirical studies in section III highlight three main challenges in license compatibility:

- Different licenses demonstrate diverse compatibility characteristics depending on the programming language and dependency behavior.
- License exceptions commonly not considered by conventional compatibility checking tools modify the fundamental nature of license compatibility.
- Existing structured approaches fail to incorporate specific usage scenarios and exception clauses, resulting in flawed compatibility judgments.

To bridge these gaps, this study proposes **LiScopeLens**, a novel license compatibility checking method that includes:

1. A model for representing license terms scopes (SRLT);
2. SRLT-based inference rules and a compatibility knowledge graph tailored to usage scenarios;
3. A scenario sniffer module and a project-specific license compatibility checker operate under defined compatibility conditions.

The LiScopeLens workflow, depicted in Fig 2, starts by gathering structured **SRLT** data to analyze the scope of terms for licenses l_a and l_b, determining if conflicts can be preventively avoided. The compatibility reasoning module uses this data to infer compatibility, illustrated through the relationships among license nodes, where the relationships store the conditions for compatibility. The scenarios sniffer checks for compliance with the predefined conditions in the code repository. The final step involves the license compatibility checker, which combines information from the behavior sensor and the knowledge graph to confirm the presence or absence of conflicts.

1) Scope Representation of License Terms: We introduce a new feature representation method named Scope Representation of License Terms (**SRLT**), which builds on structured information from licenses to ensure accurate compatibility of conditions at the term level. During the compatibility analysis,

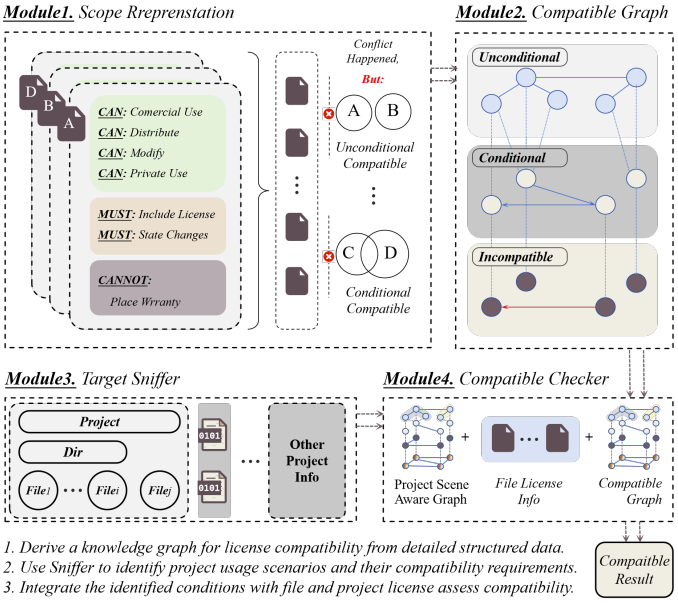

Fig. 2. **LiScopeLens**: compatibility detection method logical overview.

SRLT advances previous structural approaches by precisely defining the scope of license conflicts and examining potential conflicts beyond this scope.

Determining the **effective scope** is challenging due to its dependence on specific definitions and statements within the license terms. In practice, many license terms are unclear and do not encompass all potential scenarios, often requiring subjective explanation by experts. This paper introduces the SRLT method, simplifying the **effective scope** into **application scope** and **out of application scope**. Specifically, the scope of a license term can be formally defined using the formula $S(t) = A(t) \cap \overline{O(t)}$, where $A(t)$ includes scenarios covered by the license and $O(t)$ includes excluded scenarios. The terms are effective when the authorized work falls within $A(t)$ but not $O(t)$. Conflicts between $A(t)$ and $O(t)$, indicating overlapping definitions, are beyond this discussion's scope.

The distinction between $A(t)$ and $O(t)$ effectively resolves ambiguities in many license terms. For instance, consider a license term t_k within license l_a, where the scope is unclear and the $O(t)$ is not specified. In such cases, $A(t_k^a) = \mathbb{U}$ and $O(l) = \emptyset$, indicating that all conditions are considered within the scope. This setting aligns the **effective scope** of SRLT with conventional structuring schemes, maintaining consistency even when expert validation is absent, ensuring that outcomes do not deviate from established methods. Conversely, for a term t_k in license l_b that explicitly excludes 'dynamic linking,' we define $A(t_k^b) = \mathbb{U}$ and $O(t_k^b) = \{$'dynamic linking'$\}$. To determine the overlap in scope between licenses l_a and l_b for t_k, one should calculate $S(t_k^a) \cap S(t_k^b)$, simplifying to $S(t_k^b)$. If t_k^a is permitted under l_a but not under l_b, it signifies a conflict but also indicates a potential for conditional compatibility, depicted as $\langle l_b, l_a, \overline{t_k^b} \rangle$.

The functions $A(t)$ and $O(t)$ clarify the **effective scope** of license terms and enhance the granularity of term expressions, facilitating compatibility assessments based on standard rules. Compatibility is determined by whether the **effective scopes** overlap. The calculations for scope compatibility, as shown in Equation 1, are performed separately for the sets $A(t)$ and $O(t)$. For intersecting scopes, the computation is:

$$\begin{aligned} S(t_k^a) \cap S(t_k^b) &= (A(t_k^a) \cap \overline{O(t_k^a)}) \cap (A(t_k^b) \cap \overline{O(t_k^b)}) \\ &= (A(t_k^a) \cap A(t_k^b)) \cap \overline{O(t_k^a) \cup O(t_k^b)} \end{aligned} \tag{1}$$

We can convert the outcome of Equation 1 into the standard $S(t)$ format, but this method fails with set unions or subtractions. A binary relationship format resolves these problems by representing the $S(t)$ as a set of binary ordered pairs:

$$\begin{aligned} S(t_k) &= A(t_k) \times O(t_k) \\ &= \{< x, y > |\ x \in A(t_k) \wedge y \notin O(t_k)\} \end{aligned} \tag{2}$$

For any clause t_k, the intersection and negation operations are defined as follows in its effective range $S(t_k)$:

$$\begin{aligned} \overline{S(t_k)} &= \{< x, y > |\ x \notin A(t_k) \vee y \in O(t_k)\} \\ &= U \times A(t_k) + O(t_k)) \times \emptyset \end{aligned} \tag{3}$$

$$\begin{aligned} S(t_k^a) \cap S(t_{k)}^b = \{< x, y > |\ x \in \\ (A(t_k^a) \cap A(t_k^b)) \wedge y \notin (O(t_k^a) \cup O(t_k^b))\} \end{aligned} \tag{4}$$

In addition, $S(t_k)$ also has the following properties to ensure the simplicity of the final calculation result:

Anti-reflexivity: For $\forall < x, y >$ if $x = y$ then $< x, y > \notin S(t_k)$. According to the definition, the area where $A(t)$ and $S(t)$ intersect is meaningless. There is also an extend $\forall x \forall y \neg(\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R)$, which means that the intersection of two opposite valid ranges creates a contradiction and is meaningless.

2) Compatibility reasoning based on SRLT: SRLT provides essential information support for defining the effective scope of terms necessary to develop compatibility tools with scenario judgment capabilities. Subsequently, we introduce automated compatibility reasoning rules based on SRLT and new compatibility definitions. SRLT not only vertically refines the existing structured clause framework to enhance the analysis of complex license terms but also horizontally expands this framework. This expansion is evident in two main areas:

1. Beyond the basic CAN, CANNOT, and MUST categories, we introduce special clause features to address various exceptions.
2. We are the first to consider how individual clauses manifest different compatibility phenomena depending on their categories during the compatibility analysis.

This study introduces the fourth category T_{special}, which primarily handles exceptions and exhibits two distinct characteristics:

Triggering: Certain terms or restrictions, which become effective under specific conditions, can impose corresponding

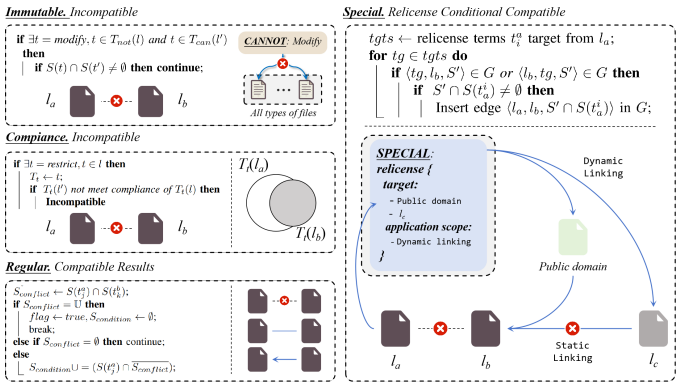

Fig. 3. **LiScopeLens**: Compatibility reasoning based on SRLT overview.

constraints once triggered. These clauses enhance compatibility in licenses with restrictive requirements. For instance, unlike in earlier versions, the GPL-3.0-only license has become compatible with Apache-2.0 due to such clauses.

Relicense: Triggering specific terms can effectively reauthorize the original license with new characteristics, e.g., classpath-exception-2.0, which, under specific conditions, gives up all rights and obligations of the original license. It is important to note that 'relicense' implies that the work, when used under the defined conditions, should be considered for compatibility under a new license. However, it does not alter the original license's terms.

We further categorize the characteristics of **SRLT** terms itself as follows:

Compliance Requirement: When a clause t_k in l_a requires that l_b imposes specific rights, obligations, or restrictions on **a designated object** that must be a subset of another corresponding set of clauses in l_a, we say that t_k has a compliance requirement. The "cannot impose further restriction" feature is an instance of this compliance requirement that prior studies have not considered.

Immutability: This attribute applies to licenses that restrict any alteration of the work to all extents. The "Cannot" clause collection flags the "modify" action when it is present.

Extending from the above structural features, the pseudocode for the inference rule is shown in Algorithm 1.

In our approach, we assess the compatibility of licenses, each characterized by a set of **SRLT** features, through pairwise comparison within the license set. The compatibility analysis is structured into four sequential stages as depicted in Fig 3. Initially, we identify any immutable licenses deemed incompatible with all others. If there is no immutability, we proceed to the compliance requirement check. This involves verifying if a license l_a mandates inclusion of its action set of type T by another license l_b, checking $S(T_a) \supseteq S(T_b)$. Failure to meet this criterion results in a declaration of incompatibility; otherwise, the analysis advances to the routine inspection. The core of this inspection is to evaluate potential conflicts between the rights and obligations specified by one license and the restrictions imposed by another. Unresolvable conflicts lead to

Algorithm 1: Reasoning license compatibility based on SRLT

Input: License set $L = \{l_1, l_2, \ldots, l_n\}$
Output: Compatibility graph G

Procedure `InferCompatiblityGraph(`L`):`
  Initialize an empty graph G;
  Initialize an empty array $Exceptions$;
  for $l_a, l_b \in Product(L, L)$ **do**
    `// Product is the Cartesian product of a set`
    $S_{condition} \leftarrow \emptyset, flag \leftarrow false$;
    if $l_a = l_b$ **then continue**;
    if $\exists t = relicense, t \in T_{special}(l_a)$ **then**
      Insert (l_a, l_b) into $Exceptions$;
    `// Special terms are handled last as callbacks.`
    if $\exists t = modify, t \in T_{not}(l)$ *and* $t \in T_{can}(l')$ **then**
      if $S(t) \cap S(t') \neq \emptyset$ **then continue**;
    `// Using` l `and` l' `indicate the process is bidirectional.`
    if $\exists t = restrict, t \in l$ **then**
      $T_t \leftarrow t$;
      if $T_t(l')$ *not meet compliance of* $T_t(l)$ **then**
        Incompatible
    `// Using` l `and` l' `indicate the process is bidirectional.`
    for $(t_j^a, t_k^b) \in$ *All conflicting terms pairs* **do**
      $S_{conflict} \leftarrow S(t_j^a) \cap S(t_k^b)$;
      if $S_{conflict} = \mathbb{U}$ **then**
        $flag \leftarrow true, S_{condition} \leftarrow \emptyset$;
        break;
      else if $S_{conflict} = \emptyset$ **then** continue;
      else
        $S_{condition} \cup = (S(t_j^a) \cap \overline{S_{conflict}})$;
    if $flag = false$ **then**
      Insert edge (l_a, l_b) in G;
      `// Unconditional compatible.`
    else if $S_{condition} \neq \emptyset$ *and* $flag = true$ **then**
      Insert edge $\langle l_a, l_b, S_{condition} \rangle$ in G;
      `// Conditional compatible.`
  for $l_a, l_b \in Exceptions$ **do**
    $tgts \leftarrow$ relicense terms t_i^a target from l_a;
    for $tg \in tgts$ **do**
      if $\langle tg, l_b, S' \rangle \in G$ *or* $\langle l_b, tg, S' \rangle \in G$ **then**
        if $S' \cap S(t_a^i) \neq \emptyset$ **then**
          Insert edge $\langle l_a, l_b, S' \cap S(t_a^i) \rangle$ in G;
  `// Handling terms in licenses that enforce the utilization of distinct legal features.`
  return G;

a finding of incompatibility.

In addition, special license terms are managed through a callback-like approach. When a usage scenario triggers special terms, the tool considers the license authorized under a new one, thus changing its compatible status. Our inference algorithms delay compatibility determinations for these instances until all standard cases have been reviewed. If a license l_a featuring a relicense clause is either incompatible with l_b or meets specific conditions for compatibility, we assess their compatibility based on the relationship between the relicense target of l_a and l_b. The critical factor is the intersection (S') between the relicense's applicable range and l_b's scope. An intersection that is not empty indicates compatibility, serving as the criterion for this assessment.

3) Target Sniffer and Compatibility checker: After creating the license compatibility graph, we identify all potential compatibility conditions based on the licenses' terms. We use parsers and sniffers to analyze usage scenarios, aligning essential license conditions with their compatible counterparts. This alignment helps verify if a license combination meets the required compatibility criteria. For instance, in low-level programming languages that involve binary distributions, we deploy sniffers to understand usage patterns. The initial set of sniffers identifies target files and their licenses in the code repository. A subsequent set analyzes compiled configurations to map source code to binary files. Additional sniffers may be employed to examine dependencies and linkage behaviors. This gathered data feeds into a compatibility check module, which then assesses the compatibility of licenses in the given scenario.

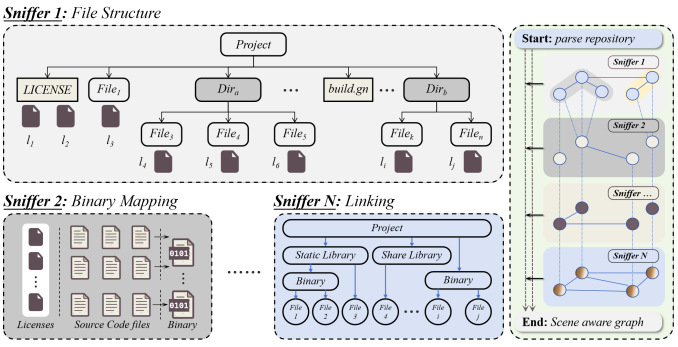

Fig. 4. **LiScopeLens**: Sniffer aware usage scenario workflow diagram.

We choose C/C++ as an example to explain how **LiScopeLens** works. Since C/C++ does not have practical package-level management tools and may have binaries with multiple licenses, it needs effective license analysis tools. The approach is detailed in the pseudocode 2.

Algorithm 2: Check license compatibility

Input: Project dependency graph G_p, Compatibility knowledge graph G_c
Output: Processed graph G_I

Procedure `CheckCompatibility`(G_p, G_c):
  $L_{sub} \leftarrow \emptyset$;
  $G_I \leftarrow$ an empty graph;
  $G_{dep} \leftarrow$ Extract dependency subgraph from G_p;
  for v_i **in** *reverse topological sort* G_{dep} **do**
    $P_i \leftarrow$ get parents from v_i;
    $C_i \leftarrow$ get children from v_i;
    for $[v_a, v_b] \in Combination(\{v_i\} \cup C_i)$ **do**
      $cond_a \leftarrow$ Obtain (v_i, v_a) from G_{dep};
      $cond_b \leftarrow$ Obtain (v_i, v_b) from G_{dep};
      $L_a \leftarrow$ licenses from v_a;
      $L_b \leftarrow$ licenses from v_b;
      for $[l_a, l_b]$ **in** *Product*(L_a, L_b) **do**
        $c_a \wedge =$ Query compatibility for $(l_a, l_b, cond_a)$;
        $c_b \wedge =$ Query compatibility for $(l_b, l_a, cond_b)$;
      Insert edge $\langle l_b, l_a, c_a \vee c_b \rangle$ in G_I;
    $L_i \leftarrow$ licenses from v_i;
    for v_j **in** C_i **do**
      $L_j \leftarrow$ the licenses spread to v_i;
      $L_i \leftarrow L_i \vee L_j$;
    `// Update` L_i, `e.g., copyleft license or usage scenario spread to parent nodes`
  return G_I;

D. Evaluation

1) Compatibility reasoning based on SRLT: The core of this research is the automation of reasoning processes using the novel license's structured data **SRLT**, aimed at refining the license compatibility knowledge graph for complex usage contexts. This method's validity was evaluated through a manual examination of 17 licenses employing the **SRLT** framework, followed by the application of rule-based reasoning (outlined in Algorithm 1), with the findings depicted in Fig 5. The horizontal axis denotes the initial license, while the vertical axis represents the final license. This setup aims to assess compatibility between the originating and concluding licenses. Specifically, red points (value of 2) signal incompatibility, gray points (value of 1) suggest conditional compatibility and yellow points (value of 0) denote full compatibility. As discussed in Section III, license compatibility does not have a one-size-fits-all answer due to varying underlying assumptions for different works. **SRLT** aims to mitigate such issues. The OSADL license compatibility matrix shows that LGPL-2.1-only and Apache-2.0 are mutually incompatible. However, our study finds no conflict under conditions of "dynamic linking" between LGPL-2.1-only licensed works and Apache-2.0, illustrating conditional compatible, **as seen in the real-world case of a more permissive link exception** [35].

Additionally, this study incorporates findings from tldrlegal [36], extending our license analysis to 113 licenses. We applied the **SRLT** modeling and inference techniques to create

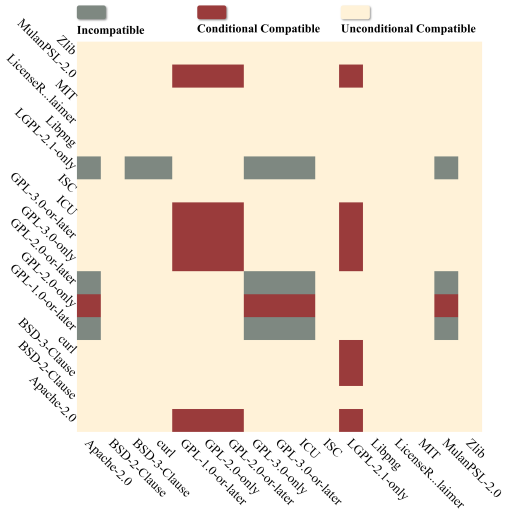

Fig. 5. Heat map of compatibility inference for licenses after manual processing.

TABLE IV
COMPARISON OF COMPATIBILITY BETWEEN EXISTING WORK

	OSADL	Openeuler
Compatible-Incompatible	661	538
Incompatible-Compatible	28	16
Condition-Compatible	9	1
Condition-Incompatible	15	14
Total Compared	2105	1126
Inconsistencies	713	571

a larger-scale compatible knowledge graph, comparing our results with those from OSADL and Openeuler in Table IV. Our analysis reveals significant differences from traditional conclusions, especially regarding license compatibility. Unlike the OSADL, which finds many license combinations incompatible, our method deems them unconditionally compatible.

Upon examining these differences, we identified two primary sources of error. First, despite tldrlegal's structured information being generally accurate, it sometimes deviates from our base data. Second, the discrepancies largely stem from how our method and traditional approaches define compatibility. Our method only analyzes license conflicts based on license terms, removing the confusion between usage scenarios and term conflicts. It models the effective scope of different terms through the **SRLT** to obtain a compatibility knowledge base with usage scenario requirements in automated inference.

TABLE V
RESULTS FOR INCOMPATIBILITY DETECTION.

Method	Projects	Incompatibilities	FP	Time Cost
LiDetector	215	15	15(100.0%)	93.42(s)
LiScopeLens	-	10	2(20.0%)	36.49(s)

2) License compatibility testing performance: In this evaluation, we sought to validate that **LiScopeLens** is a potentially powerful tool for managing real-world cases' intricate compatibility testing needs. We chose the OpenHarmony 5.0 beta, an emerging open-source operating system project, to evaluate the detection capabilities of textbfLiScopeLens. We then compared our tools with the impressive work of LiDetector [20]. The results are presented in Table V. The difference in the number of projects is that LiDetector can only perform license compatibility checks on a project-by-project basis, so we had to split the OpenHarmony 5.0 beta complete code into 215 project modules. The tool proposed in this article processes licenses at the file level, so there is no need for such a division. LiDetector identified 15 license incompatibility issues, but upon our manual verification, we discovered that all 15 were false positives. The tool we proposed in this paper reported 10 types of conflicts, and through manual simulation, we found that 2 were false positives. Furthermore, we analyzed the time costs of the two tools during the compatibility check process and established that LiScope Lens operates 62.87% faster than its predecessor. It is worth noting that this practical cost does not include the process of license scanning, and the time cost results of LiScopeLens also account for compatibility inference.

E. Case Study

1) License Compatibility Prior Knowledge: We analyzed many inconsistent results to identify the sources of these differences.

- **Structural Information Errors**: Ambiguities in tldrlegal's structural data contribute to uncertain compatibility inferences. For instance, the Unlicense is listed with contradictory permissions regarding liability, attributed to the automated parsing of terms like "released into the public domain" [37]. Such discrepancies highlight the challenges of machine interpretation of license terms.
- **Compatibility Definition Inconsistencies**: Our compatibility analysis focuses on potential conflicts in license combinations, diverging from methods that consider integrating works under different licenses. For example, although CDDL-1.0 and curl licenses do not directly conflict, the former's greater restrictions dictate the licensing terms for combined works. This distinction underscores our belief that issues of outbound licensing should not be conflated with compatibility concerns, a topic we plan to explore further.
- **Concluding Errors in Other Analyses**: Misinterpretations in other studies, such as the deemed incompatibility between MPL-2.0 and GPL-2.0 by OSADL, underscore the importance of our work. Despite claims of incompatibility, evidence shows MPL-2.0 licenses can be fully incompatible [38], highlighting the need for reassessment in license compatibility analysis.

2) Compatibility Detection: We have analyzed a large number of cases in license compatibility detection and summarized some typical problems:

- **Complex usage scenarios**: The complexity of code usage scenarios is a significant challenge: LiDetector strives to categorize licenses into project-level and file-level types, yet it overlooks extensive projects with numerous sub-projects. Furthermore, due to their construction or distribution techniques, these large-scale projects have induced interactions between previously isolated projects or components. These disregarded aspects significantly contribute to the underreporting by license compatibility detection tools.
- **Complex compatibility of terms**: As for license compatibility, prior knowledge, its accuracy, and the granularity of its modeling are crucial determinants of a detection tool's capability. For instance, in the case of LiDetector, all 15 false positives resulted from unforeseen errors in the compatibility judgment rules for project-level and file-level licenses.

V. Related Work

A. License Identification

license compliance analysis often begins with the scanning and identification of licenses. The presence of embedded and referenced licenses complicates accurate retrieval. Previous research has significantly contributed to this area, generally falling into two categories: character-based and semantic-based methods. Character-based methods, such as the approach by Jaeger et al. in Fossology, utilize short seed regular expressions and citation databases to identify licenses [19]. Tools like Scancode [34] and Ninka [17] are notable examples. On the other hand, semantic-based methods, exemplified by Wang et al. [14], employ the LDA model and doc2vec for mining potential topics and analyzing term similarity, respectively, to identify license terms. Xu et al.'s work [20] advances this approach by using natural language processing and training models for finer recognition granularity. While semantic-based methods show promise in detecting unknown licenses, their effectiveness is enhanced when combined with character-based methods, accommodating the complex real-world.

B. License Incompatibility Detection

The rapid growth of the open-source community has increasingly led to the development of proprietary software based on open-source foundations, highlighting the importance of open-source compliance. Beyond the methods discussed previously [20], [28], research has expanded into various aspects of license incompatibility.

Paschalides et al. [39] utilized a license compatibility graph and graph algorithms to identify potential incompatibility, but neglected the subtleties of license condition compatibility. Hemel et al. [40] introduced a binary code clone detection technique to uncover software license volations. Makari et al. [41] analyzed the prevalence and evolution of license violations within npm and RubyGems dependency networks, revealing that deep dependencies are less likely to cause license incompatibilities compared to shallow ones, and that GPL license issues are the primary source of incompatibility. Higashi et al. [42] focused on license incompatibilities in Docker images, employing Tern and Ninka tools to assess software packages and licenses, finding that 58.9% of Docker images exhibited license compatibility issues.

While these studies contribute significantly to our understanding of license compatibility, they largely omit the impact of license exceptions and a systematic analysis of clause compatibility, which are crucial for a comprehensive understanding of license compliance in the open-source ecosystem.

VI. Conclusion

This paper explores the complexities of open-source licenses within the context of sustainable open-source software and community development. We address challenges posed by the open-source software supply chain's intricate usage scenarios and license exceptions. To this end, we present a novel compatibility analysis tool to discern various usage scenarios and manage license exceptions and tailored clauses. This advancement is grounded in comprehensive empirical studies on license exceptions, which we have organized and analyzed alongside real-world examples. By introducing a Structured Representation of License Terms (SRLT), we clarify ambiguous license scopes and employ automated reasoning to construct a scenario-based compatibility knowledge graph, ultimately deriving precise compatibility conclusions.

Our research innovates in the area of license representation, specifically tackling exceptions. The introduced LiScopeLens tool is designed to overcome the limitations of existing compatibility checkers, which need help to provide accurate results across different usage contexts. We aim to develop a comprehensive license-checking tool that can handle exceptions and unknown compatibilities. Our plans include:

- Developing a hybrid character and semantic-based detection algorithm to pinpoint specific clauses in license texts and interpret their effective scopes using the SRLT method.
- Incorporating additional compliance checks, such as outbound licensing considerations, and validating the tool's accuracy using a wider range of real-world data samples.

Acknowledgement

This work is supported by HY-Project under No.4E49EFF3, the Gansu Province Key Research and Development Plan - Industrial Project under Grant No. 22YF7GA004, the Open Research Fund of The State Key Laboratory of Blockchain and Data Security, Zhejiang University, and Supercomputing Center of Lanzhou University.

References

[1] G. S. Kyle Daigle, "Octoverse: The state of open source and rise of ai in 2023," https://github.blog/2023-11-08-the-state-of-open-source-and-ai/, 2024, accessed: 2024-03-25.

[2] A. C. Roman, K. Xu, A. Smith, J. T. Vega, C. Robinson, and J. M. L. Ferres, "Open data on github: Unlocking the potential of ai," 2023.

[3] X. Liu, Y. Zhang, Q. Yu, J. Min, J. Shen, R. Zhou, and Q. Zhou, "Smarteagleeye: A cloud-oriented webshell detection system based on dynamic gray-box and deep learning," *Tsinghua Science and Technology*, vol. 29, no. 3, pp. 766–783, 2024.

[4] C. Vendome, M. Linares-Vasquez, G. Bavota, M. Di Penta, D. M. German, and D. Poshyvanyk, "When and why developers adopt and change software licenses," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep 2015. [Online]. Available: http://dx.doi.org/10.1109/icsm.2015.7332449

[5] "Gpl legal battle: Vizio told it will face contract claims," https://www.theregister.com/2022/05/16/vizio_gpl_contract/, 2022, accessed: 2024-03-25.

[6] "China's courts pass controversial rulings on open-source licencing," https://www.lexology.com/library/detail.aspx?g=597bfc93-0e53-4ffb-8311-a8fe3129d7f8, 2024, accessed: 2024-03-25.

[7] E. Zio, "Reliability engineering: Old problems and new challenges," *Reliability Engineering & System Safety*, vol. 94, no. 2, pp. 125–141, 2009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0951832008001749

[8] K. C. Kapur and M. Pecht, *Reliability engineering*. John Wiley & Sons, 2014, vol. 86.

[9] E. Zio, "Some challenges and opportunities in reliability engineering," *IEEE Transactions on Reliability*, vol. 65, no. 4, pp. 1769–1782, 2016.

[10] M. C. Sánchez, J. M. C. de Gea, J. L. Fernández-Alemán, J. Garcerán, and A. Toval, "Software vulnerabilities overview: A descriptive study," *Tsinghua Science and Technology*, vol. 25, no. 2, pp. 270–280, 2020. [Online]. Available: https://www.sciopen.com/article/10.26599/TST.2019.9010003

[11] J. E. Breneman, C. Sahay, and E. E. Lewis, *Introduction to reliability engineering*. John Wiley & Sons, 2022.

[12] M. Feng, W. Mao, Z. Yuan, Y. Xiao, G. Ban, W. Wang, S. Wang, Q. Tang, J. Xu, H. Su, B. Liu, and W. Huo, "Open-source license violations of binary software at large scale," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Feb 2019. [Online]. Available: http://dx.doi.org/10.1109/saner.2019.8667977

[13] C. Vendome, M. Linares-Vasquez, G. Bavota, M. Di Penta, D. German, and D. Poshyvanyk, "License usage and changes: A large-scale study of java projects on github," in *2015 IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 218–228.

[14] Z. Wang, G. Xiao, Z. Zhang, and S. Wu, "A novel model for automatic identification of open source software license terms," in *2021 IEEE 4th International Conference on Computer and Communication Engineering Technology (CCET)*, Aug 2021. [Online]. Available: http://dx.doi.org/10.1109/ccet52649.2021.9544240

[15] C. Vendome, M. Linares-Vasquez, G. Bavota, M. Di Penta, D. German, and D. Poshyvanyk, "Machine learning-based detection of open source license exceptions," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017. [Online]. Available: http://dx.doi.org/10.1109/icse.2017.19

[16] B. Moreau, P. Serrano-Alvarado, M. Perrin, and E. Desmontils, "Modelling the compatibility of licenses," in *The Semantic Web*, P. Hitzler, M. Fernández, K. Janowicz, A. Zaveri, A. J. Gray, V. Lopez, A. Haller, and K. Hammar, Eds. Cham: Springer International Publishing, 2019, pp. 255–269.

[17] D. M. German, Y. Manabe, and K. Inoue, "A sentence-matching method for automatic license identification of source code files," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*, Sep 2010. [Online]. Available: http://dx.doi.org/10.1145/1858996.1859088

[18] M. Scarlato and L. Nöth, "Lcv-cm: a fasten open source license compliance verifier with compatibility matrix." *2023 17th International Conference on Open Source Systems and Technologies (ICOSST)*, pp. 1–7, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:267338736

[19] G. M. Kapitsaki and D. Paschalides, "Identifying terms in open source software license texts," in *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, Dec 2017. [Online]. Available: http://dx.doi.org/10.1109/apsec.2017.62

[20] S. Xu, Y. Gao, L. Fan, Z. Liu, Y. Liu, and H. Ji, "Lidetector: License incompatibility detection for open source software," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 1, p. 1–28, Jan 2023.

[21] D. A. Wheeler, "The free-libre / open source software (floss) license slide," https://dwheeler.com/essays/floss-license-slide.html, 2017, accessed: 2024-03-26.

[22] "Anything important you should know about licenses," https://compliance.openeuler.org/, accessed: 2024-05-09.

[23] G. M. Kapitsaki, F. Kramer, and N. D. Tselikas, "Automating the license compatibility process in open source software with spdx," *Journal of Systems and Software*, vol. 131, pp. 386–401, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216300905

[24] W. Contributors, "License compatibility," Feb 2024. [Online]. Available: https://en.wikipedia.org/wiki/License_compatibility

[25] D. M. Germán and A. Hassan, "License integration patterns: Addressing license mismatches in component-based development," *2009 IEEE 31st International Conference on Software Engineering*, pp. 188–198, 2009. [Online]. Available: https://api.semanticscholar.org/CorpusID:7948084

[26] C. Vendome, D. M. German, M. Di Penta, G. Bavota, M. Linares-Vásquez, and D. Poshyvanyk, "To distribute or not to distribute?" in *Proceedings of the 40th International Conference on Software Engineering*, May 2018. [Online]. Available: http://dx.doi.org/10.1145/3180155.3180221

[27] G. M. Kapitsaki, A. C. Paphitou, and A. P. Achilleos, "Towards open source software licenses compatibility check," *Proceedings of the 26th Pan-Hellenic Conference on Informatics*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:257805970

[28] W. Xu, X. Wu, R. He, and M. Zhou, "Licenserec: Knowledge based open source license recommendation for oss projects," *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 180–183, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:259105134

[29] D. M. German, M. Di Penta, and J. Davies, "Understanding and auditing the licensing of open source software distributions," in *2010 IEEE 18th International Conference on Program Comprehension*, Jun 2010. [Online]. Available: http://dx.doi.org/10.1109/icpc.2010.48

[30] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017. [Online]. Available: https://api.semanticscholar.org/CorpusID:7402387

[31] S. van der Burg, E. Dolstra, S. McIntosh, J. Davies, D. M. German, and A. Hemel, "Tracing software build processes to uncover license compliance inconsistencies," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, Sep 2014. [Online]. Available: http://dx.doi.org/10.1145/2642937.2643013

[32] "Osadl compatibility matrix of foss licenses ("client-only" version)," https://www.osadl.org/html/CompatMatrix-noexpl.html, accessed: 2024-05-09.

[33] S. Zacchiroli, "A large-scale dataset of (open source) license text variants," *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*, pp. 757–761, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:247922555

[34] "scancode-toolkit," https://github.com/nexB/scancode-toolkit, accessed: 2024-05-09.

[35] "Licensing: Gpl2-only is apache 2.0 incompatible," https://github.com/libgit2/libgit2/issues/567, accessed: 2024-05-09.

[36] "Software licenses in plain english," https://www.tldrlegal.com/, accessed: 2024-05-09.

[37] "tldrlegal: Unlicense," https://www.tldrlegal.com/, 2024, accessed: 2024-05-09.

[38] "Mpl 2.0 faq," https://www.mozilla.org/en-US/MPL/2.0/FAQ/, 2024, accessed: 2024-05-09.

[39] D. Paschalides and G. M. Kapitsaki, "Validate your spdx files for open source license violations," *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:17156519

[40] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, "Finding software license violations through binary code clone detection," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 63–72.

[41] I. S. Makari, A. Zerouali, and C. D. Roover, "Prevalence and evolution of license violations in npm and rubygems dependency networks," in *International Conference on Software Reuse*, 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:249872864

[42] Y. Higashi, K. Fukui, K. Yutaro, and M. Ohira, "A preliminary analysis of gpl-related license violations in docker images," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 444–448.