APPENDIX A
BUG FIX PROCESS

A fix pattern is used as a guide to fix a bug, of which fixing process is defined as a *bug fix process*.

Definition 7. Bug Fix Process (FIX): A *bug fix process* is a function of fixing a bug with a set of fix patterns.

$$FIX : (bc, FP^{+}) \rightarrow P^{*} \tag{7}$$

where *bc* is the code block of a bug. FP^{+} means a set of fix patterns, and some of them could be applied to *bc*. P^{*} is a set of patches for *bc*, which is generated by the bug fix process. A *bug fix process* is specified by a *bug fix function* in this study.

Definition 8. Bug Fix Function (FixF): A *bug fix function* consists of two domains and three sub functions. They can be formalized as:

$$FixF : (bc, FP^{+}) = CtxF + M + R \rightarrow P^{*} \tag{8}$$

$$CtxF : bc \rightarrow Ctx_{bc} \tag{9}$$

$$M : (Ctx_{bc}, Ctx_{fp} \in FP^{+}) \rightarrow FP^{*} \tag{10}$$

$$R : (bc, Ctx_{bc}, CO \in FP^{*}) \rightarrow P^{*} \tag{11}$$

where *bc* is the code block of a given bug and FP^{+} is a set of fix patterns. *CtxF* denotes the function of converting buggy code into a code context (i.e., Ctx_{bc}). *M* means the matching function of matching the code context of the given buggy code block with fix patterns to find appropriate fix patterns (i.e., FP^{*}, $FP^{*} \subseteq FP^{+}$) for the bug, where Ctx_{fp} is the code context of a fix pattern. If $FP^{*} = \emptyset$, it indicates that there is no fix pattern matched for the bug in the whole set of fix patterns. *R* represents the function of repairing the bug with change operations (i.e., CO) in matched fix patterns. If $P^{*} = \emptyset$, it indicates that there is no any patch which could be generated by the provided fix patterns and pass test cases of the bug.

APPENDIX B
STATISTICS ON VIOLATIONS IN THE WILD

In this section, we present the distributions of violations from three aspects of violations: number of occurrences, spread in projects and category. There are 16,918,530 distinct violations distributed throughout 400 types in our dataset. We investigate which violation types are common by checking their recurrences in terms of quantity (i.e., how many times they occur overall) and in terms of spread (i.e., in how many projects they occur).

Common types by number of occurrences.

Figure 24 shows the quantity distributions of all detected violation types. The x-axis represents arbitrary id numbers assigned to violation types following the number of times that occur in our dataset. The id mapping considered in this figure by sorting occurrences (i.e., id=1 corresponds to the most occurring violation type) will be used in the rest of this paper unless otherwise indicated. The *Order_1* of Table 14 presents the mapping of top 50 types. The whole mapping is available at the aforementioned website for interested readers.

It is noted from the obtained distribution that violation occurrences for the top 50 violation types account for 81.4% of all violation occurrences. These types correspond only to about 12% of `FindBugs` violation types. These statistics corroborate our conjecture that most violation instances are associated with a limited subset of violation types.

Figure 24 further highlights the category of each violation type according to the categorization by `FindBugs`. We note that all categories are represented among most and least occurring violations alike.

TABLE 13: Category Distributions of Violations

Category	# Violation instances	# Violation types			# Projects
		top-50	top-100	All	
Dodgy code	**6,736,692**	22	29	**75**	**703**
Bad practice	**4,467,817**	11	34	**86**	**696**
Performance	1,822,063	8	13	29	**685**
Malicious code vulnerability	1,774,747	4	8	17	**634**
Internationalization	740,392	2	2	2	**632**
Multithreaded correctness	602,233	2	4	44	517
Correctness	542,687	0	6	**131**	**636**
Experimental	135,559	1	2	3	446
Security	95,258	0	2	11	219
Other	1,082	0	0	2	51

Common types by spread in projects

Figure 25 illustrates to what extent the various violation types appear in projects. The id numbers for violation types are from the mapping produced before (i.e., as in Figure 24). Almost 200 (50%) violation types have been associated with over 100 (about 14%) projects. It is further noted that there is no correlation between the spread of a violation type and its number of occurrences: some violation types among the most widespread types (e.g., top-50) actually occur less than some lesser widespread ones. Nevertheless, the data indicate that, together, the top-50 most widespread violations account also for the majority of violation instances.

Category distributions of violations

Table 13 provides the statistics on the categories of violation types regrouped in the `FindBugs` documentation. The ranking of violation types is based on overall occurrences as in Figure 24. Category `Other` contains `SKIPPED_CLASS_TOO_BIG` and `TESTING` that actually are not violation types defined in `FindBugs`. In the remainder of our experiments, instances of the two types are ignored.

`Dodgy code` and `Bad practice` appear as the top two most common categories in terms of occurrence and spread. `Security` violations are the least common, although they could be found in 30% of the projects.

In terms of violation types, `Correctness` regroups the largest number of types, but its types are not among the top occurring. Figure 26 illustrates the detailed distributions of categories. The number of violation types of `Correctness` increases sharply from the ranking 100 to 400, while there

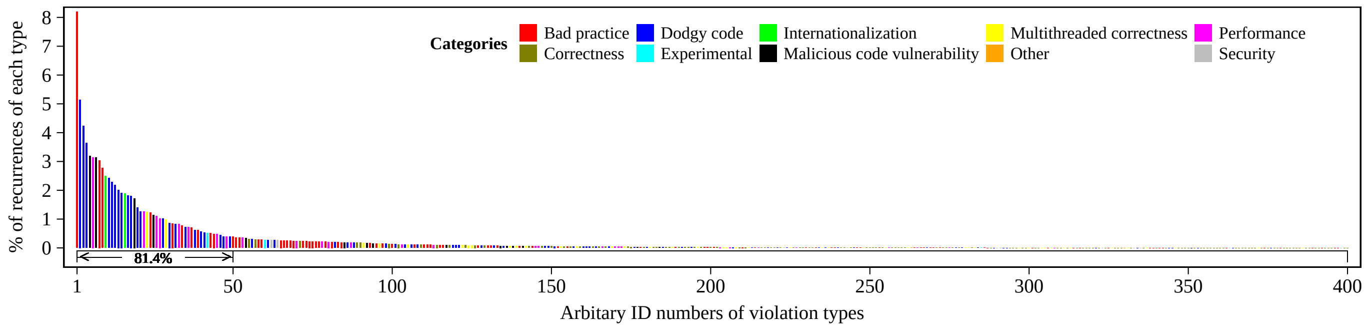

Fig. 24: Quantity distributions of violation types sorted by their occurrences. The x-axis represents arbitrary id numbers assigned to violation types. The y-axis represents the percentages of their occurrences in all violations.

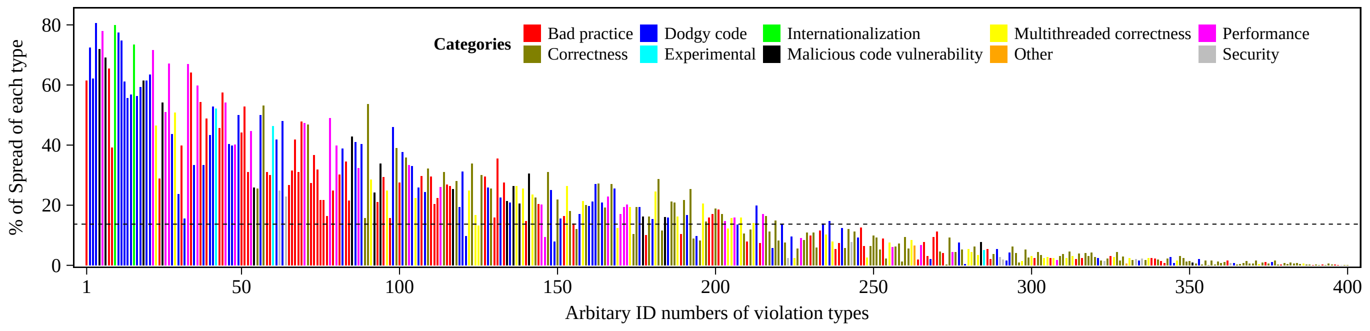

Fig. 25: Spread distributions of all violation types. The order of id numbers on x-axis is consistent with the order in Figure 24. The y-axis represents the percentages of their spread in all projects.

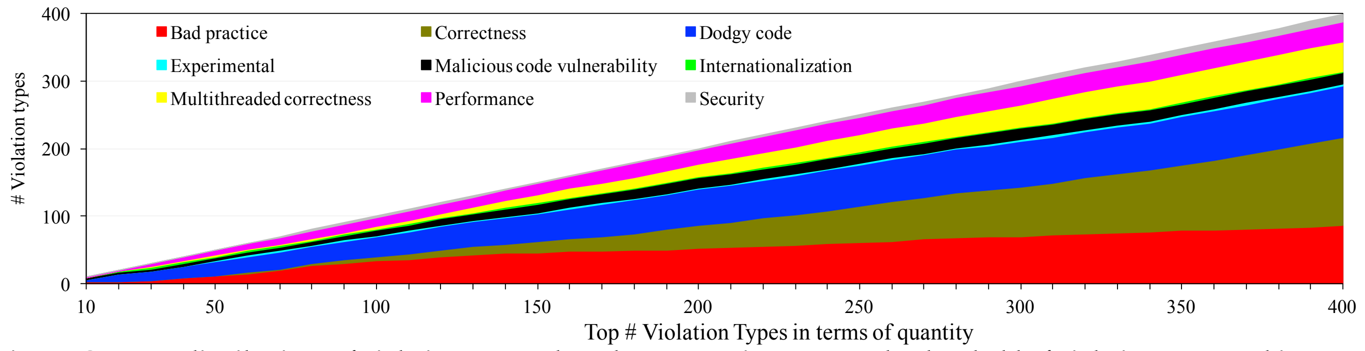

Fig. 26: Category distributions of violation types. The values on x-axis represent the threshold of violation types ranking used to carry out the category distributions, of which the order is consistent with the order in Figure 24. For example, 50 means that top 50 violation types are used to carry out the category distributions.

is no correctness-related violation type in top 50 types. The violation types out of top 100 have much lower number of occurrences compared against top 100 types. Thus, `Correctness` has a low number of overall occurrences, although it contains a large number of violation types and is seen in many projects. These findings suggest that developers commit few violations of these types.

Overall, `Dodgy code` and `Bad practice` are the top two most common categories. `Internationalization` is also found to be common since it contains only two violation types (i.e., `DM_CONVERT_CASE` and `DM_DEFAULT_ENCODING`) which are among top-20 most occurring violation types and among top-10 most widespread ones throughout projects.

`Dodgy code` represents either confusing, anomalous, or error-prone source code [108]. Figure 27 shows an example of a fixed `Dodgy code` violation, which is a fixed violation of `BC_VACUOUS_INSTANCEOF` type that denotes that the `instanceof` test would always return `true`, unless the value being checked was `null` [108]. Although this is safe, make sure it is not an indication of some misunderstandings or some other logic errors. If the programmer really wants to check the value for being `null`, it would be clearer and better to do a `null` check rather than an `instanceof` test. Consequently, this violated instance is fixed by replacing the `instanceof` test with a `null` check.

`Bad practice` means that source code violates recommended coding practices [108]. The fixed violation in Figure 2 is an example of a corrected `Bad practice` violation. It is not recommended to ellipsis an `instanceof` test when implementing an `equals(Object o)` method, so that this violation is fixed by adding an `instanceof` test.

[30]https://github.com/antlr/stringtemplate4
[31]https://github.com/apache/httpclient

Violation Type: `BC_VACUOUS_INSTANCEOF`.

Fixing DiffEntry:

```
- if (code.strings[poolIndex] instanceof String) {
+ if (code.strings[poolIndex] != null) {
```

Fig. 27: Example of a fixed `Dodgy code` **violation taken from** ***BytecodeDisassembler.java*** **file within Commit** `e2713c` **in project** `ANTLR Stringtemplate4`[30].

Violation Type: `DM_CONVERT_CASE`.

Fixing DiffEntry:

```
- cookieDomain = domain.toLowerCase();
+ cookieDomain = domain.toLowerCase(Locale.ENGLISH);
```

Fig. 28: Example of a fixed `Internationalization` **violation taken from** ***BytecodeDisassembler.java*** **file within Commit** `17bacf` **in project** `Apache httpclient`[31].

`Internationalization` denotes that source code uses non-localized method invocations [108]. Figure 28 presents an example of a fixed `Internationalization` violation, which is a fixed `DM_CONVERT_ CASE` violation that means that a String is being converted to upper or lower case by using the default encoding of the platform [14]. This may result in improper conversions when used with international characters, therefore, this violation is fixed by adding a rule of `Locale.ENGLISH`. For more definitions of categories and descriptions of violation types, please reference paper [108] and `FindBugs` Bug Descriptions [14].

Static analysis techniques are widely used in modern software projects[32]. However, developers and researchers have no clear knowledge on the distributions of violations in the real world, especially for the fixed violations (See Section 3.3). The empirical analysis can provide an overview of this knowledge from three different aspects: occurrences, spread and categories of violations, that can be used to rank violations for developers. The high false positives of `FindBugs` and the common non-severe violations could threaten the validity of the violation ranking. To reduce this threat, we further investigate the distributions of fixed violations in the next section. Fixed violations are resolved by developers, which means that they are detected with correct positions and are treated as issues being addressed, Thus they are likely to be true violations.

APPENDIX C
STATISTICS ON FIXED VIOLATIONS

This section presents the distributions of fixed violations with their recurrences in terms of quantity and in terms of spread. We further compare the distributions of fixed violations and detected ones.

Common types of fixed violations

Figure 29 presents the distributions (in terms of quantity) of fixed violation types sorted by the number of their instances. Fixed violation instances of the top 50 (15%) fixed types (presented by *Order_2* in Table 14) account for about 80% of all fixed violations. Additionally, 122 (about 37%) types are represented in less than 20 instances, 91 (about 27%) types are represented in less than 10 instances, and 20 (6%) types are associated with a single fixed violation instance. These data further suggest that only a few violation types are concerned by developers.

[32]http://findbugs.sourceforge.net/users.html

Figure 30 illustrates the appearance of violation types throughout software projects. There is no correlation between the spread of a fixed violation type and its number of instances: some fixed violation types among the most spread actually occur less than some lesser spread ones. Nevertheless, the top-50 most spread violations account for the majority of fixed violation instances. Additionally, we note that 63 (19%) fixed violation types occur in at least 10% (55/547) projects, which further suggests that only a few violation types are concerned by developers.

Recurrences of types: fixed types VS. all detected ones

Table 14 provides comparison data on the occurrence ratios of fixed violation types against detected violation types. We consider two rankings based on the occurred quantities for all detected violations and for only fixed violations respectively, and select top-50 violation types in each ranking for comparison. If the value of R1/R2 or R2/R1 is close to 1, it means that the violation type has a similar ratio in both fixed instances and detected ones. We refer to this value as *Fluctuation Ratio* (hereafter FR).

In the left side of Table 14, there are 12 violation types marked in green, for which FR values range between 0.80 and 1.20. We consider in such cases that the occurrences are comparable across all violations and fixed violations instances. These 12 violation types have one more type than the types marked in green in the right side because the last type in the left side is not in the top 50 of the right side. On the other hand, FR values of 21 violation types are over 1.5, 10 of them are over 3.0, and 4 of them are even over 10: these numbers suggest that the relevant violation types with high recurrences do not appear to have high priorities of being fixed. Combining FR values and Ratio_2 values, one can infer that developers make a few efforts to fix violation instances for types `SE_BAD_FIELD`, `NM_CLASS_NAMING_CONVENTION`, `SE_TRANSIENT_FIELD_NOT_RESTORED`, `NP_METHOD_PARAMETER_TIGHTENS_ANNOTATION` or `EQ_DOESNT_OVERRIDE_EQUALS`.

In the right side of Table 14, FR values of 23 violation types are over 1.5, 4 of them are over 3.0, and one of them is even over 20: these numbers suggest that the relevant violation types with low recurrences do appear to have high priorities of being fixed. Combining FR values and Ratio_2 values, which can infer that developers ensure that violations of type `NP_NONNULL_RETURN_VIOLATION` are fixed with higher priority than others. Additionally, 13 violation types marked in **bold** in the right side are in the top 50 ranking of fixed violations but not in the top 50 ranking of all detected violations, and vice versa to the types marked in **bold** in the left side of this table.

To sum up, these findings suggest that fixed violation types have different recurrences compared against detected violation types. The order of fixed violation types and the FR values of fixed violation types can provide better criteria to help prioritize violations than the order of all detected

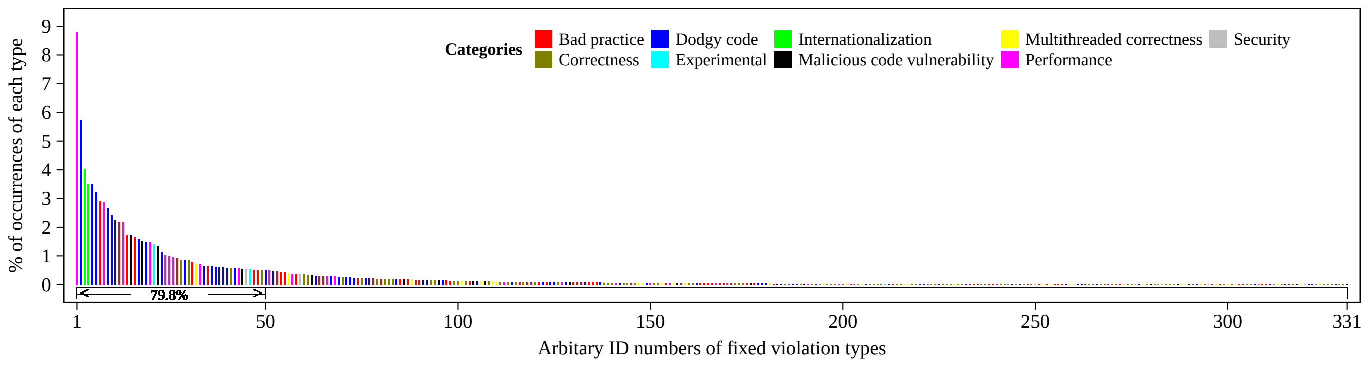

Fig. 29: Quantity distributions of all fixed violation types sorted by their occurrences. The values on x-axis are the id numbers assigned to fixed violation types, which are different from the id numbers in Figure 24. The values of y-axis are the percentages of their occurrences in all fixed violations.

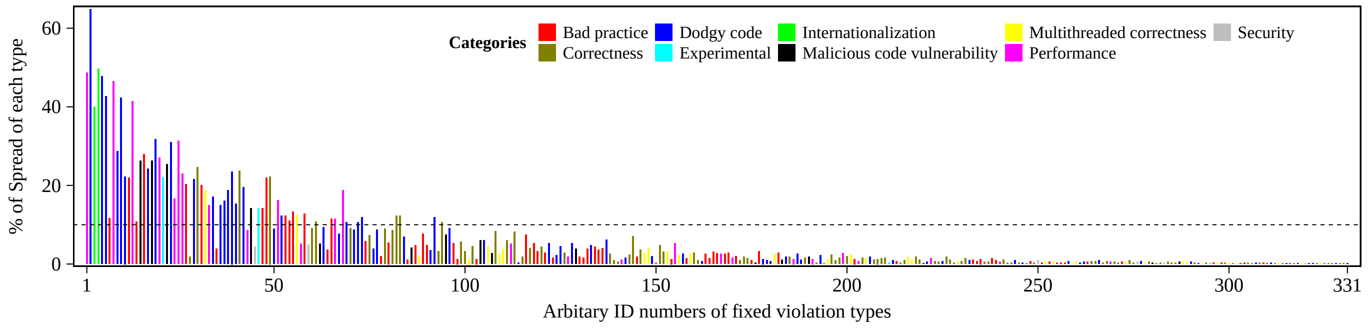

Fig. 30: Spread distributions of all fixed violation types. The x-axis is the same order as in Figure 29. The values of y-axis are percentages of all fixed types in projects.

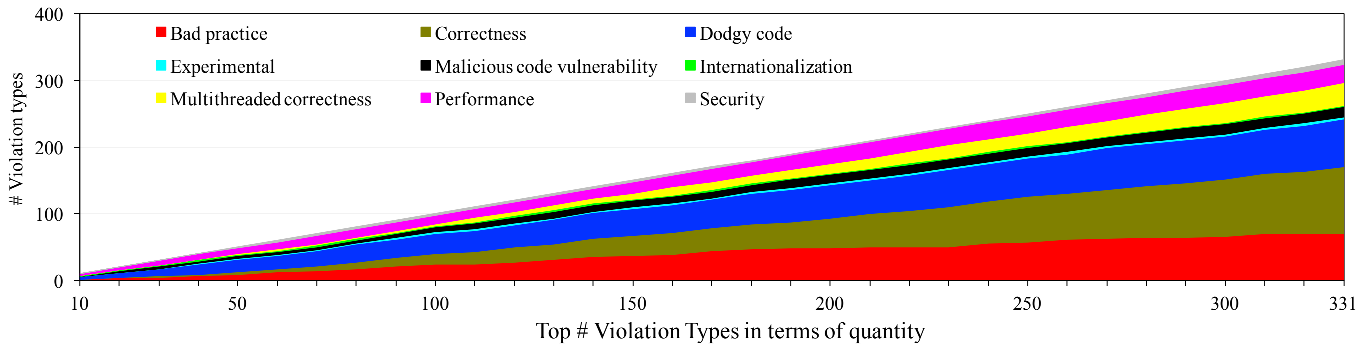

Fig. 31: Category distributions of fixed violation types. The values on x-axis represent the threshold of fixed types ranking used to carry out the category distributions, of which the order is consistent with the order in Figure 29. For example, 50 means that top 50 violation types are used to carry out the category distributions.

violation types, since fixed violations are concerned and resolved by developers.

Category distributions of fixed violations

Table 15 presents the category distributions of fixed violations. `Dodgy code` is the most common fixed category, and the following two secondary common fixed categories are `Performance` and `Bad practice` in terms of occurrences and spread. Fixed `Security` violations are the least common, although they are found in 10% of the projects with fixed violations.

In terms of violation types, `Correctness` regroups the largest number of fixed violation types, but they are not among the top occurring. Figure 31 illustrates the detailed distributions of categories. The number of violation types of `Correctness` increases sharply from the ranking 50 to 331, and there are a few correctness-related violation types in the top 50 types. However, the violation types out of top 50 have much lower number of occurrences compared to top 50 types. Therefore, `Correctness` has a low number of overall fixed occurrences, although it contains the largest number of fixed violation types and is seen in many projects. The top 50 types are mainly occupied by `Dodgy code`, `Performance` and `Bad practice` categories.

Category `Performance` represents the inefficient memory usage or buffer allocation, or usage of non-static class [108]. Figure 32 presents an example of a fixed `Performance` violation. It is a `SBSC_USE_STRINGBUFFER`

TABLE 14: Comparison of distributions of fixed violation types against all detected violation types. *Order_1* refers to the sorting order of violation types by the quantities of all detected violations (cf. the order in Figure 24). *Order_2* refers to the sorting of violation types by the quantities of all fixed violations (cf., order in Figure 29). *Ratio_1* represents the occurred ratio of a given violation type in the all detected violations. *Ratio_2* represents the occurred ratio of a given fixed violation type in all fixed violations. *R1/R2* is used to measure the fluctuation ratio of a violation type from all detected violations to fixed violations, the same as R2/R1.

id	Order_1 (Top 50 all detected types)	Ratio_1(%)	Ratio_2(%)	R1/R2	Order_2 (Top 50 all fixed types)	Ratio_1(%)	Ratio_2(%)	R2/R1
1	SE_NO_SERIALVERSIONID	8.19	2.19	3.73	SIC_INNER_SHOULD_BE_STATIC_ANON	3.15	8.81	2.80
2	RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	5.15	3.24	1.59	DLS_DEAD_LOCAL_STORE	3.64	5.74	1.58
3	BC_UNCONFIRMED_CAST	4.24	2.65	1.60	DM_CONVERT_CASE	1.89	4.04	2.14
4	DLS_DEAD_LOCAL_STORE	3.64	5.74	0.63	DM_DEFAULT_ENCODING	2.49	3.52	1.41
5	EI_EXPOSE_REP	3.20	1.35	2.36	UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	2.29	3.49	1.53
6	SIC_INNER_SHOULD_BE_STATIC_ANON	3.15	8.81	0.36	RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE	5.15	3.24	0.63
7	EI_EXPOSE_REP2	3.14	1.50	2.09	NM_METHOD_NAMING_CONVENTION	2.77	2.89	1.04
8	SE_BAD_FIELD	3.04	0.29	10.36	URF_UNREAD_FIELD	1.26	2.88	2.29
9	NM_METHOD_NAMING_CONVENTION	2.77	2.89	0.96	BC_UNCONFIRMED_CAST	4.24	2.65	0.63
10	DM_DEFAULT_ENCODING	2.49	3.52	0.71	REC_CATCH_EXCEPTION	2.43	2.42	1.00
11	REC_CATCH_EXCEPTION	2.43	2.42	1.00	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE	2.02	2.25	1.12
12	UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR	2.29	3.49	0.66	SE_NO_SERIALVERSIONID	8.19	2.19	0.27
13	PZLA_PREFER_ZERO_LENGTH_ARRAYS	2.19	0.63	3.46	UPM_UNCALLED_PRIVATE_METHOD	1.03	2.16	2.10
14	BC_UNCONFIRMED_CAST_OF_RETURN_VALUE	2.02	2.25	0.90	VA_FORMAT_STRING_USES_NEWLINE	0.25	1.72	6.90
15	RI_REDUNDANT_INTERFACES	1.91	0.62	3.10	MS_SHOULD_BE_FINAL	1.72	1.71	0.99
16	DM_CONVERT_CASE	1.89	4.04	0.47	RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	0.78	1.67	2.15
17	ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	1.82	1.57	1.16	ST_WRITE_TO_STATIC_FROM_INSTANCE_METHOD	1.82	1.57	0.87
18	SF_SWITCH_NO_DEFAULT	1.81	0.86	2.10	EI_EXPOSE_REP2	3.14	1.50	0.48
19	MS_SHOULD_BE_FINAL	1.72	1.71	1.01	URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD	1.26	1.49	1.18
20	NP_LOAD_OF_KNOWN_NULL_VALUE	1.40	1.14	1.22	WMI_WRONG_MAP_ITERATOR	0.72	1.47	2.04
21	URF_UNREAD_PUBLIC_OR_PROTECTED_FIELD	1.26	1.49	0.85	OBL_UNSATISFIED_OBLIGATION	0.52	1.40	2.69
22	URF_UNREAD_FIELD	1.26	2.88	0.44	EI_EXPOSE_REP	3.20	1.35	0.42
23	LI_LAZY_INIT_STATIC	1.25	0.39	3.20	NP_LOAD_OF_KNOWN_NULL_VALUE	1.40	1.14	0.82
24	NM_CLASS_NAMING_CONVENTION	1.23	0.10	12.89	DM_NUMBER_CTOR	1.10	1.04	0.94
25	MS_PKGPROTECT	1.14	0.55	2.07	SIC_INNER_SHOULD_BE_STATIC	0.83	1.00	1.22
26	DM_NUMBER_CTOR	1.10	1.04	1.07	SBSC_USE_STRINGBUFFER_CONCATENATION	0.48	0.97	2.03
27	UPM_UNCALLED_PRIVATE_METHOD	1.03	2.16	0.48	OS_OPEN_STREAM_EXCEPTION_PATH	0.49	0.91	1.87
28	FE_FLOATING_POINT_EQUALITY	1.02	0.48	2.14	NP_NONNULL_RETURN_VIOLATION	0.04	0.87	23.00
29	IS2_INCONSISTENT_SYNC	0.99	0.71	1.38	SF_SWITCH_NO_DEFAULT	1.81	0.86	0.48
30	NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE	0.87	0.50	1.75	UWF_UNWRITTEN_FIELD	0.18	0.84	4.73
31	SE_TRANSIENT_FIELD_NOT_RESTORED	0.85	0.04	22.95	DE_MIGHT_IGNORE	0.71	0.79	1.11
32	NP_METHOD_PARAMETER_TIGHTENS_ANNOTATION	0.83	0.04	23.16	IS2_INCONSISTENT_SYNC	0.99	0.71	0.72
33	SIC_INNER_SHOULD_BE_STATIC	0.83	1.00	0.83	DM_BOXED_PRIMITIVE_FOR_PARSING	0.24	0.71	2.90
34	RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	0.78	1.67	0.47	RV_RETURN_VALUE_IGNORED_NO_SIDE_EFFECT	0.27	0.64	2.42
35	DLS_DEAD_LOCAL_STORE_OF_NULL	0.72	0.19	3.85	ODR_OPEN_DATABASE_RESOURCE	0.22	0.64	2.86
36	WMI_WRONG_MAP_ITERATOR	0.72	1.47	0.49	PZLA_PREFER_ZERO_LENGTH_ARRAYS	2.19	0.63	0.29
37	DE_MIGHT_IGNORE	0.71	0.79	0.90	RI_REDUNDANT_INTERFACES	1.91	0.62	0.32
38	CI_CONFUSED_INHERITANCE	0.62	0.23	2.67	NP_NULL_ON_SOME_PATH_FROM_RETURN_VALUE	0.31	0.61	1.99
39	NM_CONFUSING	0.61	0.43	1.44	UCF_USELESS_CONTROL_FLOW	0.53	0.61	1.15
40	EQ_DOESNT_OVERRIDE_EQUALS	0.56	0.08	6.96	UC_USELESS_CONDITION	0.39	0.59	1.49
41	UCF_USELESS_CONTROL_FLOW	0.53	0.61	0.87	NP_NULL_ON_SOME_PATH	0.29	0.59	2.07
42	OBL_UNSATISFIED_OBLIGATION	0.52	1.40	0.37	UC_USELESS_OBJECT	0.15	0.58	3.96
43	ES_COMPARING_STRINGS_WITH_EQ	0.52	0.51	1.00	DM_FP_NUMBER_CTOR	0.40	0.57	1.43
44	OS_OPEN_STREAM_EXCEPTION_PATH	0.49	0.91	0.53	MS_PKGPROTECT	1.14	0.55	0.48
45	SBSC_USE_STRINGBUFFER_CONCATENATION	0.48	0.97	0.49	SQL_PREPARED_STATEMENT_GENERATED_FROM_NONCONSTANT_STRING	0.27	0.55	2.07
46	SF_SWITCH_FALLTHROUGH	0.44	0.15	2.95	OBL_UNSATISFIED_OBLIGATION_EXCEPTION_EDGE	0.28	0.55	1.96
47	RCN_REDUNDANT_NULLCHECK_OF_NULL_VALUE	0.40	0.24	1.68	ES_COMPARING_STRINGS_WITH_EQ	0.52	0.51	1.00
48	DM_FP_NUMBER_CTOR	0.40	0.57	0.70	OS_OPEN_STREAM	0.36	0.51	1.40
49	UC_USELESS_CONDITION	0.39	0.59	0.67	RCN_REDUNDANT_NULLCHECK_WOULD_HAVE_BEEN_A_NPE	0.24	0.50	2.08
50	HE_EQUALS_USE_HASHCODE	0.39	0.47	0.82	NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE	0.87	0.50	0.57

TABLE 15: Category distributions of fixed violations.

Category	# Violation instances	# Violation types			# Projects
		top-50	top-100	All	
Dodgy code	30,419	18	31	72	505
Performance	19,248	9	13	27	450
Bad practice	15,640	9	24	71	419
Correctness	6,809	5	16	99	384
Internationalization	6,719	2	2	2	347
Malicious code vulnerability	5,505	3	7	16	299
Multithreaded correctness	2,018	1	3	34	208
Experimental	1,748	2	2	3	162
Security	821	1	2	7	47

`_CONCATENATION` violation which denotes that concatenating strings using the + operator in a loop [14]. In each iteration, the `String` is converted to a `StringBuffer` or `StringBuilder`, appended to, and converted back to a `String`, which can lead to a cost quadratic in the number of iterations, as the growing string is recopied in each iteration.

[33]https://github.com/apache/pdfbox

Violation Type: `SBSC_USE_STRINGBUFFER_CONCATENATION.`

Fixing DiffEntry:

```
-     String colorStr = "";
+     StringBuilder sb = new StringBuilder();
      for (float f : color) {
-        if (!colorStr.isEmpty()){
+        if (sb.length() > 0){
-           colorStr += " ";
+           sb.append(' ');
         }
-        colorStr += String.format("%3.2f", f);
+        sb.append(String.format("%3.2f", f));
      }
```

Fig. 32: Example of a fixed `Performance` violation, taken from *Vertex.java* file within Commit `36e820` in Apache `pdfbox`[33] project.

`Internationalization` is also found to be a common fixed category since it has 6,719 fixed violation instances taken from 347 (63.3%) projects and contains only two violation types (i.e., `DM_CONVERT_CASE` and `DM_DEFAULT_ENCODING`)

TABLE 16: Comparison of category distributions of all detected violations and fixed violations.

Category	% Violation Occurrences in		FR values
	All detected (D)	Fixed (F)	(F/D)
Experimental	0.8	1.97	2.46
Correctness	3.21	7.66	2.37
Performance	10.77	21.64	2.01
Internationalization	4.38	7.56	1.73
Security	0.56	0.92	1.70
Dodgy code	39.82	34.21	0.86
Bad practice	26.41	17.59	0.67
Multithreaded correctness	3.56	2.27	0.64
Malicious code vulnerability	10.49	6.19	0.59

that are among top-5 most occurring violation types and among top-10 most widespread throughout projects.

To sum up, these findings suggest that developers may prefer to take more efforts on fixing violations of the four categories, i.e., `Dodgy code`, `Performance`, `Bad practice` and `Internationalization`, than others.

Category distributions: fixed Violations VS. all detected ones

Table 16 shows the comparing results of category distributions of fixed violations against all detected ones. Overall, the ratios of top-5 categories occurrences in fixed violations have increases compared against their ratios in all detected ones. Particularly, the top-3 categories have great increases (more than one fold).

The ratio of `Performance` occurrence in fixed violations has a great increase of 11% compared against its ratio in all ones, which can suggest that developers take many efforts to fix `Performance` violations. The ratio of `Internationalization` occurrence in fixed violations also has a great increase compared against its ratio in all detected ones. And the `Internationalization` contains only two violation types that have high rankings in quantity and spread distributions respectively. So that, it implies that developers take many efforts to fix `Internationalization` violations as well. Even though `Correctness` `Experimental` and `Security` occurrences in fixed violations and all detected ones do not present good rankings, their occurrence ratios in fixed violations have great increases compared against their ratios in all detected ones. The ratios of `Dodgy code` and `Bad practice` occurrences in fixed violations have great decreases compared with their ratios in all detected ones, although they occupy the main proportion in fixed violations.

To sum up, when ranking categories with their FR values, total different priorities of violation categories can be carried out.

Appendix D
Violation code patterns

Example of mined violation code patterns which are consistent with `FindBugs` documentation.

We note that identified common code patterns of many violation types are consistent with the bug descriptions by `FindBugs`. We consider in the following 10 example cases of violation types to investigate the possibility for mining patterns.

`DM_CONVERT_CASE` is converting a string variable or literal to an upper or lower case with the platform's default encoding [14]. It may result in improper conversions when used with international characters. The two patterns are method invocations of `String.toUpperCase()` and `String.toLowerCase()`.

`RCN_REDUNDANT_NULLCHECK_OF_NONNULL_VALUE` represents that the current statement contains a redundant check of a known non-null value against the constant null [14]. Four kinds of common patterns are found in this study, which are shown in Table 5.

`BC_UNCONFIRMED_CAST` denotes that the current cast is unchecked with an `instanceof` test, and not all instances can be cast from their type to the target type that is being cast to [14]. In the three patterns, `T1` is the target type, and `v2` or `exp1` are the value or expression being cast. `BC_UNCONFIRMED_CAST_OF_RETURN_VALUE` has similar patterns, which denotes an unchecked cast of the return value of a method invocation.

`RV_RETURN_VALUE_IGNORED_BAD_PRACTICE` means that the current statement does not check the return value of a method invocation which could indicate an unusual or unexpected function execution [14]. Its patterns consist of a file's creation, a file's deletion and a method invocation with a return value.

`DM_NUMBER_CTOR` is using a number constructor to create a number object, which is inefficient [14]. For example, using `new Integer(...)` is guaranteed to always result in a new `Integer` object whereas `Integer.valueOf(...)` allows caching of values to be done by the compiler, class library, or JVM. Using cached values can avoid object allocation and the code will be faster. Our mined patterns are the five types of number creations with number constructors. `DM_FP_NUMBER_CTRO` has the similar patterns with it.

`DM_BOXED_PRIMITIVE_FOR_PARSING` denotes that a boxed primitive value is created from a `String` value without using an effective static `parseXXX` method [14]. The two common patterns are `Integer.valueOf(str)` and `Long.valueOf(str)`.

`PZLA_PREFER_ZERO_LENGTH_ARRAYS` means that an array-returned method returns a `null` reference which is not an explicit presentation of an empty list of results [14]. It leads to the clients needing a null check for this return value.

`ES_COMPARING_STRINGS_WITH_EQ` denotes the comparison of two strings using `==` or `!=` operator [14]. Unless both strings either were constants in a source file or had been interned using the `String.intern()` method, the same string value might be represented by two different `String` objects.

Appendix E
Reasons for failure to resolve unfixed violations

We have identified 23 violation types where we could not successfully resolve the associated unfixed violations. According to our observation, it might be caused by the following reasons:

Reason 1. It is difficult to match effective fix patterns for specific violations. For example, `DE_MIGHT_IGNORE` violations are fixed by replacing the `Exception` with a specific exception class. Therefore, it is challenging to match an appropriate specific exception class for this kind of violations in terms of syntax without any semantic information or test cases.

Reason 2. It is challenging to identify common fix patterns from the source code changes of some violations without an exact position. For example, `UWF_FIELD_NOT_INITIALIZED_IN_CONSTRUCTOR` means that non-null fields are not initialized in any constructors [14]. Our observation shows that the positions of this kind of violations are located in one constructor. So that, it is impossible to obtain any information about these violations. Even if some information of these violations could be identified, which are the specific information, it is still a challenge to match any effective fix patterns for them.

Reason 3. It is unable to fix `NM_METHOD_NAMING_CONVENTION` violations which do not comply the method naming convention. Even if violated method names can be fixed by matched fix patterns, the changed name may cause compilation errors or API changes that may break client programs.

Reason 4. It is challenging to fix all related violations just by deleting the violated source code. For example, the common fix pattern of `EI_EXPOSE_REP` is deleting the violated source code. When the fix pattern is used to fix related violations, the changed source code may not be correctly compiled.

Reason 5. There might be a lack of effective fix patterns for some violation types. The fix patterns of some violation types are deleting the violated source code. We do not adopt this kind of fix patterns, even though the violation can be fixed or removed by deleting the violated source code, which removes the feature of original source code and many of them failed to pass compile or checkstyle.