Efficient Pattern-based Static Analysis Approach via Regular-Expression Rules

Xiaowen Zhang*
Southern University of Sci. and Tech.
Shenzhen, China
11510746@mail.sustech.edu.cn

Ying Zhou*
Southern University of Sci. and Tech.
Shenzhen, China
11610701@mail.sustech.edu.cn

Shin Hwei Tan†
Southern University of Sci. and Tech.
Shenzhen, China
tansh3@sustech.edu.cn

***Abstract*—Pattern-based static analyzers like SpotBugs use bug patterns (rules) to detect bugs may have several limitations: (1) too slow, (2) do not usually support analysis of partial programs, (3) require parsing code into AST/CFG, and (4) high false positive rate. Each pattern relies on analysis context (e.g., data flow analysis) to improve the accuracy of the analysis. To understand the analysis contexts required by each pattern, we study the design of bug patterns in SpotBugs. Based on our study, we present Codegex, an efficient pattern-based static analysis approach that uses regular expression with several strategies to extract more information from program texts (syntax and type information). It can analyze partial and complete code quickly without parsing code into AST. We evaluate Codegex using two settings. First, we compare the effectiveness and efficiency of Codegex and SpotBugs in analyzing 52 projects. Our results show that Codegex can detect bugs with comparable accuracy as SpotBugs but up to 590X faster, showing the potential of using Codegex as the fast stage of SpotBugs in a two-stage approach for instant feedback. Second, we evaluate Codegex in automated code review by running it on 4256 PRs where it generated 372 review comments and received 116 feedback. Overall, 78.45% of the feedback that we received is positive, indicating the promise of using Codegex for automated code review.**

***Index Terms*—Partial Program Analysis, Static Analysis, Regex**

I. Introduction

Pattern-based static analyzers detect bugs via a set of *bug patterns* (rules for detecting a potential problem in a given program) but there exist several barriers that hinder their wide adoption. First, developers think that static analyzers are too slow to run [1], [2]. Most static analysis tools (e.g., Coverity [3] and Fortify [4]) are designed to run in batch mode, and are not well-integrated into the development environment (IDE) where instant feedback is required [5]. Second, developers prefer static analyzers that support partial analysis, analyzing only recent code changes [2], [6], [7]. Third, most static analyzers rely heavily on compiler technologies. They either (1) require users to manually set up build configurations (cannot be easily deployed to support gazillions of repositories in GitHub [6]) or (2) fail to run due to compilation errors in analyzed project (27.5% of evaluated programs in SpotBugs fail to run [8]). In fact, compiled classes may be unavailable [9]–[11] but only Checkstyle [12] that checks for coding rules does not need compilation. Fourth, the high false positive (FP) rate is the key barrier to using static analyzers [1], [13], [14].

We propose Codegex, a pattern-based static analysis approach based on regular expression (regex). Our key insight is that *many bug patterns checked by pattern-based static analyzers can be naturally represented by regex rules, especially patterns that rely on string matching*. By using regex to match these patterns, our approach can address the aforementioned limitations, including: (1) provides instant feedback by saving compilation overhead, (2) supports analysis of partial programs, and (3) does not require parsing code into AST/CFG that may cause build failure. However, relying solely on regex may lead to high FP rate.

Pattern-based static analyzers usually use one or several types of analysis contexts (e.g., data flow analysis) to reduce FPs. To understand the analysis contexts required by each bug pattern in a pattern-based static analyzer, we conducted a study of SpotBugs. We select SpotBugs as it (1) is one of the most popular static analyzers, and (2) has the largest number of bug patterns than other tools (e.g., Error Prone, PMD, Infer) [8], [15]. Our study revealed that: (1) most bug patterns in SpotBugs do not require analysis contexts beyond the class under analysis,(2) method information (e.g., method name) required in some bug patterns can be extracted directly from the program texts, and (3) data type information is the most important one among all analysis contexts.

Inspired by our study, we design each pattern in Codegex by first using a regex to match the bug within a single statement, and then employing several heuristics to improve the efficiency and accuracy *on-demand* if bug patterns require more analysis contexts. These strategies include (1) *syntax-guided matching* (using keywords to encode class/method signature information), (2) *explicit type-driven matching* (matching implicit data type for patterns requiring type checking), (3) matching at word boundary (optimizing the analysis), (4) broadening analysis scope via "diff" search (searching across all code changes instead of a statement) and online search searching all files in the repository), (5) encoding operator precedence (increasing accuracy of analyzing arithmetic and bitwise operations), and (6) enforcing *anti-patterns* (rules for filtering FPs).

Codegex can benefit developers in two settings: (1) Codegex provides instant feedback in the IDE via 87 patterns. Motivated by a prior study where developers expressed the need for a

*Joint first authors
†Corresponding author

two-stage approach in a program analyzer [2], Codegex is used on top of SpotBugs with a two-stage approach where the first stage runs the detectors for the 87 patterns in Codegex to provide real-time feedback, and the second stage runs the remaining detectors for more sophisticated analysis during nightly builds. (2) Codegex analyzes the code snippets within a pull request (PR) during code review. Prior studies show that static analysis approaches could assist developers in automated code review [16], [17]. A prior attempt was to integrate FindBugs (a deprecated predecessor of SpotBugs) as part of a review bot [18]. As it requires bytecode for analysis, the Review Bot workaround this by (1) maintaining a local copy of the project source code, (2) synchronizing the local copy to the changelist determined by a trial-and-error approach (trying each candidate changelist until the build succeeds), (3) copying merged files, and (4) building the project. The workaround may still incur build failures, causing delays in running SpotBugs for automated code review.

Overall, our contributions can be summarized as follows:

- We perform the first study of the analysis contexts of 438 bug patterns in SpotBugs by reading their documentation and implementation. It helps us identify the set of bug patterns that can be run quickly in the two-stage approach.
- We propose Codegex, a novel static analysis approach that uses regex-based rules and several strategies that augments rules by analyzing contexts. Codegex can perform quick yet accurate analysis of partial code snippets without parsing into AST/CFG.
- We compare the effectiveness of Codegex and SpotBugs on 52 open-source projects. Our results show that Codegex can analyze real-world projects quickly with comparable accuracy as SpotBugs. Codegex runs up to approximately 24K faster than SpotBugs when considering the initial compilation time. Moreover, we manually analyzed the FP and false negative (FN) cases reported by SpotBugs. Our study reveals several limitations of SpotBugs, and provides insights on potential improvements to its bug detection capabilities. Overall, we have reported 16 bugs to SpotBugs where ten are confirmed and eight are fixed.
- We evaluate the effectiveness of Codegex in automated code review by running it against 4256 PRs from 2769 different projects. As Codegex automatically analyzes PRs and leaves code review comments, we assess whether it follows the bot ethics [19]. To our knowledge, this is the largest evaluation reported for automated code review on unresolved PRs. In the end, we received 91 positive feedback from the developers. The source code and our dataset are available at the anonymous link at https://codegex-analysis.github.io.

II. Motivating example

As Codegex only requires limited contexts (in a PR), one may think that it is essentially trading accuracy for speed. In this section, we use a simplified example from a PR [20] to show how Codegex can be faster and yet more accurate in detecting certain patterns. Consider SA_FIELD_SELF_COMPUTATION and SA_LOCAL_SELF_COMPUTATION patterns that check for nonsensical self computation (e.g., x & x) in global fields and local variables, respectively. Since Codegex relies on program text information in partial snippets, it cannot distinguish between a global field and a local variable without analyzing its scope. However, as self computation is considered problematic in any scope (local or global), Codegex can still detect the self computation. Listing 1 shows a self computation reported by Codegex in the expression `endDate.getTime() - endDate.getTime()`. Codegex detects the self computation by (1) searching for keywords representing arithmetic operators ('|', '∧', '&', '−'), and (2) if the keyword is found, it uses the following regex to detect the self computation:

named capturing group — subroutine of aux1

```
(\w(?:[\w.]|(?P<aux1>...))*)\s*([|^&-])\s*(\w(?:[\w.]|(?&aux1))*)
```

operand1 — operator — operand2

The regex above checks if `operand1` matches `operand2`. In contrast, SpotBugs fails to detect the self computation in Listing 1 because (1) it performs bytecode analysis where the check expression needs to be parsed and finds two method invocations from the stack, and (2) it needs to match the program text with the object `endDate` and the method `getTime()` (if an expression includes a long method call chain like `a.b.c()`, it needs to iteratively traverse the call chains). Based on the example (`x-x`) provided in the bug description for SA_LOCAL_SELF_COMPUTATION, we encode the '-' operator as one of the operators for self computation, thinking that SpotBugs should be able to detect the self computation `temp-temp` in Listing 2 where the expression `endDate.getTime()` is stored in `temp`. But SpotBugs still fails to detect the self computation as (1) it fails to detect double type variables (the opcodes for integer subtraction and floating point subtraction are different, adding this support requires matching opcodes for all supported data types); and (2) a bug exits in its current implementation of self computation for local variables. Although the expression `x-x` is listed as an example in its description, the current implementation only matches expressions containing the xor operator "∧", and ignores all other operators. We reported and provided corresponding PRs to the developers of SpotBugs to fix both limitations [21], [22], and the developers have been accepted them. The example shows two problems in SpotBugs: (1) *incomplete modeling of sibling types* (e.g., double and integer), and (2) *failure to handle complex expressions involving method calls*. Although it may seem trivial to extend SpotBugs to support more opcodes, it requires the designer of a bug detector to think exhaustively about the possible scenarios (e.g., different data types) in which the detector will be invoked. Our example also shows that the scope of the variables (local versus field) is irrelevant for checking for self computation. The key to check for self computations lies in the string matching of the form `x op x`. Hence, we argue that *bug patterns that involve string matching (e.g. specified method names or operators) can be more easily checked using*

```
1  double exampleSelfComputation1(Date endDate){
2   //Codegex reported a self computation
3   double temp = endDate.getTime() - endDate.getTime();
4   return temp; }
```

Listing 1: Self computation detected by Codegex

```
1  double exampleSelfComputation2(Date endDate){
2   double temp = endDate.getTime();
3   // SpotBugs should report this self computation
4   double second = temp - temp;
5   return second; }
```

Listing 2: Self computation that should be detected by SpotBugs

regex rules.

III. Motivating study

Static analysis tools like SpotBugs rely on bug patterns, and each pattern depends on different types of analysis contexts to detect a bug. However, according to SpotBugs's official documentation [23], most of its analysis is local, which means that many of the analysis contexts are not required to detect a bug pattern. To investigate the required analysis contexts, we conducted a study of 451 bug patterns supported by SpotBugs. Our study aims to answer the research questions below:

RQ1: What is the scope of analysis needed to detect a bug pattern in SpotBugs?

RQ2: What program analysis techniques are important for detecting a bug pattern in SpotBugs?

RQ1 aims to identify the scope of analysis required for the bug patterns in SpotBugs. In RQ2, we studied the program analysis techniques needed for each pattern in SpotBugs. For each question, two authors of the paper categorized the results independently and met to resolve any disagreement.

For each bug pattern b, we first obtained a high-level understanding of its design by reading b's documentation (also known as *bug description* in SpotBugs). Each bug description contains (1) the rationale behind the bug patterns (explaining why certain program behavior is problematic), (2) the condition that will trigger the bug, and (3) examples of the bug detected by a bug pattern. Then, if we failed to answer the two research questions based on the bug descriptions, we refer to the implementation of each pattern in SpotBugs. In total, there are 451 patterns listed on SpotBugs' bug description page. We excluded 12 of them as they are internal patterns used only by SpotBugs in experiments, and one of them as the pattern has been deleted from the implementation. These results in 438 patterns in our study. To study the analysis contexts required by each of these 438 patterns, we consider four scopes of analysis: (1) inter-class, (2) class, (3) method, and (4) statement. If a bug pattern requires multiple analysis scopes, we select the highest scope of analysis (e.g., if a pattern needs class level and method level information, we consider that it needs class level information). For RQ2, we study program analysis techniques (type, annotation, java version, data flow, control flow, call graph, inheritance graph) that are used in each bug pattern.

Table I shows the results of our study where the "Context" column denotes the context information used, the "Description" column explains each context, and the "Pattern (%)" shows the percentage of patterns that require specific contexts. The "Implemented (%)" shows the percentage of patterns implemented in Codegex among all patterns using a particular analysis context. Table I shows that most patterns in SpotBugs

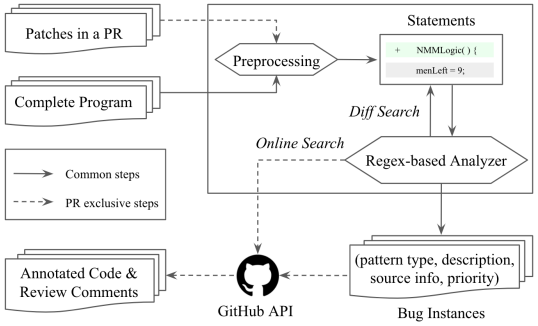

Fig. 1: Workflow of Codegex

require analyzing only method level (41.32%) or statement level (31.28%) information for their analyses, which indicates that *most patterns do not require information beyond the class under analysis*. Moreover, we observe that some method information (e.g., method name) can be obtained by analyzing the program texts. In fact, eight patterns in Codegex use the method signature information for their analysis. Among all program analysis techniques required by each pattern, our study shows that *data type information is the most important one* (44.29%). Meanwhile, the last rows of Table I show that most bug patterns do not require information from various graph representations (control flow, data flow, call graph, inheritance graph). Specifically, most graph-based analysis techniques are required in less than 10% of the bug patterns in SpotBugs (except for inheritance graph).

IV. Methodology

Figure 1 shows the workflow of Codegex. Given a PR or a complete program, Codegex first splits the code into program statements S. Then, it uses our regex-based analyzer with several heuristics to check if S matches any pattern. As the complete program has been downloaded, Codegex skips the online search, and relies on diff search for broadening its analysis scope. The analyzer produces information about the pattern type, bug description, source information (file name, line number), and the priority for a warning, which can be directly used as output for complete program. For PRs, our PR comment generator automatically produces review comments with the annotated code.

A. Preprocessing

PR: A PR in the unified diff format contains the contexts, additions, and deletions. Our analyzer checks for violations in contexts and additions but ignores the deletions (as they no longer exist in the new version).

Complete program: A complete program is a code change that adds all files within the project repository.

TABLE I: Analysis contexts used in the bug patterns in SpotBugs and those in the included patterns in Codegex

Context		Description	Patterns (%)[2]		Implemented (%)[2]	
Inter-class		Check in multiple classes whether a field/method exists or look at their content	**9.36**		**0**	
Class	Import	Check if the fully-qualified names (the import statement) contain specific substring	0.91	**18.04**	0	**16.46**
	Signature	Check the kind (class/interface/enum), modifiers and name of the inspected class	3.2		35.71	
	Field	Check the signature of fields (modifiers, type and name) in the inspected class	11.19		16.33	
	Body	Use information from (1) at least two methods, or (2) one method and other class-level info	10.96		2.08	
Method	Signature	Check modifiers, return type, method name, parameters of the inspected method	13.7	**41.32**	13.33	**18.78**
	Local variable	Check the signature or usage of local variables	4.57		40	
	Field	Check the usage of fields in the scope of the inspected method	4.79		14.29	
	Body	Use information from (1) at least two statements, or (2) one statement and other method-level info	34.02		10.74	
Statement		Use information from only one statement or expression	**31.28**		**30.66**	

Analysis Techniques	Type	Check the data type information of constants, variables or fields	44.29
	Annotation	Check annotations of class, fields, methods or parameters	5.02
	Java version	Check the JDK version	1.14
	Data flow	Calculate the set of possible values at each program point	9.36
	Control flow	Check the control flow of the program	9.36
	Call graph	Check the relationship between method calls	3.2
	Inheritance graph	Check the inheritance relationship between classes	17.81

[1] The numbers (in bold text) for inter-class, class, method and statement levels do not overlap and sum up to 100 (e.g., 9.36+18.04+41.32+31.28=100%). The numbers for the sub-contexts for class level, method level and analysis techniques are intersecting (e.g., one pattern can use method information from signature and local variable).
[2] We compute these values using the total number of patterns (438) as the denominator.

Given a PR or a complete program with code changes C, Codegex parses the program text in C, and splits the texts into program statements. Specifically, it separates statements using terminators for Java programs (i.e., semicolon, '{', and '}'). Preprocessing allows each pattern to be matched statement-by-statement, giving the exact position of the statement.

B. Regex-based Analyzer

Given statements S extracted from our preprocessing, our analyzer detects violation for bug patterns in S. The key technical challenge is to *represent the selected patterns in SpotBugs using regex rules without compromising accuracy*. We use several heuristics described below to solve this problem:

Syntax-guided matching. Table I shows that many SpotBugs patterns use class signature information(3.20%) or method signature (13.70%) for detection. To encode such information into our regex rules, we use keywords representing class/method/variable/field names, modifiers (e.g., "static"), Java keywords (e.g., "if")), and operators ("&&") that are within the Java syntax for the bug detection. To support syntax-guided matching, we use the layered analysis approach [24] by checking a pattern using two phases: (1) keyword matching, (2) pattern-based matching using regex. Keyword matching to filter statements that do not match any patterns is a faster than regex rules. Specifically, it checks whether a statement contains a keyword that represents the condition for a bug pattern or a group of bug patterns. For example, the SE_NONSTATIC_SERIALVERSIONID pattern checks whether the field name `serialVersionUID` of a serializable class is declared as `static`. Statements without the keyword "serialVersionUID" are skipped in keyword matching.

Explicit type driven matching. Our study in Section III shows that data type information is an important analysis context used in many bug patterns. Although Codegex essentially treats code changes as plain texts for pattern matching, we incorporate explicit type information into bug patterns by using data types as keywords for its analysis. For example, when detecting the pattern RV_01_TO_INT that gives a warning when a random value from 0 to 1 is being coerced to integer value, Codegex uses the regex `\(\s*int\s*\)\s*(\w+)\.(?:random|nextDouble|nextFloat)\(\s*\)` for its detection where the `\(\s*int\s*\)` part detects the type coercion. By including the data type information, Codegex can report this pattern with high confidence by setting the bug pattern to high priority (the same priority used in SpotBugs).

Optimization by matching at word boundary. One of the commonly used heuristics to optimize regex performance is to do "whole words" matching by using word boundaries [25]. In the regex syntax, '\b' matches a *word boundary* (edge between sequences of alphanumeric characters or the underscore character, and any other character). For example, the regex `\bif\b` matches the standalone string " if " but does not match "ifa" because there is no word boundary to the right of "if". As program texts are usually whole word strings, we restrict each bug pattern to search for whole words so that it will skip over non-matching input quickly.

Broaden analysis scope via diff search and online search. Our study in Section III shows that some bug patterns in SpotBugs require more analysis contexts to detect certain patterns. Codegex includes two heuristics to enhance the analysis contexts: (1) diff search, and (2) online search. When these heuristics identifies the relevant analysis contexts, Codegex will adjust the priority for a given bug pattern as it gains higher confidence with more contexts. For most implemented bug patterns, Codegex matches a regex against a single program statement st. When the diff search heuristic is activated in a bug pattern, Codegex will use additional contexts around st by searching through all code changes in the input PR. For example, consider the SpotBugs pattern UI_INHERITANCE_UNSAFE_GETRESOURCE that warns about the usage of `this.getClass().getResource()` being unsafe if this class is extended by a class in another package. Detecting this pattern requires checking whether (1) a statement contains the `getClass().getResource()` method invocations (regex can be used to match this), and (2)

if the class is extended (SpotBugs will increase the priority of the warning if this condition is met). To check for the (2) condition, Codegex searches for the "extends ClassA" keywords (ClassA is the filename of `this` instance) within the code changes ("diff") in the given PR using the *diff search* heuristic (note that this will not work for inner class with different filenames). If the diff search fails, Codegex will deploy online search to check for the (2) condition. As Codegex only pre-downloads "diff" in the PR, other classes that extend ClassA are not available for analysis (i.e., they may be in different files or different folders than the modified files in the PR). Hence, we implement *online search* using the GitHub Search API[1] to perform code search of the entire repository of the given PR for the "extends ClassA" keywords. If the query is found within the repository of the PR, Codegex will increase the priority of the bug pattern because the (2) condition is satisfied. Currently, Codegex uses online search in only one pattern because (1) it is expensive as it relies on the speed of the GitHub Search API, and (2) it requires defining a search query with exact matching (e.g., if we change the query to "extend Class", the search may return irrelevant results).

Encode operator precedence. We encode the Java operator precedence (used for determining the order in which operators are evaluated) in our analyzer to increase the accuracy of analyzing arithmetic and bitwise operations. For example, when detecting the SA_LOCAL_SELF_COMPUTATION pattern that checks for nonsensical self computation in `return i|i&j;`, if we use a regex to extract the bitwise operation, it will match the first expression `i|i` instead of `i|(i&j)` because the precedence of `&` is higher than that of `|`. Encoding operator precedence will reduce Codegex's FP rate.

Most patterns in SpotBugs have *anti-patterns* (i.e., rules () that disallow matching certain elements). To reduce FPs, we encode anti-patterns using two heuristics:

Encode anti-patterns via keyword filtering. To understand and reuse the design of each pattern, we refer to: (1) bug description, (2) source code, and (3) test cases in SpotBugs. We extract anti-patterns from these resources to improve the accuracy of our analysis. For example, the pattern NM_CLASS_NAMING_CONVENTION checks for upper camel cases of a Java class. To prevent FPs when analyzing special classes, SpotBugs added a filter for class names with the underscore character. We reuse this filter to skip the checking for class names with underscore characters.

Encode anti-patterns via negative lookahead. As Codegex analyzes incomplete programs that may contain only the declaration site or the call site of methods, it uses *negative lookahead* (a regex construct `q(?!u)` used to match *q not* followed by the regex u) for filtering negative/corner cases. For example, to detect the NM_METHOD_NAMING_CONVENTION pattern that checks whether a Java method is in the lower camel cases format, we include the regex `(?!new)` to avoid matching constructors (e.g., new Object()) that can have method names starting with a capital letter.

[1]https://docs.github.com/en/rest/reference/search

TABLE II: Statistics of implemented bug patterns.

Category	# implemented patterns	Total
CORRECTNESS	37	145
BAD_PRACTICE	22	91
PERFORMANCE	14	37
STYLE	8	86
MT_CORRECTNESS	5	46
MALICIOUS_CODE	1	17
Others	0	16
Total	87	438

C. PR Comment Generator

For each code snippet in a PR in which our analyzer produces a warning, our PR comment generator will give a review comment with the annotated code. We reuse SpotBugs's bug description for the comments. In SpotBugs, code that violates a bug pattern pat has (1) a bug category cat (e.g., STYLE), (2) a short description sd, and (3) a long description ld. Codegex produces review comments using the template below:

I detect that this code is problematic. According to the cat, sd (pat). ld

Figure 3 shows an example of Codegex's generated comment and the annotated code for the NM_METHOD_NAMING_CONVENTION pattern that belongs to the BAD_PRACTICE category.

Implementation. Codegex uses the Python built-in regex library in which the regex pattern language has been studied [26], and its extension that offers extra functionalities (e.g., named capturing group). Table II shows the statistics of the implemented bug patterns across different categories. The "# implemented patterns" column shows the number of patterns implemented in Codegex, and the "Total" column denotes the total number of patterns in SpotBugs. We only implement 87 patterns because 212 patterns require supporting multiline regex, and 139 patterns cannot be detected using regex. As shown in Table II, most patterns Codegex currently supports belong to the *CORRECTNESS* and BAD_PRACTICE categories. We prioritize these categories because prior studies of FindBugs have shown their importance (i.e., most development efforts focus on these categories [27], and they have a shorter lifetime implying that they are more serious [28]).

V. Evaluation

We evaluate two settings in which Codegex may be useful: (1) giving instant feedback for real-world projects, and (2) providing review comments for PRs. Codegex uses several heuristics to improve the effectiveness of analysis (Section IV-B). As all heuristics (except for online search) *are tightly coupled with the design of each bug pattern*, we did not separately evaluate each heuristic.

All experiments were conducted on a machine with Intel (R) Core (TM) i7-8700 CPU @3.2 GHz and 32 GB RAM.

Comparison with SpotBugs. While there are many static analyzers [12], [29]–[32], we only evaluate against SpotBugs because patterns in Codegex are derived from SpotBugs, and patterns determined the types of detected bugs (*fair comparison with other analyzers is infeasible because each tool detects*

TABLE III: Performance of Codegex versus SpotBugs (in seconds)

Project	KSLOC	SpotBugs			Codegex	Speedup		
		IC	SC	A(S)	A(C)	S1	S2	S3
community	3.72	160.80	6.55	5.13	0.58	283.91	19.98	8.78
Angular2AndJavaEE	1.42	239.20	92.80	5.44	0.25	963.40	386.87	21.42
biojava	117.78	59.22	13.70	42.44	19.30	5.27	2.91	2.20
nacos-spring	1.80	584.01	12.84	15.36	0.31	1925.08	90.57	49.33
spring-boot	0.48	59.09	3.59	3.60	0.07	904.59	103.78	51.92
spring-boot-java	5.30	70.00	9.80	5.00	0.92	81.80	16.14	5.46
quickfixj	1279.49	442.60	180.20	116.80	3.25	171.90	91.26	35.89
spring-comparing	0.91	178.40	7.22	4.36	0.14	1304.40	82.65	31.09
tij4-maven	17.89	54.19	0.24	1.05	<0.01	5524.60	129.80	105.40
fabric8-maven-plugin	48.01	2654.20	17.39	43.60	3.79	712.10	16.10	11.51
java-microservice	2.59	883.80	8.12	11.04	0.04	**24213.24**	518.55	298.78
spring-cloud-release	0.49	68.37	59.86	1.55	<0.01	6991.80	**6140.40**	154.60
java-uuid-generator	3.07	7.13	2.17	1.80	0.26	34.02	15.14	6.86
cloud-opensource	10.51	721.80	322.60	11.17	0.90	817.39	372.22	12.46
flyer-maker	0.76	13.37	1.83	3.51	0.15	114.54	36.25	23.83
gchisto	6.57	8.60	2.70	5.61	1.23	11.54	6.75	4.56
travels-java-api	2.95	204.60	5.16	4.49	0.35	597.25	27.56	12.82
spring-zeebe	1.58	951.80	8.38	10.51	0.22	4326.06	84.93	47.26
visualee	3.76	37.26	3.48	4.04	0.32	130.81	23.82	12.80
javaee7-essentials	0.01	1.61	1.29	1.11	<0.01	272.40	239.60	111.00
webcam-capture	16.02	183.20	7.24	32.07	1.93	111.40	20.34	16.60
cloud-espm-v2	4.67	4.19	3.70	5.94	0.45	22.40	21.30	13.13
reactive-ms-example	1.49	570.00	3.55	3.58	0.10	5986.65	74.42	37.32
osgi.enroute	0.99	3351.60	163.20	15.84	0.24	13815.62	734.53	64.97
kafka-streams	99.40	4203.80	23.62	9.35	16.94	248.64	1.95	0.55
code-assert	7.61	970.20	10.46	11.03	1.10	889.68	19.48	10.00
opencc4j	0.87	80.80	2.62	3.36	0.10	815.89	57.97	32.61
hprose-java	15.85	12.37	3.72	6.06	2.85	6.47	3.43	2.13
SpringBootUnity	5.86	2682.20	17.76	31.70	1.19	2287.90	41.70	26.72
triava	5.92	7.26	2.09	4.52	0.98	12.07	6.78	4.63
jol	7.20	14.01	3.13	10.73	0.86	28.73	16.10	12.46
javaee8-essentials	0.02	1.38	1.10	1.14	<0.01	251.80	224.00	113.80
cargotracker	5.97	626.00	6.40	4.54	0.73	864.99	15.01	6.23
hope-cloud	0.14	168.80	5.94	8.38	0.01	11935.33	964.24	564.37
java-speech-api	1.37	8.60	1.68	3.74	0.28	43.49	19.09	13.18
jmh	249.88	73.42	16.23	29.73	3.71	27.80	12.39	8.01
javaee-javascript	0.35	27.69	1.90	3.29	0.06	547.18	91.67	58.14
paho.mqtt.java	28.44	31.92	11.35	13.59	3.43	13.26	7.26	3.96
bitfinex-v2	6.41	24.89	2.78	4.51	0.91	32.18	7.98	4.93
aem-component	1.65	88.96	6.62	4.20	0.33	281.43	32.70	12.69
superword	9.08	115.00	23.13	6.14	1.83	66.14	15.98	3.35
reddit-bot	1.50	15.27	3.45	3.84	0.20	95.71	36.50	19.24
asmsupport	26.71	28.78	6.28	16.78	4.44	10.26	5.19	3.78
Benchmark	146.26	110.00	15.03	21.79	28.79	4.58	1.28	0.76
wro4j	33.41	356.20	44.72	11.91	3.34	110.26	16.96	3.57
spring-mvc	2.00	83.20	4.80	4.25	0.34	261.02	27.03	12.70
spring-context	3.06	14.42	1.94	3.86	0.38	48.06	15.25	10.15
iot-dc3	14.67	295.20	28.01	45.70	2.97	114.65	24.79	15.37
nacos-spring-project	7.40	104.20	6.94	8.31	0.90	124.99	16.94	9.23
spring-boot-graalvm	0.05	59.06	3.21	3.37	0.01	10927.37	1151.37	**589.86**
cms-admin-end	8.23	74.80	5.19	5.64	1.47	54.91	7.39	3.85
spring4.x-project	0.58	348.80	2.08	22.42	0.11	3495.71	230.69	211.12
Average	42.73	425.70	23.07	12.67	2.18	1979.28	237.06	55.72

different types of bugs). Our evaluation aims to address the questions below:

RQ3: Compared to SpotBugs, what is the quality of the generated warnings by Codegex?
RQ4: How responsive is Codegex compared to SpotBugs?

Selection of projects. We evaluate Codegex and SpotBugs on 52 open-source Java projects on GitHub. Projects are selected by building a crawler to get the top 100 Java projects that (1) have the greatest number of stars, and (2) use Maven for compilation (the SpotBugs Maven plugin is baseline). Although Codegex does not require compilation, SpotBugs can only be run on compiled code so we excluded 48 uncompilable projects. We manually classify the root causes of build errors from 48 projects' build logs using the definition [33]. We conclude that these errors occur due to: (1) 19 compilation errors caused by incompatible Java versions or syntax errors in the code, (2) 13 resolution errors due to missing dependencies or dependency version issues, (3) 12 errors are other causes (e.g. network problems), and (4) four build file parsing failures. The high percentage of build errors in popular Java projects is inline with the findings of prior work that broken snapshots occur in most projects [33]. It motivates the need for a tool that does not require compilation. We did not further crawl more projects due to limited resources and high manual inspection costs. In total, we evaluate on 52 Java projects. To the best of our knowledge, *the number of evaluated projects is the largest reported so far in all prior evaluations of static analyzers* [13], [34]–[36]. Table III presents the performance comparison with SpotBugs. The "Project" column denotes the abbreviated name of each project, and the "KSLOC" column means the 1000 (K) Source Lines of Code (SLOC). Overall, the evaluated projects are diverse in terms of size (0.01–1279.49 KSLOC).

When running SpotBugs, we use default configurations except for two differences: (1) using the `debug` option in SpotBugs to output a list of analyzed files and give the list to Codegex to ensure that both tools analyze the same files (note that we disable the debug option when computing the analysis time to avoid adding overhead to SpotBugs), (2) running both tools only on the 87 implemented patterns, filtering out the unimplemented patterns in SpotBugs. In our experiments, we use version v4.1.4 of SpotBugs and SpotBugs Maven Plugin (v4.2.3). To ensure fair comparison with SpotBugs, the entire repository is downloaded and the same set of files are given to both tools for analysis.

Quality of generated warnings. We measure the quality of the warnings generated by each tool. Given the set C of Codegex's generated warnings and the set S of SpotBugs' generated warnings, we use $S \bigcup C$ as our ground truth because (1) labeled dataset for all the evaluated projects is unavailable and manually labeling each statement as buggy or not is time-consuming, and (2) as FNs are absent warnings, we can only rely on the extra warnings (S' or C') to determine the FNs of each tool. We compute relative true positive (TP^R) which is true warning in $S \bigcup C$, relative false positive (FP^R) which is false warning in $S \bigcup C$, relative false negative (FN^R) refers to unreported true warning in $S \bigcup C$.

Relative Accuracy=$\frac{TP^R+TN^R}{TP^R+FP^R+FN^R+TN^R}$

Relative Recall=$\frac{TP^R}{TP^R+FN^R}$ J(S, C)=$\frac{|S \cap C|}{|S \cup C|}$=$\frac{760}{883} \approx 0.86$

Relative accuracy and recall are used to compare the quality of analysis results, whereas Jaccard index (J(S, C)) is used to measure the similarity between two sets of data (between S and C). The precision is 100% for each tool as all results will be in our ground truth dataset. The high Jaccard index between the warnings generated by SpotBugs and those generated by Codegex (0.86) indicates that *the analysis results of Codegex are comparable to that of SpotBugs*. Since Jaccard index shows high similarity between S and C, we divided the set of warnings given by both tools into:

(O) Overlaps: The set of warnings generated by both tools that match the *same bug instance* (the same statement within the same class for the same project). When both tools give the same warnings, we assume that these warnings share the same quality and label them as TP^R. We manually analyzed them to verify that they are referring to the same bug instance.
(S') Unique in SpotBugs: The set of warnings generated exclusively by SpotBugs but not produced by Codegex.
(C') Unique in Codegex: The set of warnings generated exclusively by Codegex but not produced by SpotBugs.

RQ3: Results for effectiveness. Table IV presents the results for the effectiveness of SpotBugs and Codegex for the patterns

TABLE IV: Effectiveness of Codegex versus SpotBugs

Pattern	TP^R		FN^R		O	Accuracy		Recall	
	S	C	S	C		S	C	S	C
ES_COMPARING_STRINGS_WITH_EQ	16	5	0	11	5	100.0	31.25	100.0	31.25
SA_SELF_COMPUTATION	0	1	1	0	0	0.0	**100.0**	0.0	**100.0**
EQ_COMPARING_CLASS_NAMES	0	1	1	0	0	0.0	**100.0**	0.0	**100.0**
DMI_RANDOM_USED_ONLY_ONCE	176	215	39	0	176	81.86	**100.0**	81.86	**100.0**
SA_SELF_COMPARISON	0	1	1	0	0	0.0	**100.0**	0.0	**100.0**
VA_FORMAT_STRING_USES_NEWLINE	60	66	6	0	60	90.91	**100.0**	90.91	**100.0**
NM_CLASS_NAMING_CONVENTION	6	5	0	1	5	100.0	83.33	100.0	83.33
DM_STRING_CTOR	5	1	0	4	1	100.0	20.0	100.0	20.0
UI_INHERITANCE_UNSAFE_GETRESOURCE	8	9	1	0	8	88.89	**100.0**	88.89	**100.0**
DMI_USELESS_SUBSTRING	0	1	1	0	0	0.0	**100.0**	0.0	**100.0**
DM_BOXED_PRIMITIVE_FOR_COMPARE	2	1	0	1	1	100.0	50.0	100.0	50.0
DM_BOXED_PRIMITIVE_FOR_PARSING	13	5	0	8	5	100.0	38.46	100.0	38.46
IIO_INEFFICIENT_LAST_INDEX_OF	26	25	0	1	25	100.0	96.15	100.0	96.15
DMI_HARDCODED_ABSOLUTE_FILENAME	46	16	1	31	15	97.87	34.04	97.87	34.04
IIO_INEFFICIENT_INDEX_OF	363	354	3	12	351	99.18	96.72	99.18	96.72
Others (20 unique patterns)	108	108	0	0	108	100.0	100.0	100.0	100.0
Total	829	814	54	69	760	93.88	92.19	93.88	92.19

in S' or C'. The "O" column shows the number of overlapping warnings (O) for each bug pattern where the "Others" row denotes 20 bug patterns where both tools produce the same sets of warnings (i.e., achieve same accuracy and precision). The "Accuracy" and "Recall" columns show the relative accuracy and relative recall, respectively. We observe that *the values for the relative accuracy and recall for a tool are the same in each row*. We explain this scenario by including "TP^R" columns to show the relative true positives and "FN^R" columns to show the relative false negatives (we did not show the values for relative TN and FP as TP^R=FP^R=0 for all patterns). The relative FP^R=0 for all patterns because (1) our manual analysis shows that all evaluated tools only generate true warnings (no theoretical FP), and (2) these warnings may be marked as effective FP [37] by developers but we only evaluate theoretical FP in **Q1**. Overall, Codegex achieved comparable results with SpotBugs in terms of overall accuracy and recall. As highlighted in Table IV, Codegex outperforms SpotBugs in accuracy and recall for seven patterns. We observe that these seven patterns mostly rely on string matching (e.g., matching the program text of the operands for the self computations patterns, and matching class/method names for patterns like DMI_RANDOM_USED_ONLY_ONCE). This observation is inline with our hypothesis that *bug patterns relying on string matching can be more easily matched via regex rules*. Refer to Section VII for the limitations of each tool.

Answer to RQ3: Codegex achieves comparable accuracy as SpotBugs.

Responsiveness. We compute the metrics below:
(IC) Initial compilation time: Time taken to build a project
(SC) Subsequent compilation time: Time taken to build a project with all dependencies being pre-downloaded.
(A(t)) Analysis time: Time taken for a tool t to produce analysis reports. We use A(S) to denote SpotBugs' analysis time, and A(C) for Codegex's analysis time.

S1 = $\frac{IC+A(S)}{A(C)}$ **S2** = $\frac{SC+A(S)}{A(C)}$ **S3** = $\frac{A(S)}{A(C)}$

We calculate IC and SC only for SpotBugs as it requires compilation. We include both IC and SC since one may argue that IC is unimportant as downloading dependencies is only a one-time effort. To account for performance differences across multiple runs, we *rerun each time calculation for five runs* and reported the average time in Table III.

RQ4: Results for response time. The last seven columns in Table III show the performance comparison between Codegex and SpotBugs. The "IC" column shows the initial compilation time (IC) needed for building each project, whereas the "SC" column denotes the subsequent compilation time (SC) for building each project. And the "A(S)" and "A(C)" columns denote the analysis time for SpotBugs and Codegex, respectively. The "S1", "S2" and "S3" columns show the speedup achieved by Codegex over SpotBugs taking the sum of initial compilation time and analysis time (S1); taking the sum of subsequent compilation time and analysis time (S2), and considering only the analysis time (S3). We observe quite impressive speedups up to 24k for initial compilation as it takes time to download dependencies for some Java projects but Codegex does not have this limitation. Moreover, Codegex also outperforms SpotBugs in terms of other speedups (S2 and S3). Specifically, Codegex achieves an average speedup of 237.06 over SpotBugs for S2, and average speedup of 55.72 for S3. Considering only the analysis time for both tools (A(S) and A(C)) for measuring response time, SpotBugs has an average analysis time of 12.67s, whereas Codegex has an average analysis time of 2.18s. On average, the analysis time for Codegex is below Nielsen's 10s recommended threshold for interactive feedback, suggesting that Codegex allows "keeping the user's attention focused on the dialogue", whereas SpotBugs results have exceeded the limit, indicating that "users will want to perform other tasks while waiting for the analysis to finish" [38].

Answer to RQ4: Codegex can provide instant feedback with average analysis time of 2.18s (SpotBugs takes an average of 12.67s).

Effectiveness of Codegex for code review. We evaluate the effectiveness of Codegex in generating code review comments for open PRs by answering the following questions:

RQ5: What is the quality of Codegex's generated comments?
RQ6: How efficient is Codegex in performing code review?

Crawling PRs. Existing datasets for evaluating automated code review approaches are unsuitable for our evaluation because they only contain manual code review comments (ground truth) [39]–[42], which may not cover the problems reported by a static analyzer. To evaluate the real capability of Codegex in generating review comments, we build a crawler to get the 10977 most recently opened PRs in GitHub. We select the PRs that have at least one code change in Java files (because our tool only analyzes Java files), resulting in 4256 PRs from 2769 different projects. Supplementary table shows that evaluated PRs are quite diverse as involved patches that modify 0–25267 lines of code, and span across 1 – 30 files.
Measuring quality of generated reviews. As manually check for the correctness of the 372 generated comments for 4256 PRs is time-consuming, we rely on the developer feedback to measure their qualities. Given a PR and its corresponding feedback f, we manually classify f into:

(AF) Accept and fixed: We consider f to be AF if the developer (1) gave positive feedback or acknowledgment and

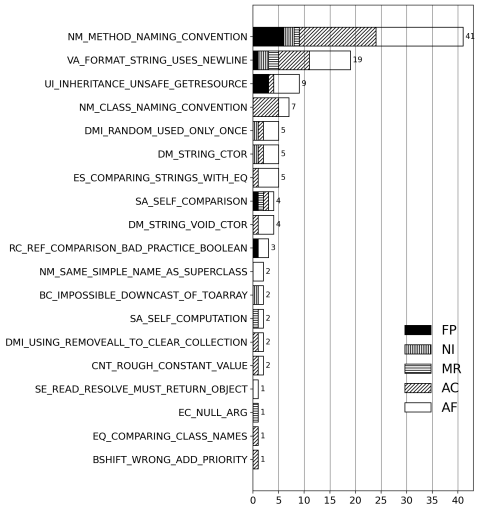

Fig. 2: Quality of Codegex's generated review comments

(2) modified or mentioned they will fix it in the future.

(AC) Accept: We consider f to be AC if the developer (1) gave positive feedback or acknowledgment, or (2) used positive emoji.

(NI) Not interested: We consider f as NI if the developer (1) used neutral emoji, (2) wanted to unsubscribe our service, (3) said that our suggestions have little impact, or (4) ignored our comment.

(MR) Mark as resolved: We consider f as MR if the developer marked our comments as resolved but did not reply.

(FP) Mark as FP: We consider f as FP if the developer (1) used negative emoji, (2) found inconsistency between their code and our comment, or (3) said that our comment is inapplicable.

RQ5: Results for quality of generated reviews. Some code changes have multiple violations for a bug pattern. Instead of leaving a comment per violation, Codegex only gives a comment per bug pattern to avoid spamming developers with too many comments. In total, we received 116 pieces of feedback from developers of corresponding PRs where AF=55, AC=36, NI=7, MR=6, and FP=12. The effective false positives (12/116≈10%) match well with the expectation in prior study [37]. We ran 12 FPs in SpotBugs and found that it gave the same ten FPs. The two remaining FPs are due to the failure of Codegex in matching special AST elements. Figure 2 presents a bar chart where the x-axis shows the number of received feedback for a given category, and the y-axis shows the names of the patterns with at least one feedback. Different shades denotes different categories, where "AF" (unshaded) and "AC" (diagonally shaded) are positive feedback. "NI" (vertically shaded) and "MR" (horizontally shaded) are neutral feedback, whereas "FP" (black) denotes FPs. Overall, most feedback is positive (only five patterns with ≥1 negative feedback).

Answer to RQ5: Among the feedback from 116 that we received from developers, 78.45% of them are positive.

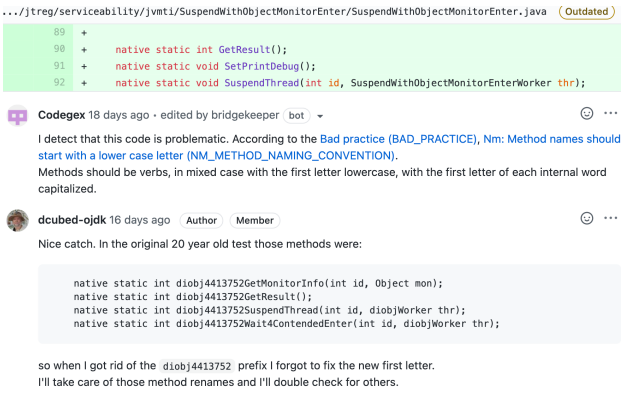

Fig. 3: Example feedback received for a PR in OpenJDK [43]

Example feedback. Figure 2 shows that developers tend to give feedback for patterns related to naming conventions (patterns with names ending with NAMING_CONVENTION). This observation is inline with prior studies of code review that show that developers usually fix maintainability-related issues [44], [45]. Figure 3 shows positive feedback for the NM_METHOD_NAMING_CONVENTION pattern for an OpenJDK's PR. In his response, the OpenJDK developer said that *Codegex helped in detecting method renaming problems in several 20-year-old methods*.Consider another positive feedback for the PR [46] where the developer not only agreed with the Codegex's generated comment, but also added the SpotBugs plugin after getting our automatically generated comment. As SpotBugs and Codegex give similar outputs (all comments in Codegex are derived from bug descriptions in SpotBugs), these developers have mistakenly treated our tool as SpotBugs. This result is inline with our results that show that Codegex can achieve comparable accuracy as SpotBugs. Moreover, the positive feedback also shows the role of Codegex as a *lightweight frontend for SpotBugs that provides better user experience* (i.e., does not require build configuration, and provides quick yet accurate feedback), prompting users to install SpotBugs after trying our lightweight frontend.

RQ6: PR Analysis time. For the 4256 evaluated PRs, Codegex automatically produces 372 review comments. In total, Codegex takes 702 seconds to analyze 4256 PRs. If we turn off the online search strategy, Codegex only takes 168 seconds to analyze 4256 PRs. The average analysis time is 0.039 seconds per PR. These results are below the Nielsen's 0.1 second recommended time limit for a system to react instantenously [38], indicating *the promise of using Codegex in an online setting (as a code review bot for checking PRs)*.

VI. Survey on the Ease of Writing Regex Rules

To evaluate the ease of writing regex rules compared to the non-regex rules in SpotBugs, we surveyed five students at the authors' university (two first-year graduate students and three juniors) for their experience in implementing regex rules in Codegex referring to the corresponding implementations in SpotBugs. The survey contains questions about (1) prior programming experience, (2) the familiarity with regex, and the ease of writing bug detection rules (see supplementary

material for the questions). The participants have 2–5 years of Java programming experience (SpotBugs is written in Java) and 1–2 years of Python programming experience (Codegex is implemented in Python). Overall, our survey results show that although most participants are moderately familiar with regex (average Likert score=2.6 with 5 being expert), they think that *implementing a bug pattern using regex-based rule is easier than using non-regex in SpotBugs* as they rated the difficulty of using regex (average Likert score=2.8 with 5 being very difficult) lower than that of using non-regex (average Likert score=3.6).

VII. Limitations

SpotBugs' limitations. Below are SpotBugs' limitations based on FN^Rs in Table IV:

Missed specific kinds of operands (35/54): As SpotBugs analyzes bytecode, its bug detection has to consider the variants of the same operation given different kinds of operands. For example, when detecting the DMI_RANDOM_USED_ONLY_ONCE pattern that warns when a random object is created and used only once, SpotBugs has FN^Rs when analyzing `int randNumber = new Random().nextInt(99);` since it only checks for instructions that load a local variable but misses those that load a constant (99). Future research can work on testing SpotBugs with different kinds of operands to find these FN^Rs.

Missed compound expressions (8/54): The bug detection rules in SpotBugs usually only consider simple expressions, and may miss violations in compound expressions. For example, when detecting the VA_FORMAT_STRING_USES_NEWLINE pattern that gives a warning when a format string statement includes a newline character `'\n'`, SpotBugs has an FN^Rs for `String.format(var+"GitHub.\n");` with a compound expression `var+ "GitHub.\n"` due to the unsupported string concatenation operation.

Incomplete modeling of sibling types (6/54): As sibling types (e.g., the two floating-point types: `float` and `double`) share similar behaviors, one would expect SpotBugs to give similar warnings during its analysis. However, when checking for the pattern RV_01_TO_INT that reports a warning when a random value is being coerced to the integer value 0, SpotBugs only checks for certain APIs that produce a random value (e.g., `nextDouble()`), and omit other similar APIs with sibling types (e.g., `nextFloat()`).

Handling method calls (4/54): In Section II, we propose expanding the detection of self computation by treating method calls as expression. For example, SpotBugs has an FN^R when checking `size+=(dom.getSegmentAtPos(a).getFrom()-dom.getSegmentAtPos(a).getFrom()+1);` as it fails to detect the self computation in the method call `dom.getSegmentAtPos(a).getFrom()`.

Inconsistent bug description (1/54): We found an inconsistency in the bug description for the EQ_COMPARING_CLASS_NAMES pattern. Specifically, it stated that "This method checks to see if two objects are the same class by checking to see if the names of their classes are equal" so we expect SpotBugs to warn about `c.getClass().getName().equals(c2.getClass().getName())` but it has an FN^R because it only checks if the comparison is inside the `equals` method. When we read the bug descriptions for other related patterns, we think that it should be changed to "This class defines an equals method that ..." [47]. As the users rely on the bug description to understand warnings, future researches could be approaches that automatically detect inconsistencies between bug descriptions of related patterns.

Limitations of a regex-based approach. Based on the FN^Rs for Codegex in Table IV, we identify the following limitations:

Only support single-statement (44/69): Due to the lack of multiline regex support, Codegex fails to detect patterns that require sophisticated analysis (data flow analysis). For example, the pattern DMI_HARDCODED_ABSOLUTE_FILENAME requires checking (1) a `File` object is created, and (2) there exists an absolute path string in the File object creation. As Codegex can only detect explicit object creation (e.g., new File("/abs")), it fails to detect the string usage in indirect object construction (e.g., inside the method `parseZip("/abs")`). In future, we plan to use the multiline mode in the regex library to support more patterns .

Fails to infer data type (24/69): As Codegex relies on explicit type driven matching, it fails to detect patterns that require type inference. For example, when detecting the ES_COMPARING_STRINGS_WITH_EQ pattern which compares `String` objects for reference equality using the `==` or `!=` operators, Codegex fails to warn about the expression `this.getName()==that.getName()` due to failure to infer the return type of the `getName()` method.

Missed special AST elements (1/69): As Codegex does not parse code into ASTs, it may miss some AST nodes. For example, when checking the NM_CLASS_NAMING_CONVENTION pattern that checks for upper camel cases, Codegex did not warn about the `enum complexFeaturesAppendEnum` expression because we do not consider `enum` as a type of special Java class.

VIII. Related Work

Enhancing static analysis. Several techniques were proposed to prioritize more important generated warnings in FindBugs [13], [14], [36]. Previous work (e.g., [5], [48]–[50])focus on improving performance of static analysis via staged analyses. Although Codegex uses a two-stage approach for quick analysis, it uses different techniques (regex rules and several strategies) from existing methods. Several approaches support analysis of partial programs [7], [51], [52]. Prior framework resolves ambiguities in partial Java programs via several heuristics [7]. Although several strategies are used to improve accuracy of Codegex, it can analyze smaller programs (programs with only one statement versus one class in prior work [7]), and it is designed for bug pattern detection whereas prior work is designed for type inference. The most relevant

work, μchex, performs bug detection on AST nodes built from code snippet by sliding window and micro-grammars [51]. While it benefits from the strength of tree representation for complex analysis (e.g. flow analysis), Codegex improves the comprehensibility and ease of implementation of bug detection rules based on text representation. Codegex did not compare with μchex since it is not open-source.

Code Reviews. Prior automated code review approaches either rely on deep learning for modeling code changes and review comments [39]–[41], [53] or code reviewer recommendation [54], [55]. Although these techniques can potentially discover new problems in given code changes, they are more suitable for code review of mature projects where many PRs and review comments exist. Codegex can handle any type of projects, including new projects which do not have enough review comments for training. Meanwhile, several studies have shown the usefulness of static analysis in automated code review [16], [17], [56], [57]. One of the most relevant works, Review Bot, produces review comments based on the output of static analyzers [18]. While Review Bot solves the compilation requirement of SpotBugs using a workaround, Codegex improves over SpotBugs via regex rules and heuristics to skip compilation.

Regex-based approaches. Regex matching has been widely used in many tasks (e.g., mutant generation [58] and detecting security vulnerabilities [59]–[61]). In the security domain, DevSkim is an IDE plugin [61] that uses regex rules for inline checks of security vulnerabilities (e.g., invoking dangerous API like `strcpy`). Regex rules cannot be directly used in other domains to detect bug patterns in SpotBugs because (1) a general-purpose analyzer may have rules that are not domain-specific and may require more contexts for accurate detection, and (2) these tools are not designed for checking partial programs. While Codegex uses regex as its core for analysis, it differs from existing approaches in several aspects: (1) it uses strategies to improve effectiveness, and (2) it is more general than domain-specific techniques that target security vulnerabilities.

A. Threats to Validity

External. Our study and our evaluation results may not generalize beyond the evaluated open-source Java projects. To mitigate this threat, we include a large number of open-source projects of diverse sizes. We also ensure that the projects used in the two experiments (Section V) do not overlap. Since we only evaluate on SpotBugs, our results may not generalize to other static analysis tools (e.g., Infer [31]). During the manual inspection of the generated warnings, two authors of the paper reviewed the results independently and met to resolve any disagreement. Moreover, whenever we found a bug or an FN in SpotBugs, we confirmed its validity with the developer by filing bug reports, leading to a total of 16 bug reports. Furthermore, due to limited resources, we conduct all experiments on a single machine. Although the initial compilation time and subsequent compilation time depend on the machine used and network latency, our results show that the speedup that Codegex offers is quite substantial.

Internal. Our code and scripts may have bugs that could affect our results. To mitigate this threat, we wrote tests for each implemented pattern. Moreover, we evaluate the effectiveness of Codegex in automated code review via developers' feedback as it is time-consuming to manually label each warning. We mitigate this threat by manually analyzing each feedback.

Ethical considerations. In principle, it is straightforward to extend Codegex to a bot automatically triggered after submitting a new PR. However, to ensure reproducible results, we first obtained a fixed list of open PRs, and then run Codegex to generate review comments for each PR in the list (i.e., our experiments only affect developers of the 372 PRs with review comments). Instead of spamming developers by contacting them via survey [62], we evaluate whether Codegex is ethical based on bot ethics [19] which checks if the bot: (1) is lawbreaking, (2) involves deception, and (3) violates social norms. To check for law breaking, we (1) obtained ethical approval from the Institutional Review Board (IRB) of our institute, and (2) manually checked the contribution guidelines, and signed the Contributor License Agreement (CLA) of each repository in the PR list (in most CLAs, "submitted" includes any form of communications, which covers code review comments). Overall, there are 31 projects with CLAs, and we have signed all of them. To avoid deception, we did not hide the fact that the comments are sent by a bot. In fact, six developers are aware that the comments are sent by a bot (e.g., one developer replied to us saying "good bot"). For (3), the fact that our bot achieves similar accuracy as SpotBugs (a widely used tool) for many patterns shows that our bot is beneficent [63] and did not "create more evil than good". Moreover, we also manually replied to 57 developers to discuss the analysis results.

IX. Conclusion

We present Codegex, a novel regex-based approach for efficient static analysis. To perform fast yet accurate analysis, our approach uses several heuristics to enrich the analysis contexts. Our experiments that compare Codegex and SpotBugs show that Codegex can analyze up to 590X faster than SpotBugs with comparable accuracy. For automated code review, we evaluate Codegex against 4256 PRs, and received 116 feedback where 78.45% are positive. Our experiments confirm the two settings in which Codegex can enhance existing static analyzers like SpotBugs, including: (1) acting as the fast stage in a two-stage approach where more sophisticated analysis can be run as part of a nightly build, and (2) supporting incremental analysis that performs automated code review for partial code in a PR without setting up build configurations.

X. Acknowledgement

This work was supported by the National Natural Science Foundation of China (Grant No. 61902170).

REFERENCES

[1] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 672–681.

[2] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, 2016, pp. 332–343.

[3] *Coverity*, 2022. [Online]. Available: https://scan.coverity.com/

[4] *Fortify*, 2022. [Online]. Available: https://www.microfocus.com/en-us/cyberres/application-security

[5] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 307–317.

[6] T. Clem and P. Thomson, "Static analysis at github: An experience report," *Queue*, vol. 19, no. 4, p. 42–67, aug 2021. [Online]. Available: https://doi.org/10.1145/3487019.3487022

[7] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, 2008, pp. 313–328.

[8] D. A. Tomassi, "Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 980–982.

[9] *[java] Allow MissingOverride rule to work without bytecode*, 2020. [Online]. Available: https://github.com/pmd/pmd/issues/2428

[10] *Analyze Java without running the build*, 2018. [Online]. Available: https://github.com/facebook/infer/issues/969

[11] *Analyse jar and not java file*, 2018. [Online]. Available: https://github.com/spotbugs/spotbugs/issues/695

[12] Checkstyle. [Online]. Available: http://checkstyle.sourceforge.net/

[13] H. Shen, J. Fang, and J. Zhao, "EFindBugs: Effective error ranking for findbugs," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2011, pp. 299–308.

[14] S. Heckman and L. Williams, "A systematic literature review of actionable alert identification techniques for automated static code analysis," *Information and Software Technology*, vol. 53, no. 4, pp. 363–387, 2011.

[15] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger, "Comparing bug finding tools with reviews and tests," in *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems*, ser. TestCom'05. Berlin, Heidelberg: Springer-Verlag, 2005, p. 40–55. [Online]. Available: https://doi.org/10.1007/11430230_4

[16] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 161–170.

[17] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *2017 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2017, pp. 101–105.

[18] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 931–940.

[19] C. A. de Lima Salge and N. Berente, "Is that social bot behaving unethically?" *Communications of the ACM*, vol. 60, no. 9, pp. 29–31, 2017.

[20] Network status check & optimize up log. [Online]. Available: https://github.com/qiniu/android-sdk/pull/453/files/7356a6c3eccb942fa90edc9e537b36b349ebc89a#diff-7f67181

[21] Sa_local_self_computation bug. [Online]. Available: https://github.com/spotbugs/spotbugs/issues/1472

[22] False negatives on sa_field_self_computation and sa_local_self_computation. [Online]. Available: https://github.com/spotbugs/spotbugs/issues/1473

[23] The architecture of findbugs. [Online]. Available: https://github.com/spotbugs/spotbugs/blob/a6f9acb2932b54f5b70ea8bc206afb552321a222/spotbugs/design/architecture/architecture.tex

[24] C. Cifuentes and B. Scholz, "Parfait: designing a scalable bug checker," in *Proceedings of the 2008 workshop on Static analysis*, 2008, pp. 4–11.

[25] Five invaluable techniques to improve regex performance. [Online]. Available: https://www.loggly.com/blog/five-invaluable-techniques-to-improve-regex-performance/

[26] C. Chapman and K. T. Stolee, "Exploring regular expression usage and context in python," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 282–293.

[27] N. Ayewah and W. Pugh, "The google findbugs fixit," 01 2010, pp. 241–252.

[28] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*, 2007, pp. 27–27.

[29] Pmd. [Online]. Available: https://pmd.github.io/

[30] Jlint. [Online]. Available: http://jlint.sourceforge.net/

[31] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of C programs," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 459–465.

[32] Error prone. [Online]. Available: https://errorprone.info/

[33] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "There and back again: Can you compile that snapshot?" *Journal of Software: Evolution and Process*, vol. 29, no. 4, p. e1838, 2017.

[34] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, 2007, pp. 1–8.

[35] A. Habib and M. Pradel, "How many of all bugs do we find? A study of static bug detectors," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018, pp. 317–328.

[36] S. Kim and M. D. Ernst, "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*, 2007, pp. 45–54.

[37] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Communications of the ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[38] J. Nielsen, *Usability engineering*. Morgan Kaufmann, 1994.

[39] J. K. Siow, C. Gao, L. Fan, S. Chen, and Y. Liu, "CORE: Automating review recommendation for code changes," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 284–295.

[40] A. Gupta and N. Sundaresan, "Intelligent code reviews using deep learning," in *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'18) Deep Learning Day*, 2018.

[41] S.-T. Shi, M. Li, D. Lo, F. Thung, and X. Huo, "Automatic code review by learning the revision of source code," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 4910–4917.

[42] R. Chatley and L. Jones, "Diggit: Automated code review via software repository mining," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 567–571.

[43] 8262881: port JVM/DI tests from JDK-4413752 to JVM/T. [Online]. Available: https://github.com/openjdk/jdk/pull/2899

[44] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: Which problems do they fix?" in *Proceedings of the 11th working conference on mining software repositories*, 2014, pp. 202–211.

[45] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.

[46] Changelog. [Online]. Available: https://github.com/foo4u/conventional-commits-for-java/pull/12

[47] Inconsistent bug description on eq_comparing_class_names. [Online]. Available: https://github.com/spotbugs/spotbugs/issues/1523

[48] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 2011, pp. 289–298.

[49] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta, "Fast and accurate static data-race detection for concurrent programs," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 226–239.

[50] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay, "Effective typestate verification in the presence of aliasing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 2, pp. 1–34, 2008.

[51] F. Brown, A. Nötzli, and D. Engler, "How to build static checking systems using orders of magnitude less code," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016, pp. 143–157.

[52] H. Zhong and X. Wang, "Boosting complete-code tool for partial program," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 671–681.

[53] H.-Y. Li, S.-T. Shi, F. Thung, X. Huo, B. Xu, M. Li, and D. Lo, "Deepreview: automatic code review using deep multi-instance learning," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2019, pp. 318–330.

[54] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 141–150.

[55] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?" *Information and Software Technology*, vol. 74, pp. 204–218, 2016.

[56] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1419–1457, 2020.

[57] M. V. Mäntylä and C. Lassenius, "What types of defects are really discovered in code reviews?" *IEEE Transactions on Software Engineering*, vol. 35, no. 3, pp. 430–448, 2008.

[58] A. Groce, J. Holmes, D. Marinov, A. Shi, and L. Zhang, "An extensible, regular-expression-based tool for multi-language mutant generation," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 2018, pp. 25–28.

[59] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu, "Fast regular expression matching using small tcams for network intrusion detection and prevention systems," in *Proceedings of the 19th USENIX conference on Security*, 2010, pp. 8–8.

[60] T. Liu, Y. Sun, A. X. Liu, L. Guo, and B. Fang, "A prefiltering approach to regular expression matching for network security systems," in *International Conference on Applied Cryptography and Network Security*. Springer, 2012, pp. 363–380.

[61] Devskim. [Online]. Available: https://github.com/microsoft/devskim

[62] S. Baltes and S. Diehl, "Worse than spam: Issues in sampling software developers," in *Proceedings of the 10th ACM/IEEE international symposium on empirical software engineering and measurement*, 2016, pp. 1–6.

[63] N. E. Gold and J. Krinke, "Ethical mining: A case study on msr mining challenges," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 265–276.