User-Assisted Code Query Optimization

Ben Liblit
Amazon
USA

Yingjun Lyu
Amazon
USA

Rajdeep Mukherjee
Amazon
USA

Omer Tripp
Amazon
USA

Yanjun Wang
Amazon
USA

Abstract

Running static analysis rules in the wild, as part of a commercial service, demands special consideration of time limits and scalability given the large and diverse real-world workloads that the rules are evaluated on. Furthermore, these rules do not run in isolation, which exposes opportunities for reuse of partial evaluation results across rules. In our work on Amazon CodeGuru Reviewer, and its underlying rule-authoring toolkit known as the Guru Query Language (GQL), we have encountered performance and scalability challenges, and identified corresponding optimization opportunities such as, *caching*, *indexing*, and *customization of analysis scope*, which rule authors can take advantage of as built-in GQL constructs. Our experimental evaluation on a dataset of open-source GitHub repositories shows 3× speedup and perfect recall using indexing-based configurations, and 2× speedup and 51% increase on the number of findings for caching-based optimization.

CCS Concepts: • **General and reference** → **Performance**; *Experimentation*; • **Theory of computation** → **Automated reasoning**; **Programming logic**; • **Software and its engineering** → *Software defect analysis.*

Keywords: AWS, caching, GitHub, Guru Query Language (GQL), performance optimization, static analysis

ACM Reference Format:
Ben Liblit, Yingjun Lyu, Rajdeep Mukherjee, Omer Tripp, and Yanjun Wang. 2023. User-Assisted Code Query Optimization. In *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '23), June 17, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3589250.3596148

SOAP '23, June 17, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0170-2/23/06.
https://doi.org/10.1145/3589250.3596148

1 Problem Setting

Amazon CodeGuru Reviewer [10] is a commercial product that performs source-code repository scans as well as integrates into the code review process as an automated reviewer, leaving comments on pull requests. Its underlying architecture is based primarily on "micro-analyzers", which run narrow yet precise analysis scenarios. These are built atop a common abstraction layer, GQL, which contains reusable constructs such as forward/backward slicing, taint analysis, and filters to match code entities based on data types or call signatures.

GQL is our main vehicle to "democratize" CodeGuru Reviewer by empowering domain experts to directly specify, then tune and productionize, micro-analyzers. The GQL toolbox provides the building blocks for such micro-analyzers, which the expert then composes to express a property of interest, e.g. regarding correct usage of some cryptography or machine-learning library.

Our experience in supporting rule authors, and growing the CodeGuru Reviewer rule base, has exposed many cases where rules can — and in some cases, should — be optimized to run faster and make more frugal usage of compute and memory resources. This, in turn, has led us to design and implement several optimization features as part of the GQL toolbox, which are made available to rule authors to tune their rules' performance and resource consumption. In what follows, we set up the technical background for these optimizations, then describe them and report on their impact.

2 Background

CodeGuru supports Java and Python, and integrates with different code hosting platforms including GitHub and BitBucket. CodeGuru supports three code scanning modes:

- Incremental: A code review is created automatically when a pull request is raised.
- Full: The entire code base is analyzed upon request from a developer.
- CI/CD: The entire code base is analyzed as part of CI/CD workflows.

In any of the above modes, CodeGuru operates by (1) constructing an analysis-friendly *intermediate graph representation* of the target code base, then (2) applying a set of *rules* to

search for graph nodes in that representation that correspond to buggy code patterns.

2.1 Intermediate Graph Representation

CodeGuru's intermediate representation is the *MU graph* [21]. A MU graph is essentially a data-dependence graph overlaid with a control-flow graph, all in static single assignment (*SSA*) form. An individual MU graph node might represent a piece of data, an action that transforms input data into output data, or a control operation such as a branch. Nodes and edges carry additional details specific to their type and role. For example, a single action node that represents a function call might have:

- zero or more incoming data edges, each from some data node representing an argument to the call;
- an optional outgoing data edge, the target of which is some data node that receives the result of the call; and
- one incoming and one outgoing control edge, connected to the action or control nodes that execute immediately before or after this call.

The MU graph representation is language-independent. Actions are fairly fine-grained, and unnamed temporary values are made explicit. For example, the representation of print(a + b) would include a sum action node; a call action node; and three data nodes representing a, b, and the unnamed temporary value of a + b.

2.2 Rules

Finding buggy code patterns in a fine-grained MU graph can be cumbersome. To make this task easier, CodeGuru includes the Guru Query Language (*GQL*), a domain-specific language for operating on MU graphs [21]. A GQL rule consists of a sequence of operations on a set of MU graph nodes, called the *match frontier*. The match frontier is initially the set of all nodes in the MU graph representation of one function. GQL operations transform this set, such as by filtering it or by traversing the graph in a systematic way.

For example, one GQL operation might filter the match set to only the subset of data nodes that represent string literals. Another operation might transform each node in the match set to its data-flow successor. A higher-order operation could repeat the previous transformation while collecting a fixed-point. By chaining together these and a few more operations, one might create a rule that identifies all literal strings that can transitively flow into the second argument of a call to a function named "login". Thus, we have built a rudimentary rule that detects hard-coded passwords.

GQL is implemented as a Java library that relies heavily on the builder pattern. Starting with a fresh builder, one adds operations using calls like withDataByTypeFilter(...) or withOutNodesTransform(...). A final call to build() returns a constructed rule: an instance of GQL's CustomRule type that can be applied to functions or whole programs to

```
1  public void doPost(
2      HttpServletRequest servletRequest,
3      HttpServletResponse servletResponse,
4      FilterChain chain) {
5    String user =
6      servletRequest.getParameter("user");
7    String userPath = ".\\Data\\" + user;
8    ...
9    findUserDirectory(userPath);
10   ...}
11
12 private void findUserDirectory(String userPath) {
13   ...
14   File file = new File(userPath);
15   if (file.exists() && file.isDirectory()) {
16     String[] commands =
17         { "/bin/sh", "-c", "ls " + userPath };
18     Process process =
19         Runtime.getRuntime().exec(commands);
20     ...
21   }...}
```

Figure 1. Code snippet demonstrating Injection vulnerabilities

detect bugs. Unlike other rule-based static analysis languages such as, CodeQL [1], GQL does not require building the codebase (then compilation of the facts database), which limits adoption, blocks use cases like ad-hoc queries, etc. In terms of analysis capabilities, GQL offer codebase-wide data-flow and type-state capabilities, that is, deeper and more semantic analysis, unlike tool such as, Semgrep [2].

3 Motivating Example

To illustrate the insights feeding into the optimizations described in this paper, and the benefits that these optimizations introduce, we consider the code example in Figure 1, inspired by real-world code that our rules were evaluated on, where (untrusted) user input read via the getParameter call at line 6 reaches both the File constructor at line 14 and the exec call at line 19 through inter-procedural data flow. These flows give rise to path traversal and command injection vulnerabilities, respectively.

The corresponding CustomRule rule excerpt is shown in Figure 2. This rule detects taint-flow by using different optimization strategies, such as caching that caches various taint sources, configuration based indexing that dynamically indexes into a matching taint configuration, and specification of the tracking and analysis scope. In what follows is the discussion of the rule in Figure 2 using the code example in Figure 1.

```
1  CustomRule rule = new CustomRule.Builder()
2      ...
3      .withCachedDependency(b -> b
4        .withRuleConfigurationItemMatchFilter(
5          "$.Sources[*].method",
6          (n,c) -> n.isCall() && n.getName().matches(c))
7        .withInterproceduralDataDependentsTransform(
8            TrackingScope.FILE))
9      ...
10     .build();
```

Figure 2. Rule snippet demonstrating caching and configuration indexing

Caching. Injection vulnerabilities, such as those illustrated in Figure 1, are typically modeled as taint problems, where source/sink reachability is checked. The sources are often shared in common across multiple vulnerability categories, since these represent the reading of untrusted data into the program's state.

Caching provides a medium to exploit the following observation. Forward data-flow slices, starting from sources, can be computed once per function, then reused across other rules that agree on the sources as well as validators and sanitizers. In this case, reuse enables amortization across the path traversal and command injection rules.

The withCachedDependency statement at line 3 in Figure 2 illustrates this scenario. The subrule logic in the cached block reads sources from a configuration, then performs forward slicing from these sources.

Configuration indexing. Many rules are backed by a configuration, where the rule serves as a "template" that can be instantiated to model different code scenarios.

In a production setting, these configurations can reach the order of 10,000 entries, if not more, which mandates efficient handling. Brute-force iteration over the configuration to identify matching code entities (for example, source or sink calls) becomes prohibitive. We later describe an "inversion" of the configuration lookup, where an index is computed and code entities are then represented as keys enabling constant-time index lookup.

The withRuleConfigurationItemMatchFilter statement at line 4 in Figure 2 corresponds to this optimization. We omit the code to index into the configuration for space constraints, and instead focus on how the configuration is accessed. The first argument is a JSONPath query specifying which configuration items should be matched against entities in the code, whereas the second argument relates nodes n in the graph representation to configuration items c: in the example, call nodes whose name matches the configuration item.

Scope customization. In GQL, taint queries can be composed with other constructs, as well as instantiated in different ways at different points in the overall query. We have observed that in some cases, the return-on-investment from limiting the scope of a taint query, where we trade off time/resource costs versus recall, leans towards running the query on a smaller scope.

For the example in Figure 1, constraining the taint query to functions in the same file (while excluding functions in other files in the same repository) enables more precise analysis (less room for error, for example due to incorrect call resolutions), alongside faster and more resource-efficient analysis. The scope specification appears as the TrackingScope.FILE argument to the slicing operation at line 7 in Figure 2.

4 Optimization Strategies

4.1 Caching

The caching algorithm is based on a simple yet important observation. Given rules r_1 and r_2 with respective subrules sr_1 and sr_2, if

1. sr_1 and sr_2 are evaluated on equivalent states;
2. sr_1 and sr_2 perform the same operations; and
3. sr_1 and sr_2 both have sufficient analysis budget to complete their evaluation, or else both lack sufficient budget to complete their evaluation,

then the evaluation result due to sr_1 in the context of r_1 can be "reused" for sr_2 in the context of r_2, and vice versa. In what follows, We go over these criteria, and the meaning of "reuse", in turn.

Starting from the first criterion, a rule evaluation state consists of (i) the incoming match frontier, (ii) the match frontiers stored as variables (or IDs), and (iii) any additional metadata stored as part of the state. State equivalence reduces to equivalence along these three dimensions.

Rule as well as subrule isomorphism is checked in an inductive manner. Starting from the base case of atomic operations, these are compared directly. Composite operations, which consist of subrules and the operations therein (for example, withAnyOf or withAllOf), are compared starting from the subrules comprising them.

Finally, we check the analysis budgets attached to sr_1 and sr_2, where a budget is a bag of aspects, an aspect being a measurable "cost unit": wall-clock time, number of atomic analysis operations executed, number of functions visited during operation evaluation, and so on. We ensure that the budgets are *compatible*, in that both are simultaneously either sufficient to complete evaluation of sr_1 and sr_2, respectively, or both would be exhausted during subrule evaluation.

Assuming sr_1 and sr_2 are isomorphic and have compatible analysis budgets b_1 and b_2, respectively, the application of sr_1 to state σ_1 can be reused for sr_2 and σ_2 provided $\sigma_1 \equiv \sigma_2$, where by reuse, we mean that

1. the output state $\hat{\sigma}_1$ due to sr_1 is provided as the result of $[\![sr_2]\!]\sigma_2$; and
2. the budget cost recorded during evaluation of $[\![sr_2]\!]\sigma_2$ is deducted from sr_2's budget.

At the implementation level, the caching algorithm is built atop a thread-safe map. Map keys are rule/input pairs, where the values are the respective evaluation results. The caching algorithm checks, in an atomic block, whether the mapping is already established. If not, then the value is computed and inserted into the map.

While designed to be generic, caching is particularly useful when a potentially expensive subrule with a same set of matching frontier is embedded inside multiple rules. For example, taint tracking is a particularly helpful application of caching. Consider, as an example, distinct injection rules that share the same user input surface, thus same sources, yet differ in terms of sinks. The subrule that computes the forward slice from sources can be cached, hence amortized across all rules with only one of the rules performing the evaluation. This needs not be explicitly coordinated across the rules. Suffice it that they all wrap this evaluation step into a withCachedDependency statement, as shown at line 3 of Figure 2, and the reuse will emerge at run time. It is worth noting that caching is not free. There are performance overheads of writing to and reading from the cache. More importantly, there is a memory cost. Given that the memory used for caching is not unlimited, users shall use that memory to cache the expensive operations to optimize the performance gains of withCachedDependency.

4.2 Configuration Indexing

Analysis rules often cover multiple scenarios from one or more libraries. Examples include (i) flagging deprecated methods in the AWS Java API; (ii) tracking untrusted data from APIs that read user input; or (iii) checking that Closeable types are used correctly.

These are examples of rules backed by a configuration, listing the different instances that the rule logic applies to. In our experience, these configurations can reach the order of 10,000 entries, if not more. A naïve approach for evaluating configuration-backed rules is to iterate over all the configurations when evaluating the rule on a function f, for example by matching all calls made by f to a deprecated API, as listed in the configuration. For a configuration C, this means that the rule is evaluated, fully or in part, $|C|$ times on f. That is, evaluation time grows linearly with the size of the configuration.

We have designed and implemented an alternate scheme, where evaluation time is fixed irrespective of $|C|$. Our scheme stems from the observation that the configuration relates to entities in the code. Thus, we can start from the function under analysis, and relate entities therein to the configuration. As a simple example, for deprecated APIs, we can mine all the function calls in the function, and consult the configuration for any matches.

More generally, configuration indexing is backed by two functions provided by the rule author:

- An indexing function ι, mapping the configuration items $c \in C$ to key/value pairs $c \mapsto k$.
- A mapping function τ from entities in the code to the same domain of keys plus $\perp$ (for configuration-irrelevant entities).

Back to the example of deprecated APIs, the keys are the names of deprecated functions, i.e. ι projects deprecated API configurations — consisting of the AWS service, declaring class, and API name — on the API name as the key, whereas τ maps function calls within the target analysis scope to the callee name (and other code entities, like variables and control statements, to $\perp$). Thus ι, starting from configuration items, and τ, starting from the target scope, agree on how the configuration would be searched based on the code being analyzed: via function names.

With this "inversion", and assuming a good indexing function (such that there are few collisions, thus effective distribution across buckets), consulting the configuration requires nearly constant time regardless of its size. In practice, this has proven easy to achieve, since we typically make use of types and identifiers. The indexing and mapping functions are then both cheap to compute and yield effective distribution of configuration items.

5 Evaluation

In this section, we report on experimental evaluation.

5.1 Input Dataset

We have conducted the experiments on GitHub packages that have Apache or MIT licenses, and popularity of at least 4 stars. To evaluate the impact of different optimization strategies, we have selected two different datasets. The first dataset was used to evaluate configuration indexing optimization strategy. It consists of 200 randomly selected Java and Python GitHub repositories which have specific SDK usages, such as AWS Java SDK [9] or AWS Python SDK [8]. The second dataset was used to evaluate different caching strategies and analysis scopes. It consists of another 180 randomly selected Java GitHub repositories which have specific APIs that are identified as tainted sources. The average number of lines of code in repositories from the dataset is 25697.

5.2 Experimental Setup

The experiments were run on an Amazon EC2 machine with 48 cores, 384 GB of memory, and 2 hard drives of size 1 TB each. We have selected 5 AWS best practice rules and 7 taint-flow rules to demonstrate the impact of the configuration indexing, and caching, respectively. Depending on the usage scenarios, users could have different requirements about

time limits to run the analysis. For example, an offline scan could have a longer time limit, while an online scanning during Code Review typically demands a shorter time limit. We evaluated our rules on open-source GitHub packages, with a time limit of 30 minutes and 5 minutes per package.

5.3 Experiment 1: Configuration Indexing

Table 1 presents the impact of indexing based configuration using 5 CodeGuru rules [24], that specifies a set of guidelines for correct, secure, and performant usage of AWS cloud Java and Python SDKs.

Column 1 in Table 1 gives the rule id, and Column 2 presents the total number of configurations that each rule evaluates on. Columns 3–4 report the run times and number of findings or detection from the rules *without* indexing optimization. Columns 5–6 report the same *with* indexing optimization. Comparing the run times of the rules without indexing and with indexing in Table 1, it is evident that when the total number of configurations are large (>1,000), the unoptimized rules without indexing, *Rule 1* and *Rule 2*, timed out. The evaluation time of the unoptimized rules grow linearly with the size of the configuration that the rules operate on. For rules that evaluate on few hundred configurations, such as *Rule 3*, *Rule 4*, and *Rule 5*, the speedup is 3× or more. Furthermore, the number of findings (reported in #Findings) show that the unoptimized rules, *Rule 1* and *Rule 2*, did not produce any findings, while the optimized rules produced same number of findings for different time limits. This demonstrates that the dynamic indexing of configurations help uncover more number of bugs overall.

5.4 Experiment 2: Caching and Scope Customization

In this experiment, we evaluated the impact of caching and scope customization. We ran seven rules, targeting different kinds of injection vulnerabilities, including command injection, SQL injection, cross-site scripting, log injection, path traversal, LDAP injection, and XPath injection [24]. All the rules shared a same set of tainted sources, the vast majority of which represent data coming from the Internet and is generally considered to be untrusted input to the program. Depending on the injection issue, the rules differ in sinks. For example, the rule for command injection considers APIs responsible for OS command execution as sinks. We used various combinations of analysis scopes (i.e., file-level and package-level) and caching configurations (i.e., with and without caching) on these rules. We ran these rules against 180 repositories in the second dataset with a time limit of 5 minutes and 30 minutes.

The results based on a time limit of 5 minutes are shown in Table 2. In this table, column 1 indicates whether the static analyzer performed a whole-program inter-procedural analysis, i.e, package-level, versus, a more contained file-level inter-procedural analysis. Columns 2–3 list the caching configuration we set for each rule. We experimented with different cache sizes, which specify the maximum number of cached tainted program points. Column 4 presents the number of cache hits and misses for each experiment. Under columns 5–9, we first list the evaluation time for all the rules, and then for the first rule, and then for the rest of the rules. The reason of splitting the rules in this way is to show the effect of caching. We also present the average and median evaluation time it takes for analyzing a repository. Column 10 summarizes the number of findings we obtained for all the rules. Column 11 reports the number of repositories that our analysis timed out on the given time limit. Due to the space constraint, we did not list the results based on a time limit of 30 minutes. We will discuss about the numbers during comparison.

Impact of scope customization: Our evaluation results demonstrate the importance of customizing the analysis scope. When caching is unavailable, file-level analysis scaled well on a time limit of 5 minutes. Comparing to package-level analysis, the speedup of the total rule evaluation time was more than 1.5×, which also reduced the number of timeouts from 17 repositories to 1. As for the number of findings, the file-level analysis only reported 29 fewer findings (8% less) than the package-level analysis. These numbers suggest that given a short time limit, even if we enabled package-level analysis, the analysis was not able to scale properly without caching. On the other hand, file-level analysis performed well in terms of meeting the time limit without sacrificing too much recall.

If users have more budgets in terms of time limit and are willing to increase the limit to 30 minutes, we observed obvious improvement on recall using package-level inter-procedural analysis. Even without caching, comparing to file-level analysis, the number of findings increased 56% from 339 to 529. When caching is in place, the improvement is even more significant, as we discuss below.

Impact of caching: Results show that caching can significantly improve the recall of package-level inter-procedural analysis. When the time limit is 5 minutes, the speedup of overall rule execution was more than 1.7×, comparing to no caching. This directly resulted in an increased number of findings by 44% from 343 to 495. When the time limit is 30 minutes, the speedup was even increase to 2×. The number of findings was increased by 51% from 529 to 799. Caching is effective even when the time limit is 5 minutes. Comparing to no caching, only the analysis time of the first rule slightly increased, likely caused by the overhead from cache writes. Such overhead was well offset by later-on savings when the rest of the rules were executed. If users have more budgets on memory, they can increase the cache size and maximize the benefits of caching. Looking at the last row of Table 2 where a cache size of 100,000 was used, the six rules that can make use of cached entries in total only took 15% more analysis time than the time of the first rule alone. The overall

Table 1. Comparison for rules with and without configuration indexing. Evaluation times are in seconds, given as "x / y" for for 30-minute and 5-minute limits, respectively.

Rule	# Configurations	Without Indexing		With Indexing	
		Evaluation Times	# Findings	Evaluation Times	# Findings
		30 mins / 5 mins		30 mins / 5 mins	
Rule 1 [6]	1,117	Timeout / Timeout	N/A	215.3s / 215.3s	78
Rule 2 [3]	8,411	Timeout / Timeout	N/A	296.7s / 296.7s	136
Rule 3 [5]	81	427.7s / Timeout	32	144.5s / 144.5s	32
Rule 4 [4]	186	694.5s / Timeout	56	139.4s / 139.4s	56
Rule 5 [7]	126	673.1s / Timeout	47	126.4s / 126.4s	47

Table 2. Comparison for rules with and without caching at file or package scope with time limit of 5 minutes.

	Cache			Analysis Time (in seconds)						
Scope	Configuration	Size	# Hits/Misses	All Rules	First Rule	The Rest	Mean	Median	# Findings	# Timeouts
File	Disabled	N/A	N/A	2,796.4	443.0	2,353.4	15.7	2.7	314	1
File	Enabled	10,000	39,800/6,653	2,730.9	442.0	2,288.9	15.3	2.6	314	1
File	Enabled	100,000	39,800/6,653	2,725.8	440.3	2,285.5	15.3	2.7	314	1
Package	Disabled	N/A	N/A	4,327.6	1,121.3	3,206.3	39.1	3.7	343	17
Package	Enabled	10,000	20,565/5,597	2,679.9	1,138.5	1,541.4	28.4	3.0	406	12
Package	Enabled	100,000	21,705/4,651	2,448.8	1,134.5	1,314.3	26.9	2.9	495	11

speedup helped to discovered 44% more findings given the short time limit.

Caching was even more effective when the time limit is 30 minutes, comparing to the time limit of 5 minutes. The speedup and the percentage of increased number of findings both improved. When the time limit was longer, the first rule had its chance to finish on more complex repositories and wrote to the cache. Due to complex taint flows in these repositories, a larger cache size was needed otherwise the cache can only hold a portion of the tainted program points. Once the required resources on both time and memory were met, users can maximize the benefits of caching.

6 Related Work

Toman and Grossman [26] note caching as widely used technique to make static analysis tractable [11, 12, 18–20, 22, 25], but limited to reanalysis of the same program or of shared library code [17]. Prior work on analysis caching has generally keyed the cache on coarse-grained program components, such as functions or files. By contrast, we can cache results of whole rules, subrules, or even individual GQL operations. Our approach is well-matched to a feature-rich analysis service that checks many aspects of a single code base [24], as our cache can accelerate common intermediate steps across multiple rules. Our focus on efficiently applying many checks to varied programs contrasts with, and is complementary to, that of Gu et al. [13], who focus on scaling any single analysis to large programs.

Toman and Grossman [26] propose a community database of analysis-relevant API information. If this effort succeeds, then the sheer number of annotated APIs may become a scaling challenge. We have shown that configuration indexing works well for rules that operate with thousands of configurations, that are mined from different SDKs.

Schubert et al. [23] discuss the importance of understanding analysis performance so that it can be tuned to perform well. Toman and Grossman [26] also note the use of tunable "knobs" to balance precision and performance [14–16]. Our analysis scopes are one such group of knobs, but we have not detailed a procedure for selecting the best scopes for any given task. The instrumentation-directed strategies of Schubert et al. [23] are likely applicable here.

7 Conclusion

In this paper, we have presented an interactive approach for rule authors—encoding their domain expertise as GQL rules evaluated through Amazon CodeGuru Reviewer—to optimize their rules' performance. Specifically, rule authors can (i) cache rule steps for reuse by co-evaluated rules; (ii) control the scope of interprocedural queries at a granular level; as well as (iii) scale a rule "template" to a large number of configurations using efficient indexing. Our evaluation of these optimizations on a GitHub dataset indicates significant performance gains, e.g. ×3 speedup thanks to configuration indexing and ×2 speedup thanks to caching.

References

[1] 2019. CodeQL. https://codeql.github.com

[2] 2020. Semgrep. https://semgrep.dev

[3] 2022. CodeGuru Rule: Batch request with unchecked failures. https://docs.aws.amazon.com/codeguru/detector-library/java/aws-unchecked-batch-failures/.

[4] 2022. CodeGuru Rule: Check uncaught exceptions High. https://docs.aws.amazon.com/codeguru/detector-library/java/check-uncaught-exceptions/.

[5] 2022. CodeGuru Rule: Inefficient polling of AWS resource High. https://docs.aws.amazon.com/codeguru/detector-library/java/aws-polling-instead-of-waiter/.

[6] 2022. CodeGuru Rule: Missing pagination. https://docs.aws.amazon.com/codeguru/detector-library/java/missing-pagination/.

[7] 2022. CodeGuru Rule: Use of a deprecated method. https://docs.aws.amazon.com/codeguru/detector-library/java/deprecated-method/.

[8] Amazon Web Services. [n. d.]. AWS SDK for Python (Boto3). https://aws.amazon.com/sdk-for-python/

[9] Amazon Web Services. [n. d.]. Boto3 - The AWS SDK for Java. https://github.com/aws/aws-sdk-java

[10] Amazon Web Services. [n. d.]. What is Amazon CodeGuru Reviewer? https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html

[11] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, 288–298. https://doi.org/10.1145/2568225.2568243

[12] Cristiano Calcagno and Dino Distefano. 2011. Infer: An Automatic Program Verifier for Memory Safety of C Programs. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6617)*, Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, 459–465. https://doi.org/10.1007/978-3-642-20398-5_33

[13] Rong Gu, Zhiqiang Zuo, Xi Jiang, Han Yin, Zhaokang Wang, Linzhang Wang, Xuandong Li, and Yihua Huang. 2021. Towards Efficient Large-Scale Interprocedural Program Static Analysis on Distributed Data-Parallel Computation. *IEEE Trans. Parallel Distributed Syst.* 32, 4 (2021), 867–883. https://doi.org/10.1109/TPDS.2020.3036190

[14] Ben Hardekopf, Ben Wiedermann, Berkeley R. Churchill, and Vineeth Kashyap. 2014. Widening for Control-Flow. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8318)*, Kenneth L. McMillan and Xavier Rival (Eds.). Springer, 472–491. https://doi.org/10.1007/978-3-642-54013-4_26

[15] Vineeth Kashyap, Kyle Dewey, Ethan A. Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: a static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 121–132. https://doi.org/10.1145/2635868.2635904

[16] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. 2015. Practically Tunable Static Analysis Framework for Large-Scale JavaScript Applications (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 541–551. https://doi.org/10.1109/ASE.2015.28

[17] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. 2016. Accelerating program analyses by cross-program training. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*, Eelco Visser and Yannis Smaragdakis (Eds.). ACM, 359–377. https://doi.org/10.1145/2983990.2984023

[18] Yingjun Lyu, Sasha Volokh, William G. J. Halfond, and Omer Tripp. 2021. SAND: a static analysis approach for detecting SQL antipatterns. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 270–282. https://doi.org/10.1145/3460319.3464818

[19] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 554–564. https://doi.org/10.1145/2491411.2501854

[20] Rashmi Mudduluru and Murali Krishna Ramanathan. 2014. Efficient Incremental Static Analysis Using Path Abstraction. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8411)*, Stefania Gnesi and Arend Rensink (Eds.). Springer, 125–139. https://doi.org/10.1007/978-3-642-54804-8_9

[21] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. 2022. Static Analysis for AWS Best Practices in Python Code. In *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany (LIPIcs, Vol. 222)*, Karim Ali and Jan Vitek (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:28. https://doi.org/10.4230/LIPIcs.ECOOP.2022.14

[22] Lori L. Pollock and Mary Lou Soffa. 1989. An Incremental Version of Iterative Data Flow Analysis. *IEEE Trans. Software Eng.* 15, 12 (1989), 1537–1549. https://doi.org/10.1109/32.58766

[23] Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. 2019. Know your analysis: how instrumentation aids understanding static analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, Neville Grech and Thierry Lavoie (Eds.). ACM, 8–13. https://doi.org/10.1145/3315568.3329965

[24] Amazon Web Services. 2023. CodeGuru Rules. https://docs.aws.amazon.com/codeguru/detector-library/.

[25] Amie L. Souter and Lori L. Pollock. 2001. Incremental Call Graph Reanalysis for Object-Oriented Software Maintenance. In *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001*. IEEE Computer Society, 682–691. https://doi.org/10.1109/ICSM.2001.972787

[26] John Toman and Dan Grossman. 2017. Taming the Static Analysis Beast. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 18:1–18:14. https://doi.org/10.4230/LIPIcs.SNAPL.2017.18

Received 2023-03-10; accepted 2023-04-21