An Empirical Study of Suppressed Static Analysis Warnings

HUIMIN HU, University of Stuttgart, Germany
YINGYING WANG, The University of British Columbia, Canada
JULIA RUBIN, The University of British Columbia, Canada
MICHAEL PRADEL, University of Stuttgart, Germany

Scalable static analyzers are popular tools for finding incorrect, inefficient, insecure, and hard-to-maintain code early during the development process. Because not all warnings reported by a static analyzer are immediately useful to developers, many static analyzers provide a way to suppress warnings, e.g., in the form of special comments added into the code. Such *suppressions* are an important mechanism at the interface between static analyzers and software developers, but little is currently known about them. This paper presents the first in-depth empirical study of suppressions of static analysis warnings, addressing questions about the prevalence of suppressions, their evolution over time, the relationship between suppressions and warnings, and the reasons for using suppressions. We answer these questions by studying projects written in three popular languages and suppressions for warnings by four popular static analyzers. Our findings show that (i) suppressions are relatively common, e.g., with a total of 7,357 suppressions in 46 Python projects, (ii) the number of suppressions in a project tends to continuously increase over time, (iii) surprisingly, 50.8% of all suppressions do not affect any warning and hence are practically useless, (iv) some suppressions, including useless ones, may unintentionally hide future warnings, and (v) common reasons for introducing suppressions include false positives, suboptimal configurations of the static analyzer, and misleading warning messages. These results have actionable implications, e.g., that developers should be made aware of useless suppressions and the potential risk of unintentional suppressing, that static analyzers should provide better warning messages, and that static analyzers should separately categorize warnings from third-party libraries.

CCS Concepts: • **Software and its engineering** → **Software verification and validation**; **Software post-development issues**.

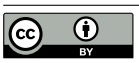
Additional Key Words and Phrases: Static code analysis, false positives, warning management

ACM Reference Format:
Huimin Hu, Yingying Wang, Julia Rubin, and Michael Pradel. 2025. An Empirical Study of Suppressed Static Analysis Warnings. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE014 (July 2025), 22 pages. https://doi.org/10.1145/3715729

1 Introduction

Lightweight static analyzers are popular tools for finding incorrect, inefficient, insecure, and hard-to-maintain code early during the development process. Given a code base, they report warnings about potential bugs, security vulnerabilities, and other violations of common programming rules. Static analyzers are widely used in practice and available for practically all popular programming languages, e.g., Pylint [11] and Mypy [9] for Python, PMD [10], ErrorProne [13], and Infer [18] for Java, and ESLint [5] and Flow [7] for JavaScript. Unfortunately, not all warnings reported by

Authors' Contact Information: Huimin Hu, University of Stuttgart, Stuttgart, Germany, huhuimin236@gmail.com; Yingying Wang, The University of British Columbia, Vancouver, Canada, wyingying@ece.ubc.ca; Julia Rubin, The University of British Columbia, Vancouver, Canada, mjulia@ece.ubc.ca; Michael Pradel, University of Stuttgart, Stuttgart, Germany, michael@binaervarianz.de.



ACM 2994-970X/2025/7-ARTFSE014
https://doi.org/10.1145/3715729

```
267  class stdout(object):
268    # pylint: disable=function-redefined
269    @staticmethod
270    def write(x):
271      # customized code for writing files
```

Fig. 1. Suppression of a warning produced by the static analyzer Pylint in real-world code.

a static analyzer are immediately useful to developers [28, 42], e.g., because a warning is a false positive or the developers do not consider it important enough to address. Since static analyzers are commonly run as part of the build process, developers may easily be overwhelmed by warnings, especially when the same warning appears again and again without being addressed [27].

To avoid overwhelming developers, static analyzers provide a mechanism for suppressing warnings, e.g., by adding annotations or comments to the code. For example, Figure 1 shows some Python code that, when checked with the Pylint analyzer, triggers a warning because the function `stdout` redefines the built-in object with the same name. The warning can be suppressed by adding the highlighted comment. Once a suppression is in place, the static analyzer will not report the corresponding warning anymore, allowing developers to focus on the remaining warnings.

Because suppressions are an important mechanism at the interface between static analyzers and software developers, studying them has the potential to provide insights that are interesting for both software engineering and static analysis. From a software engineering perspective, suppressions clutter the code and may hide actual problems. Understanding to what extent developers accumulate suppressions as the code evolves will provide insights for improving how warnings are reported and managed. From a static analysis perspective, understanding how developers use suppressions will provide insights for improving static analyzers, e.g., by automatically suggesting fixes for easy-to-address warnings or by not reporting commonly suppressed warnings.

This paper presents the first in-depth empirical study on suppressions of static analysis warnings. We study suppressions in code written in three popular languages, Python, Java, and JavaScript, and for warnings produced by four popular static analyzers, Pylint, Checkstyle, PMD, and ESLint. The study addresses five research questions:

- *RQ1: How prevalent are suppressions and what kinds of warnings do developers suppress?* Answering this question helps understand the importance of suppressions and may help analysis creators to improve checks that produce warnings developers tend to suppress. We find that suppressions are relatively common, with a total of 7,357 suppressions in 46 Python projects, which corresponds to a suppression in one out of six code files. Many suppressions address warnings related to conventions, e.g., the "invalid-name" warning by Pylint, which alone accounts for 639 suppressions.
- *RQ2: How do suppressions evolve over time?* After a suppression is added to a code base, it may be addressed in a future commit or remain in the code base forever. Understanding how suppressions typically evolve will help quantify the amount of accumulated suppressions and provide guidance for tools to better handle their evolution. We find the number of suppressions in a project to continuously increase over time. Suppressions typically get either removed within a few months, e.g., by fixing the underlying problem, or they remain in the code for a long time.
- *RQ3: What is the relationship between suppressions and warnings?* A single suppression may hide zero, one, or multiple warnings. Studying how many warnings a suppression is related to, and how this number changes as the code evolves, will help understand whether the suppressions in a code base are still necessary. We find that, surprisingly, 50.8% of all suppressions do not

affect any warning because they have become useless and could simply be removed. The phenomenon of useless suppressions is widespread, affecting all the ten Python projects that we study in detail.
- *RQ4: Do suppressions unintentionally hide future warnings?* When a suppression covers a line, block, or file, it may unintentionally hide warnings introduced *after* the suppression itself. Among 1,537 histories of suppressions, we identify 59 cases of such potentially unintended suppressions of at least 184 warnings. This includes six useless suppressions that later hide warnings.
- *RQ5: Why do developers suppress warnings?* Studying the reasons why developers add and remove suppressions will provide insights about how to improve static analyzers. We find that developers typically add suppressions to address false positives, because the developer misunderstands the warning message, and due to custom coding conventions.

Our findings provide actionable insights for both software engineering and static analysis. We discuss implications for developers, such as that they should be aware of useless suppressions and the potential risks of unintentional suppression; for creators of static analyzers, such as the need for better warning messages, framework-specific customizations, and mechanisms to identify warnings that pertain to third-party code; and for creators of other software engineering techniques, such as the need for automatic repair [35] aimed at suppressions.

Our work complements prior work on studying static analyzers and their warnings. Existing studies investigate how many of all bugs static analyzers find [24, 45], how static analyzers are configured [16, 49], and what limitations developers perceive in them [19, 28]. Most closely related to this paper, previous work has mentioned anecdotally that suppressions are common [15] and that 46% of developers self-report to suppress warnings [19]. To the best of our knowledge, this work is the first to empirically study suppressions of static analysis warnings at the code level.

In summary, this paper makes the following contributions:

- *In-depth study.* The first in-depth study of suppressions of static analysis warnings.
- *Insights.* Insights relevant for software engineering and static analysis.
- *Reusable dataset.* A dataset of 1,873 suppressions and 2,146 suppression histories, which may serve as a basis for future work on techniques for managing suppressions.

2 Methodology

2.1 Terminology

Definition 1 (Warning). *A warning by a static analyzer is a tuple* $w = (f_w, l_w, k_w)$ *where*

- f_w *is the file path of the file containing the warning,*
- l_w *is the line number of the warning, and*
- k_w *is the kind of warning.*

The kind of warning refers to the checks performed by static analyzers, e.g., Pylint reports about 450 different kinds of warnings, such as “bad-super-call”, “arguments-differ”, and “too-many-lines”.

Definition 2 (Suppression). *A suppression is a tuple* $s = (f_s, l_s, k_s)$ *where*

- f_s *is the file path of the file containing the suppression,*
- l_s *is the line number of the suppression, and*
- k_s *is the kind of warning that gets suppressed.*

The kind k_s refers to the same identifiers of different checks performed by a static analyzer as in Definition 1, e.g., the suppression in Figure 1 is represented as the tuple (“`tqdm/cli.py`”, `268`, “`function-redefined`”). The above definitions refer to a specific commit of a code base, and we

discuss the evolution of suppressions across commits in Section 2.3. In addition to suppressions added directly into the source code, some static analyzers support filtering entire categories of warnings via configuration files. Such configuration file-based filters affect the entire code base and are not specific to individual warnings. Because the focus of this work is on understanding how developers suppress specific warnings, we do not consider configuration file-based suppressions.

2.2 Subjects of Study

We study three datasets of projects: a set of 46 open-source Python projects, used in RQ1, a subset of ten of these Python projects, used throughout RQ2–5, and a set of 46 student projects written in Java and JavaScript, used in RQ1 and RQ5.

2.2.1 Open-Source Python Projects. We retrieve projects with a high number of stars, which results in 281 projects with 2,372–136,571 stars per project. We focus on projects that use Pylint, i.e., one of the most popular static checkers for Python. To this end, we search for "pylint" in the projects' file names and file contents, and then manually inspect all matches. We keep a project only if there is evidence that the developers use Pylint, e.g., in the form of a GitHub workflow or a Pylint configuration file, which results in 46 projects with a total of 6,690,087 lines of Python code. We use these 46 projects to study the prevalence of suppressions in RQ1. For the remaining RQs, we identify a random subset of ten of the 46 projects that match all of the following criteria: (i) at least 30 suppressions in the newest commit, to ensure that we have enough suppressions to study; (ii) at least 1,000 commits[1], to ensure that we have enough data to study the evolution of suppressions; (iii) an actual software development project, e.g., an application or library, as opposed to, e.g., a tutorial or collection of interview questions. The resulting set of ten projects comprises 1,071,138 lines of Python code, and in the newest commit of each project, a total of 1,873 suppressions.

Pylint supports suppressing warnings via special comments, as illustrated in Figure 1. We extract suppressions in a given commit of a project through a regular expression that either directly extracts the kind of warning to suppress, e.g., "function-redefined", or a numeric code, which we then map to the kind of warning. The extraction also supports multiple suppressions in a single comment, e.g., `pylint: disable=arguments-differ,too-many-lines`, which we handle as separate suppressions.

2.2.2 Student Projects in Java/JavaScript. We also study 46 Java/JavaScript projects written by undergraduate students taking a third- or forth-year Software Engineering course. All students completed at least two programming courses and other fundamental computer science courses, and about 60% of students have prior internship experience in industry. For the course, students form groups of four to design and develop an Android application written in Java with a cloud-based backend written in JavaScript. Each group proposes their own project idea. The 46 projects have an average of 2,678 lines of Java code and 2,173 lines of JavaScript code.

The students use Codacy [3], an automated code review platform that integrates multiple static analyzers, to identify and subsequently fix issues. Students are instructed to enable Checkstyle [2] for Java, ESLint [5] for JavaScript, and PMD [10] for both Java and JavaScript. The students report the initial commit on which they first run the tool, the initial set of warnings Codacy reported on this commit, the fixes they made, and most importantly for this study, the list of warnings they decide not to fix, together with the reasons for not fixing those warnings. We analyze this data to understand the reasons that lead users to suppress warnings reported by static analysis tools.

2.3 Tracking Suppressions Over Time

At the core of RQ2 is the need to track the evolution of individual suppressions over time:

[1]Project yapf had 967 commits but was included because it had a comparable number of commits and met other criteria.

Algorithm 1: Extract suppression histories.

Input: List *Commits* of commit hashes,
map *Commit2Suppressions* that maps a commit to the suppressions present at the commit
Output: Set *Histories* of suppression histories

$Histories \leftarrow \emptyset$
// **Part 1: Find deleted suppressions**
for c *in* $Commits$ **do**
  $Suppressions_{deleted} \leftarrow extractDeletedSuppressions(c)$
  for s *in* $Suppressions_{deleted}$ **do**
    $h \leftarrow [(\text{“}delete\text{”}, c, s)]$
    $Histories \leftarrow Histories \cup \{h\}$
// **Part 2: Find suppressions that are still present in newest commit**
$c_{newest} \leftarrow newestCommit(Commits)$
$Suppressions_{remaining} \leftarrow Commit2Suppressions(c_{newest})$
for s *in* $Suppressions_{remaining}$ **do**
  $h \leftarrow [(\text{“}remaining\text{”}, c_{newest}, s)]$
  $Histories \leftarrow Histories \cup \{h\}$
// **Part 3: Find beginning of suppression histories**
for h *in* $Histories$ **do**
  $s \leftarrow h[0].s$
  $c_{end} \leftarrow h[0].c$
  $log \leftarrow traverseGitLog(c_{end}, s.f, s.l)$
  $c_{start} \leftarrow oldestEntry(log, Commits)$
  $h \leftarrow [(\text{“}add\text{”}, c_{start}, s)] + h$
return *Histories*

Definition 3 (Suppression history). *A suppression history h consists of two change events. A change event is a tuple (o, c, s), where*

- *o is an “add”, “delete”, “unchanged”, or “remaining” operation,*
- *c is a commit, identified by the commit hash, and*
- *$s = (f_s, l_s, k_s)$ is a suppression.*

Intuitively, a suppression history combines the commits where a suppression gets added to the code base and removed from it. For a suppression present in the newest commit, the history ends with a change event where the operation is “remaining”. If a suppression is removed and later added again, we consider it as two separate suppression histories.

To compute the suppression history of every suppression in a project, we present Algorithm 1, which traverses all commits and extracts events that affect suppressions. The inputs to the algorithm are a list of commits *Commits* and a map *Commit2Suppressions* from commits to the suppressions (obtained as mentioned in Section 2.2.1). The algorithm performs three steps:
1) Find deleted suppressions: For each commit c in *Commits*, the algorithm extracts all suppressions that are present in the parent commit of c but not in c itself. To this end, we use a helper function *extractDeletedSuppressions* to parses the diff. For every suppression that is deleted in c, the algorithm creates a change event with operation “delete” and commit c, and then adds it into a new history.
2) Find suppressions present in the newest commit: To also consider suppressions that never get removed within the given sequence of commits, the algorithm extracts all suppressions that are

```python
class Bag:
  def info(self, price, discount):
    return price * discount

class PaperBag(Bag):
  def info(self, price, discount, coupon): # pylint:
        disable=arguments-differ
    return price * discount - coupon
```

(a) Commit #10.

```python
class Bag: # pylint: disable=too-few-public-methods
  def info(self, price, discount):
    return price * discount
class PaperBag(Bag):
  def info(self, price, discount, coupon): # pylint:
        disable=arguments-differ
    return price * discount - coupon
```

(b) Commit #25.

```python
class Bag: # pylint: disable=too-few-public-methods
  def info(self, price, discount, coupon=None):
    return price * discount
class PaperBag(Bag):
  def info(self, price, discount, coupon):
    return price * discount - coupon
```

(c) Commit #98.

```
("add", 10, ("B.py", 6, "arguments-differ"))
("delete", 98, ("B.py", 11, "arguments-differ"))

("add", 25, ("B.py", 1, "too-few-public-methods"))
("remaining", 150, ("B.py", 1, "too-few-public-methods"))
```

(d) Extracted suppression history.

Fig. 2. Example on how to extract suppression histories.

present in the newest commit c_{newest}. For every such suppression, the algorithm creates a change event with operation "remaining" and commit c_{newest}, and then adds it into a new history.
3) Find beginning of suppression histories: At this point, all histories in set *Histories* only contain information about the end of a suppression's lifetime, but not about the beginning. To fill this gap, the third part of the algorithm identifies the commit c_{start} where each suppression was added. This step is based on a helper function *traverseGitLog*, which uses the `git log` command to obtain the history of a file and a line range in chronological order[2]. The oldest entry in the log corresponds to the commit c_{start} where the suppression was added. The algorithm then creates a change event with operation "add" and commit c_{start}, and adds it to the beginning of the history of the suppression.

Figure 2 shows an example of how Algorithm 1 works. One suppression was introduced in the 10th commit, moved to another line in the 25th commit, and deleted in the 98th commit. Another suppression was introduced in the 25th commit and remains in the code until the most recent commit. Our algorithm will extract two suppression histories (Figure 2d): Commit 25 is not included in the first history, because it moves the suppression but otherwise does not affect it.

Our algorithm relates to work on tracking warnings [48] or code entities [22, 37], e.g., classes and methods, across commits. We address the complementary problem of tracking suppressions, which differs in two ways. First, unlike code entities, suppressions do not have a unique qualified name, but they are simply comments in the code. We address this challenge by tracking the comment line based on the line history provided by `git log`. Second, we need to extract all histories of all suppressions that start or end in the commit history, instead of tracking a specific code element. We address this challenge through the three steps explained above.

2.4 Associating Suppressions and Warnings

RQ3 is about the relationship between suppressions and warnings. To answer this question, we associate suppressions with the warnings they suppress (Section 2.4.1), which then allows us to check whether a warning is still useful in the current commit of the code (Section 2.4.2).

2.4.1 Suppression-Warning Map. To understand the relationship between suppressions and warnings, we need to associate suppressions with the warnings they suppress:

[2] `git log -C -M -L <start,end>:target_file –first-parent`

```
1  class code_reader: # pylint: disable=invalid-name
2      def __init__(self, path):
3          self.path = path
```

(a) Old code: a suppression effectively hides a warning.

```
1  class CodeReader: # pylint: disable=invalid-name
2      def __init__(self, path):
3          self.path = path
```

(b) New code: a suppression now is useless.

Fig. 3. Example of a useless suppression (highlighted in red) caused by fixing the underlying problem.

Definition 4 (Suppression-warning map). *A suppression-warning map is a binary relation $M = S \times W$ between the set S of suppressions and the set W of warnings in a project at a specific commit. For a suppression $s \in S$ and a warning $w \in W$, $(s, w) \in M$ holds if and only if s suppresses w.*

Whether a suppression s suppresses a specific warning w depends on the semantics of the static analyzer. In the most simple case, $s = (f_s, l_s, k_s)$ suppresses $w = (f_w, l_w, k_w)$ because $f_s = f_w$, $l_s = l_w$, and $k_s = k_w$, i.e., both refer to exactly the same code location and the same kind of warning. Beyond this simple case, a suppression may also cover a broader scope, e.g., because the suppression is placed at the beginning of a file, class, or function and hence suppresses all warnings in it. Instead of trying to infer and reimplement the semantics of the static analyzer, we exploit the fact that Pylint offers an option for printing all currently suppressed warnings along with the suppression line, which we add into our suppression-warning map M.

2.4.2 Useless Suppressions. Given the suppression-warning map M, we can identify suppressions that have become *useless*, or stale, in the sense that despite being present in the code, the suppression does not suppress any warning.

Definition 5 (Useless suppression). *Given a suppression-warning map M, a suppression s is useless if and only if there is no warning w such that $(s, w) \in M$.*

As a concrete example, consider Figure 3, which shows a piece of code before and after a commit. The old version (Figure 3a) raises an "invalid-name" warning because the class name does not conform to Python's naming conventions. The developer suppresses this warning using the suppression in line 1. The new version (Figure 3b) fixes the underlying problem by renaming the class, but then forgets to remove the suppression, which becomes useless at this point.

Removing a useless suppression does not cause any additional warnings to be reported by the static analyzer. Instead, such a suppression unnecessarily clutters the code, and perhaps even worse, it may unintentionally hide new warnings in the future. For these reasons, we consider it important to identify and remove useless suppressions from the code.

2.5 Identifying Potentially Unintended Suppressions

RQ4 is about the phenomenon that a warning in newly added or modified code may get hidden due to an existing suppression. Such suppressions may be unintended because the developer never sees the new warning, and hence, does not take an explicit decision to suppress it.

Definition 6 (Potentially unintended suppression). *Suppose a suppression s present in two commits of a file, where the warnings suppressed in the older and the newer commit are given by the suppression-warning maps $M(s)$ and $M'(s)$, respectively. The suppression is potentially unintended if $|M(s)| < |M'(s)|$, i.e., the suppression hides more warnings in the newer than in the older commit.*

The definition focuses in the number of warnings suppressed by a suppression instead of comparing the individual warnings. The reason is that accurately tracking warnings across commits is a research problem on its own [36, 48]. As a result, our estimate of potentially unintended suppressions is an underestimate of the actual number.

To identify potentially unintended suppressions, we iterate through an extended version of all suppression histories (Definition 3) and compare the number of suppressed warnings for all pairs of consecutive commits that affect the file of a suppression. The extended suppression histories do not only include the change events at the beginning and end of a suppression's lifetime, but also all commits that affect the file in between, which are marked with a change operation "unchanged". For each commit in the extended history, we compute the suppression-warning map and then compare the number of suppressed warnings in the older and newer version of the file.

2.6 Understanding the Reasons for Using Suppressions

2.6.1 Data Collection. To address RQ5, i.e., when and why developers use suppressions, we consider three sources of information. First, for Python projects, we inspect commit histories. Searching for all commits that match a broad set of keywords ("lint", "warning", "suppress", and "disable"), results in 2,925 commits, i.e., too many to inspect manually. Our initial inspection also showed that many commits do not provide reasons for adding or removing suppressions. Based on this preliminary result, we refined our search to include commits that (i) contain "lint" in their commit message and that (ii) are either at the beginning or at the end of a suppression history (Definition 3). The first criterion is to find explanations related to linter-produced warnings and their suppressions. The second criterion is to focus on commits where suppressions are added or removed, as these are more likely to contain explanations. The refined search results in 116 commits. We select for the manual inspection all 23 commits that have suppressions removed and match that with 23 randomly selected commits with suppressions added, which add a total of 113 suppressions that suppress 176 warnings. For each of these commits, we inspect the code change, the commit message, and any relevant pull requests or issues. When unavailable, we clone the repository, reproduce the suppressed warnings, and assess potential reasons for using suppressions from code changes ourselves. As a result of this process, we extract specific explanations for 77 suppressions, which correspond to 154 suppressed warnings; we mark the remaining suppression as "unknown".

To further enrich the dataset and validate our manual analysis, we reached out to 35 developers who produced all the 116 identified commits and have publicly available contact information, asking them about reasons they used suppressions. We received 11 replies (31.4% response rate) which provided explanations for 31 suppressions, which correspond to 31 suppressed warnings in 11 commits. Interestingly, one of these commits is already included in our manual analysis and the result of our analysis is consistent with the developer's explanation in this case; the remaining 10 commits are new and are not included in the subset we analyzed manually. Combining the two datasets, we gather explanations for 143 suppressions affecting 206 warnings across 33 commits in Python projects.

We refer the developers who reply to us as D1–D11. Out of these developers, six shared demographic information with us: they are experienced software professionals, with 15 years of software development experience on average (ranging from 6 to 35 years) and 3.71 years of experience in their corresponding projects (ranging from 3 months to 10 years). Their ages range from 28 to 60 years old. Two developers are female and four are male. Five hold degrees in Computer Science (three Master's and two PhDs); one holds a degree in engineering.

As the final source of information, we analyze student reports for Java/JavaScript projects. The students were incentivized to fix all but "truly unuseful" warnings and provide a comprehensive justification for why the remaining warnings are not useful (and thus, should be suppressed going forward). As students were instructed to carefully justify the decision not to fix a particular static analysis warning in the reports they submit, we use these justifications in our study, referring to the warnings they left in their code as "suppressed warnings", for simplicity of presentation.

Table 1. Prevalence of suppressions for different warnings.

Unified warning kind	# Suppressions		
	Python	Java	JavaScript
Code complexity	447 (6.1%)	0 (0%)	2 (1.1%)
Convention violation	2,609 (35.5%)	53 (86.9%)	15 (8.4%)
Exception type is too broad	1,356 (18.4%)	0 (0%)	0 (0%)
Redundant language construct	196 (2.7%)	1 (1.6%)	51 (28.5%)
Runtime error and bug	767 (10.4%)	0 (0%)	1 (0.6%)
Security vulnerability	39 (0.5%)	0 (0%)	14 (7.8%)
Undefined element	760 (10.3%)	0 (0%)	95 (53.1%)
Unused element	1,020 (13.9%)	7 (11.5%)	1 (0.6%)
Others	163 (2.2%)	0 (0%)	0 (0%)
Total	7,357 (100.0%)	61 (100.0%)	179 (100.0%)

Starting from all 46 student projects, we exclude 16 projects with no warnings left. The remaining 30 projects had 274 warnings in total; we further exclude 34 warnings that were not fixed due to students' mistakes. We thus focus our analysis on 240 warnings for which there is a solid justification for keeping the original code. As multiple warnings in a project can have the same underlying reason, these warnings are further grouped into 64 unique justifications.

2.6.2 Data Analysis. We use all 207 collected explanations (Python: 143, Java: 27, JavaScript: 37) as an input to open coding, a qualitative data analysis technique borrowed from the grounded theory for deriving theoretical constructs from qualitative data [44]. For open coding, two authors of the paper independently read each explanation line by line and identify *concepts*, i.e., key ideas contained in the data for why the warnings are not fixed. When looking for concepts, we search for the best phrase that conceptually describes what we believe is indicated by the raw data. We further use axial coding [44] to group the identified concepts into *sub-categories* and *categories*, which represent *reasons* for suppressing warnings. In total, we identify nine sub-categories, which are further grouped into six higher-level categories indicating reasons for using suppressions. We describe them in detail in Section 3.5. All our results are grounded and linked to the underlying 176 explanations for using suppressions, which are available as part of our supplementary material.

3 Results

3.1 RQ1: Prevalence of Suppressions

We start by investigating how common suppressions are in the studied projects and what kinds of warnings developers typically suppress. All results on this RQ are obtained on the newest commit of each of the 46 projects found to use Pylint (Section 2.2). Overall, these projects comprise 45,446 Python files with a total of 6,690,087 lines of code (LoC). This code contains a total of 7,357 suppressions, which corresponds to a suppression in one out of six files, or 1.1 suppressions per 1,000 LoC. Looking at individual projects, the number of suppressions ranges from zero (in 14 out of 46 projects) to 3,200 (in the salt project). Suppressions tend to occur proportionally to the amount of code, with a Pearson correlation coefficient of 0.30.However, some projects have few suppressions despite their large size (e.g., rasa has 116,615 LoC but zero suppressions), whereas others have a relatively large number of suppressions (e.g., thumbor has 18,775 LoC and 255 suppressions).

To better understand what kinds of warnings developers typically suppress, we count the frequencies of kinds of each suppression across all studied Python and Java/JavaScript projects. We unify the kinds of warnings reported by the different static analyzers into a smaller set of categories,

Table 2. Python projects studied in RQ2 to RQ5.

Project	Python		Suppr.	Commits		Histories
	Files	LoC		Total	Studied	
buildbot	899	149,779	157	7,912	1,000	13
celery	340	58,644	87	7,009	1,000	99
clusterfuzz	660	96,880	308	2,439	1,000	237
dgl	1,180	189,502	395	3,210	1,000	564
dvc	521	58,409	213	5,975	1,000	42
kedro	299	34,004	104	1,083	1,000	540
magenta	332	44,830	184	1,280	1,000	189
PaddleNLP	2,465	401,420	32	2,771	1,000	17
thumbor	238	18,775	255	1,593	234	296
yapf	72	18,895	138	967	967	149
Total	7,006	1,071,138	1,873	34,239	9,201	2,146

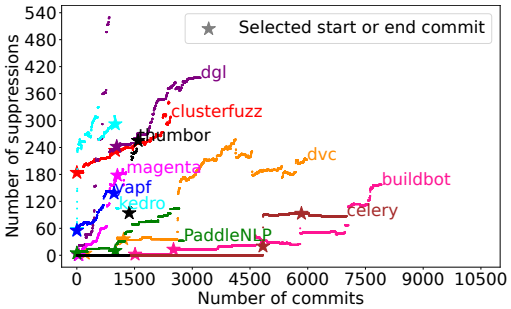


Fig. 4. Number of suppressions in commits of the main branch.

as shown in Table 1. The mapping from individual warning kinds to the unified warning kinds is available in our supplementary material. The table shows that the most commonly suppressed kinds of warnings are about conventions, such as naming conventions and code style, which account for 35.5% of all suppressions in the Python projects and 86.9% in the Java projects. Another common kind of suppressed warnings are about exceptions that are too broad, e.g., catching `Exception` instead of a more specific exception class, which accounts for 18.4% of all suppressions in the Python projects. In JavaScript code, the most commonly suppressed kinds of warnings are about undefined elements (53.1%), many of which are due to a warning by ESLint about using an undeclared variable.

Finding 1: Suppressions are relatively common in the studied projects, e.g., with a total of 7,357 suppressions across 46 Python projects, which corresponds to 1.1 suppressions per 1,000 lines of code. Many suppressions target warnings about coding conventions, too broad exception types, and undeclared variables.

3.2 RQ2: Evolution of Suppressions

We tackle questions about the evolution of suppressions on two levels. First, we consider the project-level evolution of suppressions, where we study how the number of suppressions evolves over time. Second, we consider the suppression-level evolution by studying the histories of individual suppressions. Because only the open-source Python projects provide sufficiently long and complex version histories, we focus on these projects for this research question. As described in Section 2.2, we focus on ten projects with long histories and a large number of suppressions (Table 2).

3.2.1 Project-Level Evolution. To illustrate the evolution of suppressions at the project level, Figure 4 shows how the number of suppressions changes across all commits in main branch of the studied projects. The overall trend is that, as the number of commits increases, the number of suppressions also increases. In other words, the studied projects accumulate more and more suppressions over time. For example, the builtbot project starts with no suppressions at all, and has accumulated 157 suppression in the newest studied commit. Contrary to the generally increasing trend, there occasionally are sudden jumps and drops of the number of suppressions in a project. To understand this phenomenon, we manually inspect commits that lead to a significant change in the number of suppressions, which leads to the following observations. First, the main reason for a sudden increase of suppressions is that the developers spend a focused effort on fixing warnings and suppressing

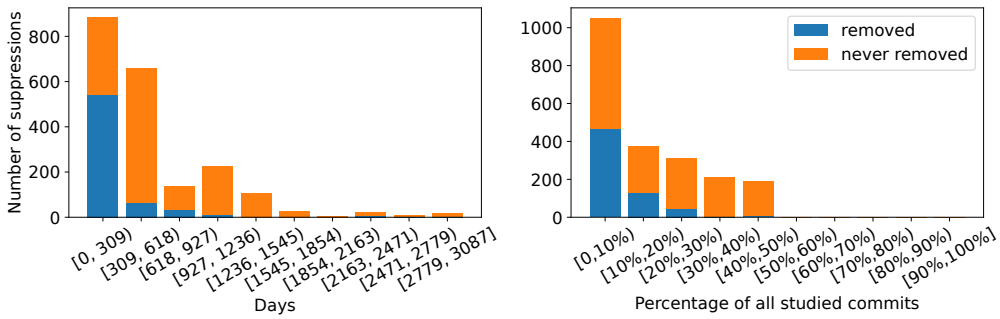


Fig. 5. Lifetimes of individual suppressions.

the remaining ones. For example, in the dvc project, the developers add 55 suppressions in a single commit[3]. Second, one common root cause for a sudden drop in the number of suppressions is deleting a large amount of code, which coincidentally also removes suppressions. For example, the dgl developers delete 199 files (155 Python files and some other files)[4], which causes a single commit to reduce the number of suppressions from 530 to 230. Third, another recurring reason for a sudden drop in the number of suppressions is that the developers fix previously suppressed warnings and remove useless suppressions. Finally, we also notice a project-specific reason for a sudden drop in the number of suppressions in the kedro project. At some point[5], the developers decided to switch from Pylint to another static analyzer, and along with this change, remove many suppressions targeted at Pylint, which reduces the total number of suppressions from 290 to 105.

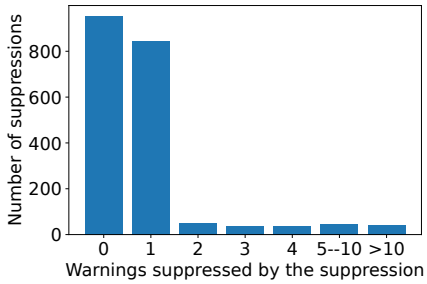
Finding 2: The number of suppressions in a project generally increases over time. There are occasional jumps, typically caused by developers carefully considering static analysis warnings in a focused effort, and also drops, typically caused by removing large amounts of code.

3.2.2 Histories of Individual Suppressions. Next, we consider the evolution of individual suppressions. We apply Algorithm 1 to commits in the main branch of the studied projects, which yields suppression histories (Definition 3). Because some projects have many more commits than others, we select a sequence of at most 1,000 consecutive commits per project, starting from the first commit that contains at least one suppression, and then apply the algorithm to these commits.

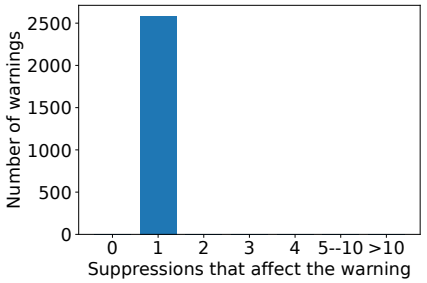
Table 2 shows the total number of commits in each project and the number of commits we select for studying individual suppressions. To connect the selected commits to the project-level evolution, the star symbols in Figure 4 show the starting and ending points of the commits selected for studying individual suppressions. The thumbor project has only 234 selected commits in total (Table 2) because the first commit with a suppression in this project has been relatively recent, i.e., there are only 234 commits for us to consider. Also note the buildbot project, which during the 1,000 selected commits has only 13 suppression histories, but 157 suppressions in the newest commit. Applying Algorithm 1 to the selected commits yields a total of 2,146 suppression histories. The last column of Table 2 shows how these histories are distributed across the studied projects.

Having histories of individual suppressions allows us to answer questions about the lifetime of suppressions. In particular, we are interested in how long suppressions typically remain in the code

[3]https://github.com/iterative/dvc/commit/1c0f5cdf
[4]https://github.com/dmlc/dgl/commit/36c7b771
[5]https://github.com/kedro-org/kedro/commit/638c01b6



(a) Number of warnings suppressed by a single suppression.



(b) Number of suppressions that affect a single warning.

Fig. 6. Relation between suppressions and warnings.

base. Figure 5 answers this question by illustrating the distribution of lifetimes in two ways. The left-hand side of the figure shows how many days suppressions remain in the code base, whereas the right-hand side of the figure normalizes the lifetimes relative to the number of studied commits. Both plots distinguish between suppressions that are still present in the newest studied commit ("never removed") and suppressions that have been removed at some point ("removed").

The figure allows for several observations. First, many suppressions exist only for a relatively short amount of time. For example, as shown by the first bar on the left of Figure 5, over 800 suppressions remain in the project for less than 309 days. Second, suppressions that get removed usually get removed relatively quickly. As shown by both plots in the figure, the fraction of "removed" suppressions quickly drops close to zero with increasing lifetimes. Finally, we observe that some suppressions remain in a project for a very long time. For example, the yapf project contains suppressions that were added in April 2015 and are still present in August 2023, despite active development throughout this entire period.

Finding 3: Suppressions often remain in a project for less than a year, but some suppressions remain in a project for a very long time (e.g., over 8 years).

3.3 RQ3: Relation Between Suppressions and Warnings

Next, we investigate the relationship between suppressions and warnings. As in RQ2, we focus on the open-source Python projects because the Java/JavaScript projects have a one-to-one relationship between suppressions and warnings. To determine which warnings a suppression hides, we compute the suppression-warning map (Definition 4) of the newest studied commit of each project. We then use this map to measure how many warnings are affected by each suppression. Figure 6a shows how many warnings are suppressed by a single suppression. Perhaps surprisingly, the figure reveals that commonly (50.8%) a suppression affects zero warnings. In other words, these suppressions are useless (Definition 5) and could be removed from the code without affecting the warnings reported by the static analyzer. The second-most common case 45% is a suppression that affects exactly one warning. Finally, there are also suppressions that affect more than one warning, with the largest number of warnings suppressed by a single suppression being 24.

Figure 6b shows the inverse relationship, i.e., how many suppressions affect a single warning. In principle, a warning can be suppressed by multiple suppressions, e.g., both a line-level and a function-level suppression. In practice, we find that all suppressed warnings are affected by exactly

Table 3. The number of suppressions (useless/all) at different levels.

Project	Number of suppressions at each level					Total
	File level	Class level	Function level	Block level	Line level	
buildbot	1/6	2/4	4/5	3/14	42/128	52/157
celery	0/0	0/0	1/13	7/32	40/42	48/87
clusterfuzz	1/2	1/1	6/6	27/92	57/207	92/308
dgl	17/33	0/1	2/3	197/306	18/52	234/395
dvc	2/21	0/1	8/12	14/30	72/149	96/213
kedro	0/11	0/0	0/2	0/12	3/79	3/104
magenta	0/1	0/0	5/16	31/50	65/117	101/184
PaddleNLP	6/6	0/0	3/4	3/3	14/19	26/32
thumbor	11/24	0/0	0/0	120/128	33/103	164/255
yapf	0/0	0/0	0/0	7/7	129/131	136/138
Overall	38/104	3/7	29/61	409/674	473/1,027	952/1,873

Table 4. Top-10 most common kinds of useful and useless suppressions.

Useful	Useless
protected-access (172)	invalid-name (151)
unused-argument (163)	no-member (124)
invalid-name (144)	broad-except (99)
broad-exception-caught (71)	arguments-differ (99)
line-too-long (68)	line-too-long (64)
import-outside-toplevel (40)	unused-argument (28)
unused-import (35)	attribute-defined-outside-init (27)
redefined-outer-name (30)	abstract-method (24)
no-member (27)	protected-access (21)
consider-using-with (19)	g-missing-super-call (21)

Table 5. Potentially unintended suppressions.

Project	Number of suppressions at each level					All
	File level	Class level	Function level	Block level	Line level	
buildbot	1	0	0	0	0	1
celery	0	0	0	4	0	4
clusterfuzz	16	1	0	5	3	24
dgl	7	2	0	2	0	11
kedro	10	0	1	7	1	19
Total	34	2	1	18	4	59

one suppression. This is good news, in the sense that developers are not unnecessarily adding multiple suppressions to hide a single warning.

Finding 4: Most suppressions that effectively suppress a warning suppress exactly one warning, but there are also suppressions suppress many warnings (up to 24 in our dataset). Surprisingly, 50.8% of all suppressions do not suppress any warning, i.e., they are practically useless.

To better understand the surprisingly common phenomenon of useless suppressions, we investigate them in more detail. Table 3 shows the number of useless suppressions and the total number of suppressions in each project. The numbers vary between projects, but overall useful and useless suppressions are roughly equally common: 921 useful and 952 useless. That is, about half of all suppressions actually have no impact anymore on the current commit of the code. Remarkably, each of the ten studied projects has at least a few useless suppressions, indicating that the problem is widespread. Useless suppressions range from 36.5% at the file level to 60.7% at the block level.

We also investigate which kinds of suppressions are most commonly useful and useless, as shown in Table 4. The results show some kinds of warning to be relatively common among both useful and useless suppressions, such as “invalid-name” and “line-too-long”. In contrast, some kinds

```
14   """Tests for cleanup task."""
15   # pylint: disable=protected-access
56       actual = cleanup._get_predator_result_item(
57           testcase, 'suspected_components', default=[])
279      self.issue{+._monorail_issue+}.open = False
1679     self.assertNotIn('is associated with an obsolete fuzzer',
1680                      self.issue{+._monorail_issue+}.comment)
```

Fig. 7. A potentially unintended, file-level suppression.

```
92   class RequestHandler([-BaseHTTPServer-]{+http.server+}.BaseHTTPRequestHandler):
121      def log_message(self, fmt, *args): # pylint: disable=arguments-differ
122          """Do not output a log entry to stderr for every request made."""
123          pass
```

Fig. 8. A potentially unintended, line-level suppression.

of suppressions are more likely to be useful than useless, e.g., "protected-access" and "unused-argument", whereas the inverse holds, e.g., for "no-member". A special case is the "broad-except" kind, which accounts for a total of 99 useless suppressions, but no useful suppressions. The reason is that "broad-except" has been renamed to "broad-exception-caught" at some point by the developers of Pylint, but several projects have not updated their existing suppressions to the new name.

Finding 5: Each of the studied projects has at least a few useless suppressions, and some projects even have multiple hundreds, showing that the problem is widespread. The most common kinds of useful suppressions differ from the most common kinds of useless suppressions.

3.4 RQ4: Potentially Unintended Suppressions

Table 5 shows the projects for which we identify potentially unintended suppressions. Our automated methodology (Section 2.5) identifies 63 cases, which we further check manually. The manual check confirms that 59 of these cases are indeed potentially unintended suppressions. We exclude the four remaining cases, because they are due to a single line being split into multiple lines, which turns a single warning into multiple warnings, without introducing any really new warning.

Figure 7 shows a potentially unintended, file-level suppression (note the line numbers). It initially suppresses three warnings, but later hides 56 more warnings. While the three old warnings are all related to `._get_predator_result_item`, the additionally suppressed warnings are all about another attribute `_monorail_issue`. The newly suppressed warnings occur between lines 279 and 1680, i.e., at least 264 lines away from the suppression at line 15, suggesting that the developer did not actively decide to suppress these warnings.

Examples like Figure 7 motivate us to study the line distance between a newly suppressed warning and the corresponding suppression, as a way of assessing whether developer are likely aware of the suppression. Our analysis reveals that only 46 warnings are up to 10 lines away, making the suppression easily visible to developers. Another 115 warnings are between 11 and 100 lines away, which developers might notice while scrolling. Yet, there are 558 warnings that are between 101 and 1,000 lines away, and another 411 warnings that are even further away from the suppression. In these latter cases, developers need to either remember the suppressions or actively search for them.

To better understand the potential risks of unintentional suppressions, we further study their properties. Out of the 59 cases, six suppressions are useless at some point, but then suddenly suppress

newly introduced warnings. Figure 8 shows one such case where the class body is unchanged, but its base class was changed (line 92), causing the useless suppression at line 121 to suddenly suppress a warning. Such examples show that keeping useless suppressions in a code base not only clutters the code but also can lead to unintentional suppression of new warnings. We also find that for 36 out of the 59 cases, different authors have edited the file in the commits where the suppression was introduced and where the additional warnings were suppressed. Moreover, we find that the time gap between the commits ranges from 0 to 1,491 days, with an average of 230.8 days. These results suggest that the developers who are not seeing a specific warning may be unaware of the suppression that hides that warning.

Finding 6: Half of the projects have potentially unintended suppressions, which may cause developers to overlook important warnings hidden by suppressions that are many lines away and often added by another developer. In particular, some useless suppressions later on hide newly added warnings, indicating the need to remove useless suppressions.

3.5 RQ5: Reasons for Using Suppressions

3.5.1 When and How Suppressions Are Added and Removed. Our investigation on when and how developers add suppressions shows that commits frequently (17/23) add suppressions along with newly added code, oftentimes (16 out of the 17) in a separate commit that eventually gets combined into a single merge commit. We also observe a few cases where suppressions are added without touching the surrounding code, typically to address warnings introduced at an earlier point in time. As shown in RQ2, many suppressions are never removed from the code base. When developers remove suppressions, they do it either coincidentally when performing a larger code change that has another purpose (9/23) or because the suppression has become useless (8/23). Interestingly, all removed useless suppressions we observe were already useless in their parent commit, i.e., they could (and ideally should) have been removed earlier. We also observe that developers sometimes remove suppressions after fixing the root cause of the suppressed warning (5/23), i.e., the suppressions have served as a way to post-pone fixing a problem.

Finding 7: Developers mostly add suppressions along with new code, in an explicit effort to reduce new warnings. The most common reason for intentionally removing suppressions is that the suppression has become useless. However, it is even more common to coincidentally remove suppressions as part of another code change.

3.5.2 Why Suppressions Are Used. Table 6 shows the six higher-level categories, further divided into nine sub-categories, which describe the main reasons for suppressing warnings. We also specify the number of warnings that belong to each category, for Python, Java, and JavaScript separately. As a warning can have multiple reasons for being suppressed, it can be counted in multiple categories; thus, the number of warnings in all categories does not sum up to 446.

R1: Static analyzer false positives. False positives are the most common reason for using suppressions (34.4% of all cases). We further classified them into two sub-categories:

R1.1: Language support. Incorrect warnings in this sub-category are caused by "classical" limitations of static analysis techniques, such as dealing with reflection, incorrect parsing of wildcards in imports, incorrect AST generation, and versions of the language. For example, one developer mentions: «[...] so sometimes we have to make compromises and ignore linting warnings that simply can't be solved for all python versions at the same time» (D7). As another example, 51 false positives in JavaScript are due to the last reason: a JavaScript AST parser, Rhino [12], that the PMD

Table 6. Reasons for using suppressions.

Reasons	# Warnings			
	Python	Java	JavaScript	Total
R1: Static analyzer false positives	**63 (13.7%)**	**42 (9.2%)**	**53 (11.5%)**	**158 (34.4%)**
R1.1: Language support	48 (10.5%)	5 (1.1%)	53 (11.5%)	106 (23.1%)
R1.2: Framework/library support	15 (3.3%)	37 (8.1%)	0 (0%)	52 (11.3%)
R2: Imprecise warning messages	**12 (2.6%)**	**1 (0.2%)**	**18 (3.9%)**	**31 (6.8%)**
R3: Inaccurate configuration	**0 (0%)**	**0 (0%)**	**97 (21.1%)**	**97 (21.1%)**
R4: Correct warnings in third-party code	**9 (2%)**	**5 (1.1%)**	**14 (3.1%)**	**28 (6.1%)**
R5: Developer decision	**100 (21.8%)**	**13 (2.8%)**	**10 (2.2%)**	**123 (26.8%)**
R5.1: Work-in-progress code	1 (0.2%)	13 (2.8%)	1 (0.2%)	15 (3.3%)
R5.2: Custom coding conventions	73 (15.9%)	0 (0%)	1 (0.2%)	74 (16.1%)
R5.3: High effort, low importance	26 (5.7%)	0 (0%)	8 (1.7%)	34 (7.4%)
R6: Unknown	**22 (4.8%)**	**0 (0%)**	**0 (0%)**	**22 (4.8%)**
Total	**206 (44.9%)**	**61 (13.3%)**	**192 (41.8%)**	**459 (100%)**

```
app.get('/events', async (req, res) => {
    try {
        const result = await client.db('db').collection('event').find().sort({ 'rating' : -1 }).toArray()
        res.status(200).send(result)
    }
    catch(err) {
        res.send(400).send('No Events!')
 }})
```

Fig. 9. PMD false positive: “Unnecessary block”.

analyzer uses cannot handle `async` and `await` keywords correctly, omitting some statements within these blocks. This causes the tool to incorrectly mark the try/catch block as unnecessary in the example in Figure 9.

R1.2: Framework/library support. Linters often incorrectly deal with code that uses a particular library or framework (11.3% of all cases). For example, PMD deems tests from the Android Espresso UI testing framework [1] built on top of JUnit [8] as standard JUnit tests. This leads to false positives for Espresso-specific assertions.

R2: Imprecise warning messages. In a few cases (6.8%), the analysis correctly alerts on a potential issue in the code but provides an inaccurate message, misleading the developers and causing them to believe that the warnings should not be fixed. For example, when JavaScript constructors are called as regular functions, without the `new` operator, the warning suggests that the constructor methods should be renamed instead of suggesting to explicitly add the `new` operator.

R3: Inaccurate configuration. A large fraction of suppressed warnings (21.1%) could be avoided with a more nuanced configuration of analyzers. For example, a warning about exceeding the cyclomatic code complexity [4] can be avoided by changing the default complexity threshold set by Codacy from 4 to, say, 10, as proposed in the literature [40].

R4: Correct warnings in third-party code. In 6.1% of cases, the analysis warnings are correct, but these warnings are in the third-party code that was imported by developers, and hence, cannot be easily modified. One example is the Flipper [6] debugging platform for React Native, which provides a boilerplate Java class named `ReactNativeFlipper`. To enable Flipper in their project,

application developers must add this boilerplate Java class to their code, even though they do not have control over the class implementation.

R5: Developer decision. In another 26.8% of the cases, developers understand and agree with the reported warnings but decide not to fix them due to the following reasons:

R5.1: Work-in-progress code. These warnings are expected to disappear in later development stages, e.g., when a field is added and is still used in one method only.

R5.2: Custom coding conventions. Sometimes developers prefer custom coding conventions, e.g., to avoid turning `{poster: poster, date: today, comment: comment}` into the shorter but less consistent `{poster, date: today, comment}` even though the latter is suggested by ESLint when the key name matches the name of the assigned variable. As another example, Pylint warns that a developer-defined function `getvalue()` does not follow the suggested camel-case style; D5 suppresses the warning to preserve consistency with the naming convention of an external library: «I wanted to use the same name [...] to be consistent with the underlying object [from a library].»

R5.3: High effort, low importance. Fixing these warnings requires significant work which developers deem of low importance, e.g., adding documentation to 21 Python methods that developers believe are sufficiently self-explanatory.

R6: Unknown. For 4.8% of all cases we could not reliably recover the reasons for using suppressions.

Finding 8: The main reasons developers suppress warnings include false positives resulting from incomplete language and development framework support, imprecise and misleading messages, improper configurations, and warnings deemed not important enough.

4 Discussion

4.1 Implications for Developers

Using suppressions is a well-established practice. Given the overall prevalence of suppressions in popular, open-source Python projects, it is fair to say that using suppressions is a well-established practice. That is, developers should not feel bad about using suppressions, but rather see them as a mechanism that allows them to make best use of static analyzers, despite their limitations.

Awareness of useless suppressions. As shown in RQ3, over half of all suppressions are useless, i.e., they do not suppress any warning. The answer to RQ4 further reveals the potential risk that suppressions, including the useless ones, could unintentionally hide future warnings. Developers should be aware of this fact and remove useless suppressions from their code base.

4.2 Implications for Creators of Static Analyzers

Warnings in third-party code. As shown in RQ5, some warnings are caused by problems in third-party code, e.g., libraries or frameworks. While such warnings offer valuable insights into potential risks, developers often cannot fix the underlying problems and may instead resort to suppressing them. To balance the benefits of these warnings without overwhelming developers with non-actionable noise, static analyzers should recognize third-party code and categorize these warnings under a label like “third-party”, while also marking the warnings that are fixable within the project’s codebase. This allows developers to make informed decisions about seeking updates, patches, or replacements for the libraries, or to suppress the warnings under the “third-party” label.

Avoiding warnings that are commonly suppressed. Motivated by the observation that warnings about naming conventions and stylistic issues are particularly often suppressed, future analyzers should be aware of the conventions and style used in the project, instead of imposing general rules.

Better warning messages. As shown in RQ5, some warnings are not fixed because developers misunderstand the warning message. In these cases, the static analyzer correctly identifies a

problem, but because the warning message fails to convey the problem to the developer, the developer suppresses the apparently spurious warning. Improving warning messages will help existing analyzers to live up to their full potential.

Framework-specific analysis and configuration. As shown in RQ5, some warnings are due to the lack of framework-specific analysis and an incorrect, framework-specific configuration of the analyzer. Future analyzers should try to automatically detect the frameworks used in a project and automatically configure themselves accordingly.

4.3 Implications for Creators of Development Tools

Resurfacing suppressed warnings. As shown in RQ2, some suppressions remain in a code base for a very long time, and the number of suppressions in a project tends to continuously increase. This leads to a risk of overlooking problems that a developer wanted to temporarily suppress, but then forgot about. Future development tools could resurface suppressed warnings after a certain amount of time, giving developers a chance to reconsider their decision.

Automatic repair of code and removal of suppressions. The relatively large number of suppressions in widely used code bases (RQ1), which includes many useless suppressions (RQ3), calls for automated techniques to repair the underlying code and remove the useless suppressions. To the best of our knowledge, there currently is only a single analyzer, Pylint, that offers an option to report useless suppressions (but it is disabled by default), while others, e.g., Mypy, ErrorProne, or Flow, lack such a mechanism. Moreover, we are not aware of any tools to automatically remove useless suppressions. Based on our results, most developers seem unaware of useless suppressions and would likely benefit from tools to find and remove them. Ideally, future techniques would automatically determine whether a suppression can be removed by fixing the underlying problem, or whether a suppression is useless and can be removed while affecting neither the static analysis warnings nor the behavior of the code. As shown in RQ5, practitioners currently perform such tasks manually, but this is a tedious and error-prone process.

Cross-analysis suppression mechanisms. As observed in RQ2, projects sometimes switch from one static analyzer to another. Currently, suppressions added for one analyzer may not be recognized by another analyzer, which means that developers have to add suppressions again when switching analyzers. Future techniques could (i) support developers in transitioning from one analyzer to another by automatically converting suppressions from one format to another, and (ii) support suppression formats recognized by multiple analyzers.

4.4 Limitations and Threats to Validity

All findings are limited to the studied subjects and may not generalize to other projects, static analyzers, or languages. To mitigate this threat, we study three languages, four static analyzers, and select projects from different domains. Selecting projects based on popularity (measured by the number of stars) may exclude less well-maintained projects, potentially limiting the generalizability of our findings. However, this criterion helps ensure that the selected projects follow state-of-the-art practices regarding their use of suppressions. The student projects may not be representative for professionally developed code, but because they were produced as part of graded assignments and further filtered by us, are likely of good quality. When extracting suppressions from Python projects, our methodology ignores warnings filtered via configuration files, which 26 of 46 studied Python projects use. Future work could investigate how code-level suppressions and global configurations of static analyzers interact. Additionally, our algorithm for extracting suppression histories might incorrectly track a suppression across commits. To validate the correctness of the extracted histories, we inspect a random sample of 20 extracted suppression histories, which we find to be all correct.

5 Related Work

5.1 Scalable and Lightweight Static Analyzers

There are various static analyzers, e.g., the original lint tool [29], FindBugs [25], ErrorProne [13], Infer [18], PMD [10], ESLint [5], and Pylint [11], including commercial tools, e.g., Coverity [17]. Our study focuses on Pylint [10], Checkstyle [2], ESLint [5], and PMD [10], which we consider to be representative for a wide range of analyzers.

5.2 Studies of Static Analyzers

Studies investigate recall [24, 45] and precision [42, 50] of analyzers, or how they are typically integrated into the development process [16, 49]. These studies show that projects sometimes suppress entire categories of warnings, whereas we focus on fine-grained suppressions via code comments or annotations. Other studies try to understand the expectations developers have about static analyzers [19, 27, 28]. They find that how to filter and suppress warnings is the most common question [27] and that suppressing individual warnings is important to developers [19]. The importance of suppressions is also confirmed by a study of how developers act on static analysis warnings [26]. While some existing work refers to the importance of suppressions, our work is the first to investigate in-depth how developers use suppressions in practice.

5.3 Improving Static Analyzers

Techniques for improving static analyzers filter false positives, e.g., based on a learned models [32, 38], prioritizes warnings, e.g., based on the frequency of true positives and false positives [34], version histories [33], user reviews [47], or continuous feedback from developers [39, 43], and validates warnings, e.g., by generating tests [30]. Other work automatically modifies code to tailor or prevent static analysis warnings [46], which can be seen as an alternative to suppressing warnings. Our findings motivate future work on improving static analyzers, e.g., by avoiding warnings that are commonly suppressed and by improving warning messages (Section 4).

5.4 Code Evolution

RQ2 relates to prior work on code evolution, such as studies of the evolution of type annotations [21], repetitive code changes [41], and concurrency problem [23], and to techniques that automatically mine frequent changes from version histories [20]. To the best of our knowledge, the evolution of suppressions has not yet been studied. Our algorithm for computing suppression histories relates to prior work on tracking warnings [31, 48], code lines related to a bug [14], or code elements [22] across commits. Some of that work also uses "git log" [14, 48]. Our work differs by focusing on the complementary problem of tracking suppressions.

6 Conclusions

This paper presents the first in-depth empirical study of suppressions of static analysis warnings. We investigate five major questions related to the prevalence of suppressions, their evolution, their usefulness, the potential risk of unintentional suppressing, and the reasons for using suppressions. Our results show that suppressions are a relatively common phenomenon (e.g., 7,357 suppressions in about 6.69 million lines of code), that the number of suppressions in a project tends to continuously increase over time, that many suppressions (50.8%) are useless, that some suppressions potentially and unintentionally hide future warnings, and that developers use suppressions to address false positives, because warning messages are imprecise, and to post-pone fixing a problem. These findings have implications for static analysis and software engineering practices in general, such as a call for better support for handling suppressions that become useless over time, better warning

messages, and cross-analysis suppression mechanisms. We envision our study to be a starting point for future work on improving the way suppressions are handled during the development process.

Data Availability

Our source code and data are available at https://github.com/sola-st/suppression_study/.

Acknowledgments

This work was supported by the European Research Council (ERC, grant agreements 851895 and 101155832), by the German Research Foundation within the ConcSys, DeMoCo, and QPTest projects, and by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] [n. d.]. Android Espresso UI Testing Framework. https://developer.android.com/training/testing/espresso.
[2] [n. d.]. Checkstyle. https://checkstyle.sourceforge.io/.
[3] [n. d.]. Codacy. https://www.codacy.com/.
[4] [n. d.]. Cyclomatic complexity. https://en.wikipedia.org/wiki/Cyclomatic_complexity.
[5] [n. d.]. ESLint. https://eslint.org/.
[6] [n. d.]. Flipper - Extensible mobile app debugger. https://fbflipper.com.
[7] [n. d.]. Flow: Static Type Checker for JavaScript. https://flow.org/. accessed: 2018-09-12.
[8] [n. d.]. JUnit4. https://junit.org/junit4/.
[9] [n. d.]. Mypy. https://mypy-lang.org/.
[10] [n. d.]. PMD. http://pmd.sourceforge.net.
[11] [n. d.]. Pylint. pylint.readthedocs.io/.
[12] [n. d.]. Rhino. https://github.com/mozilla/rhino.
[13] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *12th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2012, Riva del Garda, Italy, September 23-24, 2012*. 14–23. https://doi.org/10.1109/SCAM.2012.28
[14] Gabin An, Jingun Hong, Naryeong Kim, and Shin Yoo. 2023. Fonte: Finding Bug Inducing Commits from Failures. In *Proceedings of the 45th International Conference on Software Engineering*. 589–601. https://doi.org/10.1109/ICSE48619.2023.00059
[15] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130
[16] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 470–481. https://doi.org/10.1109/SANER.2016.105
[17] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75. https://doi.org/10.1145/1646353.1646374
[18] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium*. Springer, 3–11.
[19] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 332–343. https://doi.org/10.1145/2970276.2970347
[20] Malinda Dilhara, Danny Dig, and Ameya Ketkar. 2023. PYEVOLVE: Automating Frequent Code Changes in Python ML Systems. In *ICSE*. https://doi.org/10.1109/ICSE48619.2023.00091
[21] Luca Di Grazia and Michael Pradel. 2022. The Evolution of Type Annotations in Python: An Empirical Study. In *ESEC/FSE*. https://doi.org/10.1145/3540250.3549114
[22] Felix Grund, Shaiful Alam Chowdhury, Nicholas Bradley, Braxton Hall, and Reid Holmes. 2021. CodeShovel: Constructing Method-Level Source Code Histories. In *ICSE*. https://doi.org/10.1109/ICSE43902.2021.00135
[23] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. 2015. What change history tells us about thread synchronization. In *ESEC/FSE*. 426–438. https://doi.org/10.1145/2786805.2786815
[24] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *ASE*. https://doi.org/10.1145/3238147.3238213

[25] David Hovemeyer and William Pugh. 2004. Finding bugs is easy. In *Companion to the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 132–136. https://doi.org/10.1145/1052883.1052895
[26] Nasif Imtiaz, Brendan Murphy, and Laurie Williams. 2019. How do developers act on static analysis alerts? an empirical study of coverity usage. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 323–333. https://doi.org/10.1109/ISSRE.2019.00040
[27] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. 2019. Challenges with responding to static analysis tool alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 245–249. https://doi.org/10.1109/MSR.2019.00049
[28] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681. https://doi.org/10.1109/ICSE.2013.6606613
[29] S. C. Johnson. 1978. Lint, a C Program Checker.
[30] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating Static Warnings via Testing Code Fragments. In *ISSTA*. https://doi.org/10.1145/3460319.3464832
[31] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting false alarms from automatic static analysis tools: how far are we?. In *Proceedings of the 44th International Conference on Software Engineering*. 698–709. https://doi.org/10.1145/3510003.3510214
[32] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. 2022. Learning to Reduce False Positives in Analytic Bug Detectors. In *ICSE*. https://doi.org/10.1145/3510003.3510153
[33] Sunghun Kim and Michael D Ernst. 2007. Which warnings should I fix first?. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 45–54. https://doi.org/10.1145/1287624.1287633
[34] Ted Kremenek and Dawson R. Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *International Symposium on Static Analysis (SAS)*. Springer, 295–315.
[35] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65. https://doi.org/10.1145/3318162
[36] Junjie Li and Jinqiu Yang. 2024. Tracking the Evolution of Static Code Warnings: The State-of-the-Art and a Better Approach. *IEEE Trans. Software Eng.* 50, 3 (2024), 534–550. https://doi.org/10.1109/TSE.2024.3358283
[37] Bo Liu, Hui Liu, Nan Niu, Yuxia Zhang, Guangjie Li, and Yanjie Jiang. 2023. Automated Software Entity Matching Between Successive Versions. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1615–1627. https://doi.org/10.1109/ASE56229.2023.00132
[38] Kui Liu, Dongsun Kim, Tegawendé F Bissyandé, Shin Yoo, and Yves Le Traon. 2018. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2884955
[39] Anton Ljungberg, David Åkerman, Emma Söderberg, Gustaf Lundh, Jon Sten, and Luke Church. 2021. Case study on data-driven deployment of program analysis on an open tools stack. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 208–217. https://doi.org/10.1109/ICSE-SEIP52600.2021.00030
[40] Thomas J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* 2, 4 (Dec. 1976), 308–320.
[41] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *European Software Engineering Conference and International Symposiumon Foundations of Software Engineering (ESEC/FSE)*. ACM, 268–278. https://doi.org/10.1145/2491411.2491416
[42] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, 245–256. https://doi.org/10.1109/ISSRE.2004.1
[43] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608.
[44] Anselm Strauss and Juliet Corbin. 1998. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Thousand Oaks, CA: Sage.
[45] Ferdian Thung, Lucia, David Lo, Lingxiao Jiang, Foyzur Rahman, and Premkumar T. Devanbu. 2012. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *Conference on Automated Software Engineering (ASE)*. ACM, 50–59. https://doi.org/10.1145/2351676.2351685
[46] Rijnard van Tonder and Claire Le Goues. 2020. Tailoring programs for static analysis via program transformation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 824–834. https://doi.org/10.1145/3377811.3380343

[47] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2017. Oasis: prioritizing static analysis warnings for android apps based on app user reviews. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 672–682. https://doi.org/10.1145/3106237.3106294

[48] Ping Yu, Yijian Wu, Xin Peng, Jiahan Peng, Jian Zhang, Peicheng Xie, and Wenyun Zhao. 2023. ViolationTracker: Building Precise Histories for Static Analysis Violations. In *ICSE*. https://doi.org/10.1109/ICSE48619.2023.00171

[49] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. 334–344. https://doi.org/10.1109/MSR.2017.2

[50] Jiang Zheng, Laurie Williams, Nachiappan Nagappan, Will Snipes, John P Hudepohl, and Mladen A Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE transactions on software engineering* 32, 4 (2006), 240–253. https://doi.org/10.1109/TSE.2006.38

Received 2024-09-13; accepted 2025-01-14