User-Assisted Code Query Customization And Optimization

Ben Liblit[1], Yingjun Lyu[1], Rajdeep Mukherjee[1], Omer Tripp[1], Yanjun Wang[1]

[1]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 data-flow specification*, 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, 2× speedup and 51% increase on the number of findings for caching-based optimization. Customizing the data-flow specification, such as expanding the tracking scope, can yield a remarkable increase in the number of findings, as much as 136%. However, this enhancement comes at the expense of a longer analysis time. Our evaluations emphasize the importance of customizing the data-flow specification, particularly when users operate under time constraints. This customization helps the analysis complete within the given time frame, ultimately leading to improved recall.

Keywords: Program analysis, Query Language, Data-flow, Taint-flow, AWS, Security

1 Problem Setting

Amazon CodeGuru Reviewer [9] is a commercial product that performs source-code repository scans. CodeGuru Reviewer also integrates into the code review process as an automated reviewer, leaving comments on pull requests. It is based primarily on "micro-analyzers", which run narrow yet precise analysis scenarios. These are built atop the Guru Query Language (*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 "democratizes" CodeGuru Reviewer by empowering domain experts to directly specify, then tune and productionize, micro-analyzers. GQL provides the building blocks for such micro-analyzers, which the expert then composes to express a property of interest, such as 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 GQL, which are made available to rule authors to tune their rules' performance and resource consumption.

Along with these enhanced control facilities over rule performance and behavior comes great

responsibility. As an example, providing a fine-grained specification for a data-flow problem requires careful vetting to avoid unintended precision or recall gaps. The advantage of our proposed approach is that optimizations made by the rule author are explicit, inlined into the rule context, and directly available to review and tune.

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, JavaScript, Python, and TypeScript. It 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.

The main impact of the different modes on the analysis algorithm is in deciding the target scope and evaluation budget. The "full" and "CI/CD" modes do not restrict the analysis scope, while "incremental" mode applies the rules only to files that contain code changes. However, rules can still pull in additional context through inter-procedural constructs, such as forward or backward slicing. The "full" mode imposes no rule evaluation budget, but the "incremental" and "CI/CD" modes require that analyses terminate within a designated number of steps or time units.

In all modes, CodeGuru leverages a caching layer, such that the intermediate representation can be fetched for graphs and files that have not been invalidated by code changes. As explained above, the "incremental" mode builds on the diffs computed at text level to determine the files that require re-evaluation.

Combined, these different options enable situation-appropriate analysis across the development life cycle: developers obtain feedback on their code changes during code review; then additional issues may be surfaced as the code is integrated through the pipeline; and lastly, exhaustive analysis is executed at regular intervals (e.g., nightly), which may expose yet additional issues.

2.1 Intermediate Graph Representation

CodeGuru's intermediate representation is the *MU graph* [22]. A MU graph is a data-dependence graph overlaid with a control-flow graph, all in static single assignment (*SSA*) form. An individual MU graph node represents 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 has:

- one data edge from some data node representing the callee;
- 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 correspond to primitive language operations, with analogous operations across multiple languages sharing the same representation. For example, "`a / b`" yields a binary arithmetic division action regardless of whether the source code was Java, JavaScript, Python, or TypeScript. Unnamed temporary values are made explicit, as is the order of operations. For example, the representation of `print(a + b)` includes a sum action node followed by 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: the data structure is complex and traversal-intensive, making it time-consuming for an analyst to code up exactly the right set of checks to detect a pattern (or

anti-pattern) of interest. To make this task easier, CodeGuru includes GQL, a domain-specific language for operating on MU graphs [22]. 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 leverages 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 detect bugs.

3 Motivating Example

To illustrate the insights feeding into the optimizations described in this paper, and the benefits that these optimizations introduce, consider the code example in figure 1. This example is inspired by real-world code that our rules were evaluated on. 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. The core of this rule performs an inter-procedural taint analysis, via the operation withDataDependentsTransform, wrapped inside an operation called withInterproceduralMatch. These two operations together take the sources as the match frontiers, and return the inter-procedural nodes tainted by the frontiers. The

```
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   }...}
```

Fig. 1 Code snippet demonstrating injection vulnerabilities

remaining operations and their parameters denote the various kinds of optimization and customization that users can make via GQL, such as using withCachedDependency to cache the forward flows of various taint sources, using withRuleConfigurationItemMatchFilter to perform configuration based indexing that dynamically indexes into a matching taint configuration, and using the parameters of withDataDependentsTransform and withInterproceduralMatch to adjust the tracking and analysis scope.

In the discussion that follows we further elaborate on the rule in figure 2 and these various operations using the code example in figure 1.

3.1 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 use

```
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        .withInterproceduralMatch(
8            TrackingScope.FILE,
9            b -> b.withDataDependentsTransform(
10               Arrays.asList(/* passthrough = */
11                   "java.lang.String.format",
12                   "java.lang.String.toLowerCase",
13                   "java.lang.String.toUpperCase"),
14               Arrays.asList(/* blocking = */
15                   "boolean.!=", "boolean.=="),
16               Arrays.asList(/* sideEffecting = */ ".*\\.set.*"),
17               Arrays.asList(/* reading = */ ".*\\.get.*")
18           ))
19     ...
20     .build();
```

Fig. 2 Rule snippet demonstrating caching and configuration indexing in the context of a custom data-flow specification

the same sources, validators, and sanitizers. In this case, reuse enables amortization across the path-traversal and command-injection rules.

Note that this mechanism differs from memoization in data-flow frameworks like IFDS [24], where data-flow edges are reused when solving a given data-flow problem. In this case, state is persisted across independent tasks. The decision as to which state to persist is entrusted to the user, and the system provides performance metrics on cache hits versus misses.

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.

GQL rule authors need to identify rules sharing the same source specification, to recognize candidates for caching. However, GQL rule authors do not need to think about the execution order of the rules where the rules share the source specification. For a given set of sources, validators, and sanitizers, the cached content will be the same no matter which rule populates the cache by running first.

3.2 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 have tens of thousands of entries, which makes efficient handling essential. 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.

3.3 Data-flow Customization

In GQL, different forms of data-flow analysis, including e.g. forward/backward taint tracking and slicing, can be composed with other constructs. In addition, the data-flow problem can be specified with a high degree of user control over the target scope (local versus same file versus entire code package, and so on).

We illustrate the main specification dimensions through the withInterproceduralMatch compound statement at line 7 in figure 2. In particular, the scope is set as FILE-wide at line 8, and forward tracking is instantiated at line 9. Arguments consist of "pass-through" actions that admit input/output data-flow, followed by actions that truncate the flow, then methods that induce a side effect on their receiver, and finally methods that read from their receiver's internal state.

This fine level of user control allows the rule author to tune their rule to maximize return on investment. Increasing the tracking scope, and enabling more propagation by enhancing the pass-through, side-effecting and reading specifications, results in more detections but potentially also slower evaluation, more resource consumption, and a higher rate of false detections. Restricting the data-flow algorithm has the opposite effects.

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 detail 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 inductively. 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.

The reason behind rule budget adjustment even in cases where the rule "benefits" from the results cached thanks to another rule is to ensure consistent and reproducible results for the rule suite as a whole. The rules are evaluated concurrently, typically as a large suite of hundreds of rules with a high degree of parallelism, which means that in different runs different rules will trigger the caching mechanism. If budgets are not aligned appropriately, then rule budgets will fluctuate across runs, potentially leading to different findings.

If sr_1 and sr_2 are isomorphic and have compatible analysis budgets b_1 and b_2, then 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, caching 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 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 the same matching frontier is embedded inside multiple rules. 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 amortization need not be explicitly coordinated across the rules. Rather, each need only wrap its 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. Rule authors should use memory frugally to cache the expensive operations to optimize the performance gains of withCachedDependency. There are two recommended ways to carry out the caching optimization. First, when the rule authors use expensive GQL constructs, we recommend that they look for opportunities of caching by identifying the common subrules in other existing rules that already use withCachedDependency. Second, while the GQL rules are modular, i.e., there is not a single developer who writes all GQL queries, changes can be made to a subset of the existing rules to leverage caching in a holistic and consistent manner.

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. As noted in section 3.2, these configurations can contain tens of thousands of entries. 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 approach leverages 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 $c \mapsto k$; and
- a mapping function τ from entities in the code to the same domain of keys plus $\perp$ (for configuration-irrelevant entities).

Returning 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, good indexing and mapping is easy to achieve, since we typically make use of types and identifiers. The indexing and mapping functions are both cheap to compute and yield low collision rates.

4.3 Fine-grained Data-flow Specification

Among the most powerful features of semantic code analysis is the ability to track data flow across the application. Notable use cases include security (use of untrusted user input); privacy (release of private information); and code transformation/optimization (e.g. parallelization of a loop structure if its iterations are data independent).

Data-flow analysis is governed by a multi-faceted specification, sometimes implicit in the tracking algorithm, that can significantly impact accuracy and efficiency. Instead of pre-determining every single aspect of the internal mechanics the data-flow analysis, GQL makes the data-flow analysis configurable as follows.

- *Tracking scope*: Either the entire package (or code repository), or a more granular code context, which enables higher efficiency and accuracy. These include the file or type containing the function. The tracking scope allows GQL rule authors to make trade-offs based on their needs for accuracy and efficiency. For example, when the time budget is limited, a file-level tracking scope can be chosen to favor efficiency over recall, thereby increasing the likelihood that rules finish before timing out.
- *Pass-through actions*: Functions and operators that admit input/output data flow, such as string concatenation.
- *Blocking actions*: The opposite of pass-through actions, where data flow is truncated across evaluation of such functions and operators.
- *Side-effecting actions*: These mutate the receiver object and/or other invocation arguments. An example is adding an element to a collection, where the collection itself should be

tracked subsequently with the added element forming part of the collection's internal state.
- *Reading actions*: The complement of side-effecting actions, where what is read from internal state should be tracked by the data-flow analysis algorithm (e.g., accessing an element in a tracked collection).

Note that all of the above specification dimensions are orthogonal to the design of the core data-flow analysis algorithm, and whether or not it is field sensitive, object sensitive, flow sensitive, or so on.

The specification is consumed by GQL's core data-flow analysis engine, which supports both forward and backward propagation. The analysis scope is determined according to the user specification, where if the scope falls below the entire package, then we eliminate functions from consideration if they reside outside the type or file stated by the user.

As is standard, the analysis algorithm performs forward slicing from sources (*resp.* backward slicing from sinks), then searches for sinks (*resp.* sources) in the resulting slice. In scenarios where source/sink reachability is established, a witness generation algorithm is invoked to compute a source/sink path by retracing backwards from sinks (*resp.* forward from sources) through data-flow transitions within the slices. In cases where caching is further used, as with the rule in figure 2, the slices are of course reused across different analysis rules with compatible tracking configurations: a significant performance optimization.

- A pass-through specification matches against actions, and enables forward (*resp.* backward) propagation across the action. This is particularly helpful in scenarios where the action is a function coming from a library, thus not amenable to direct analysis. As an example, a pass-through action gives rise to a definition-to-arguments flow in the backwards direction, and an argument-to-definition flow in the dual case of forward propagation.
- A blocking specification applies in the same manner, though with the opposite effect of terminating rather than enabling data propagation when the data-flow engine is able to match the specification against an action in the code. A more advanced scenario, which we illustrate at line 14 in figure 2, is data-flow truncation through a validation action (specifically, (in)equality testing). Unlike sanitization, which has a "local" effect, validation draws on the notion of control dependence. To enforce this variant of blocking actions, we compute the control dependence graph of every visited function that contains actions matching the blocking specification. We then check whether data-flow transitions intersect with control dependence edges, in which case we abort propagation.
- A side-effecting specification applies in scenarios where an action is matched (e.g., List::set), where in the forward (*resp.* backward) direction, the argument (*resp.* receiver) is tracked thus triggering tracking of the receiver (*resp.* argument). Note that there is built-in handling for field assignments. The value of the specification is in scenarios where side effects take place due to a function call (in particular, when using library types like java.util.Collection).
- Finally, the reading specification complements the side-effecting specification, reading from tracked "memory" (e.g., via List::get). In the forward (*resp.* backward) direction, an action applied to a tracked receiver (*resp.* definition) triggers tracking of the definition (*resp.* receiver). Again here this enables more complete treatment of data flow in scenarios that involve library types.

When rule authors determine the specification dimensions, they are making trade-offs between precision, recall, and efficiency of the rules. We discuss the impact of the various types of configurations further in section 5. In practice, rule authors continuously monitor rule performance against a large number of codebases. They define common values of the data-flow specification that are proven to meet the requirements of their rules or campaigns, and often also share the data-flow specification across different rules. The specification is continuously adjusted as rules are refined and optimized.

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 four 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 [1] or AWS Python SDK [3]. The second dataset was used to evaluate different caching strategies and data-flow specifications. 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 25,697.

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 to assess the impact of the configuration indexing, and 7 taint-flow rules to assess the effects of caching and customizing the data-flow specification. 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 long time limit, while an online scanning during a code review typically demands speed. We evaluated our rules using 30- and 5-minute time limits that correspond to limits commonly enforced for other analyzers—running internally in our organization—in compatible scenarios.

5.3 Experiment 1: Configuration Indexing

Many AWS operations return paginated results when the response object is too large to return in a single response. One best practice for using operations which may return paginated results is checking a specific object in the response to verify that all results are returned. One of the configurations for the "Missing Pagination" trait is described in the JSON fragment in figure 3. This snippet describes list_dataset_groups as a function that returns paginated results, and provides

```
{
  "expectedPaginationMethods": [
    "IsTruncated",
    "NextToken"
  ],
  "paginatedMethod": "list_dataset_groups",
  "resultKeys": [
    "DatasetGroups"
  ],
  "serviceId": "forecast"
}
```

Fig. 3 Partial configuration for the "Missing Pagination" trait

additional details as to which objects should be checked, the data type of returned results, and the associated AWS service. These API specifications are automatically mined from corresponding API models, for example, Boto3 (AWS Python SDK) [2].

Table 1 presents the impact of indexing based configuration using five CodeGuru rules [28], that reflect guidelines for correct, secure, and performant usage of AWS cloud Java and Python SDKs. These rules operate with hundreds and thousands of AWS API configurations derived from hundreds of public AWS services.

All CodeGuru rules are documented online, in the CodeGuru Detector Library [28]. For brevity, we present few representative rules in table 1, which are briefly described below.

- *Rule 1: Misuse of paginated APIs:* The pagination API is used when the result set due to a query is too large to fit within a single response. The best practice is to use pagination token to perform iterative requests and receive the response in parts.
- *Rule 2: Error handling for batch operations:* Batch operations can succeed without throwing an exception even if processing fails for some items. Therefore, a best practice is to explicitly check for failures in the response due to the batch API call.
- *Rule 3: Use waiters in place of polling API:* Waiters are utility methods that make it easy to wait for a resource to transition into a desired state by abstracting out the polling logic into a simple API call. Our rule detects code that appears to be polling for a resource

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 [4]	1,117	Timeout / Timeout	N/A	215.3s / 215.3s	78
Rule 2 [5]	8,411	Timeout / Timeout	N/A	296.7s / 296.7s	136
Rule 3 [6]	81	427.7s / Timeout	32	144.5s / 144.5s	32
Rule 4 [7]	186	694.5s / Timeout	56	139.4s / 139.4s	56
Rule 5 [8]	126	673.1s / Timeout	47	126.4s / 126.4s	47

before it runs. In such cases, it recommends using efficient waiters instead.

- *Rule 4: Detect uncaught exceptions for AWS APIs:* Detect uncaught exceptions involving AWS APIs and recommend handling those uncaught exceptions as well.
- *Rule 5:* Detect usage of deprecated APIs in AWS 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.

By contrast, the optimized rules take the same amount of time for different time limits (30 minutes versus 5 minutes), regardless of configuration sizes. This clearly demonstrates the positive impact of configuration indexing. For rules that evaluate on few hundred configurations, such as *Rules 3–5*, the speedup is 3× or more. The total number of findings depends on the the extent of usage of the AWS SDKs in the dataset under analysis. Furthermore, the number of findings (reported as "# Findings") show that the unoptimized *Rules 1* and *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 bugs overall, and improves the recall of these rules.

5.4 Experiment 2: Caching and Data-flow Specification

This experiment evaluated the impact of caching and data-flow specification. 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 [28]. All the rules shared the same set of tainted sources. The vast majority of these represent untrusted data coming from the Internet. 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 ran these rules against 180 repositories in the second dataset with 5- and 30-minute time limits. We used various combinations of analysis scopes (i.e., file-level and package-level), caching configurations (i.e., with and without caching), and data-flow actions on these rules. We designed the experiment in this way because in practice, rule developers are given different time budgets of rule execution. They tune their data-flow specifications, making trade-offs between precision, recall, and efficiency, to stay within their assigned time budgets. GQL itself does decide which dimension is more important; this is for rule authors to decide. Based on the scanning requirement, there exist scenarios where the time limit is longer, and recall is more important. There also exist scenarios where efficiency and precision are favored over recall. This experiment give GQL users more insights with heuristic data about the practical gains and losses when tuning a data-flow specification.

In particular, we compared two typical combinations of data-flow actions used by rule authors in real world. The first variant is conservative and

Table 2 Comparison for rules with and without caching using different data-flow specifications with time limit of 5 minutes.

		Cache			Analysis Time (in seconds)					
Scope	Actions	Config.	Size	# Hits/Misses	All	First	Rest	Median	# Findings	# TOs
File	Cons.	Disabled	N/A	N/A	2,796.4	443.0	2,353.4	2.7	314	1
File	Cons.	Enabled	10,000	39,800/6,653	2,730.9	442.0	2,288.9	2.6	314	1
File	Cons.	Enabled	100,000	39,800/6,653	2,725.8	440.3	2,285.5	2.7	314	1
Pkg	Cons.	Disabled	N/A	N/A	4,327.6	1,121.3	3,206.3	3.7	343	17
Pkg	Cons.	Enabled	10,000	20,565/5,597	2,679.9	1,138.5	1,541.4	3.0	406	12
Pkg	Cons.	Enabled	100,000	21,705/4,651	2,448.8	1,134.5	1,314.3	2.9	495	11
File	Perm.	Disabled	N/A	N/A	3,540.8	642.4	2,898.4	2.2	401	4
File	Perm.	Enabled	10,000	13,868/31,506	3,493.7	643.2	2,850.5	2.1	401	4
File	Perm.	Enabled	100,000	39,723/6,645	3173.7	644.1	2529.6	2.1	711	3
Pkg	Perm.	Disabled	N/A	N/A	4,131.4	784.8	3,346.6	3.0	472	28
Pkg	Perm.	Enabled	10,000	7,204/12,651	3,698.6	790.7	2,907.9	2.4	470	28
Pkg	Perm.	Enabled	100,000	17,369/3,422	1,840.3	790.9	1,049.4	2.4	607	23

Table 3 Comparison for rules with and without caching using different data-flow specifications with a time limit of 30 minutes.

		Cache			Analysis Time (in seconds)					
Scope	Actions	Config.	Size	# Hits/Misses	All	First	Rest	Median	# Findings	# TOs
File	Cons.	Disabled	N/A	N/A	2,959.9	455.5	2,504.4	2.8	339	0
File	Cons.	Enabled	10,000	41,804/6,965	2,890.2	454.8	2,435.4	2.7	339	0
File	Cons.	Enabled	100,000	41,804/6,965	2,899.3	457.4	2,441.9	2.7	339	0
Pkg	Cons.	Disabled	N/A	N/A	16,172.9	6,033.7	10,139.2	3.8	529	10
Pkg	Cons.	Enabled	10,000	28,798/9,935	10,672.2	5,992.3	4,679.9	3.0	558	8
Pkg	Cons.	Enabled	100,000	38,210/6,471	8,059.8	5,945.6	2,114.2	2.9	799	3
File	Perm.	Disabled	N/A	N/A	4,288.3	652.8	3,635.5	2.2	764	0
File	Perm.	Enabled	10,000	14,024/34,745	4,286.3	655.9	3,630.4	2.1	764	0
File	Perm.	Enabled	100,000	41,804/6,965	3,459.3	650.6	2,808.7	2.1	764	0
Pkg	Perm.	Disabled	N/A	N/A	9,906.1	3,412.3	6,493.8	2.9	617	22
Pkg	Perm.	Enabled	10,000	7,425/16,911	8,647.4	3,540	5,107.4	2.4	652	21
Pkg	Perm.	Enabled	100,000	25,501/4,633	4,709.4	3,399	1,310.4	2.4	786	18

targets high precision. It uses a small set of pass-through actions, such as string concatenation, and reading actions, such as getters of tainted sources. Rule authors did not need to specify the blocking actions because only a limited set of actions was allowed to admit data flow. Rule authors also did not specify any side-effecting actions to prevent imprecision introduced by data flowing in and out of collections. Using this combination of data-flow actions, a finding from the rule indicates with high confidence that untrusted user input can direct control the sink. For example, if a rule targets command injection, then the untrusted input will be able to append additional malicious commands to the OS command execution API.

The second variant is more permissive than the first. The consumers of findings generated by these permissive actions are more tolerant to imprecise results in exchange of a higher recall. The pass-through actions in this combination included any function that is not defined internally in the analyzed program. The side-effecting actions and reading actions included common read/write operators against collections. For example, adding an element to and reading an element from a list. As for the blocking actions, the specification defined a set of approved sanitizers. Using this set of permissive actions, a finding from the rules indicates that untrusted user input is able to reach the sink without approved sanitization and validation against the input.

Tables 2 and 3 show the impacts of scope customization, caching, and data-flow actions. The results in these two tables were based on rules running on the benchmarks using 5- and 30-minute time limits, respectively. In these two tables, 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. Column 2 indicates whether the data-flow actions used in the experiments are conservative ("Cons.") or permissive ("Perm."). Columns 3–4 list the caching configuration we set for each rule. We experimented with

different cache sizes, which specify the maximum number of mappings from tainted sources to cached tainted program points. Column 5 presents the number of cache hits and misses for each experiment. In columns 6–9, 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 median evaluation time required for analyzing a repository. Note that the analysis times we report in these columns do not include time spent on repositories where the analysis timed out. 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 for the given time limit.

5.4.1 Impact of Caching

Our evaluation indicates that caching can have a substantial impact on the analysis speed, resulting in fewer timeouts and better recall. As cache size increases, the improvements become more pronounced due to more frequent cache hits. In addition to memory considerations, the time limit allocated by users for the analysis also positively influences the benefits of caching.

Overall, caching is more effective on package-level analysis than file-level analysis. For example, rows 4 and 6 (respectively rows 1 and 3) of table 2 display the results where the rules conducted package-level (respectively file-level) analysis with and without caching at a time limit of 5 minutes. Speedup of rule execution for file-level analysis was negligible with no extra findings with caching. The speedup of overall rule execution for package-level analysis was more than 1.7× relative to no caching. When examining the breakdown of the analysis time for package-level analysis, we observed only a slight increase in the analysis time for the first rule, as indicated in the "First" column. This increase is likely due to the overhead from cache writes. Such overhead was well offset by later-on savings when the rest of the rules were executed. The analysis time of the remaining six rules (displayed at the column "Rest") only took 15% more analysis time than the first rule alone. This is because those six rules can leverage the tainted program points cached by the first rule execution. This performance improvement from caching 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×, as shown in rows 4 and 6 of table 3. The number of findings increased by 51% from 529 to 799. A longer time limit helped because the first rule had its chance to finish on more complex repositories and wrote to the cache.

If users larger memory budgets, our evaluation shows that larger caches are beneficial. In both tables 2 and 3, except for the conservative file-level analysis, we see the positive impact of larger caches on analysis time, the number of timeouts, and the number of findings. For example, rows 5 and 6 of table 2 show that a larger cache slightly increases the number of cache hits and reduces the number of cache misses for conservative package-level analysis, resulting in a speedup of 1.1×, 1 fewer timeout, and 89 (22%) more findings. The benefits of larger caches were more significant when the time limit was 30 minutes. Rows 5 and 6 of table 3 show that the larger cache increased the number of cache hits of the conservative package-level analysis by 1.3×, resulting in a speed up of 1.3×, 5 fewer timeout, and 241 (43%) more findings. We see the same trend for analyses with permissive data-flow actions. As shown in rows 8 and 9 of table 2, and rows 11 and 12 table 3, the number of findings from a permissive file-level or package level analysis both increase, and the number of timeouts both decrease. Even in cases where the analysis completed without timeouts, a larger cache improved analysis time by 19%, primarily attributable to a significant increase in cache hits, as shown in rows 8 and 9 of table 3. Due to the complex taint flows in these repositories, a larger cache was needed otherwise the cache could only hold a subset of the tainted program points. Once the required resources on both time and memory are met, users can maximize the benefits of caching.

The only case where we did not observe benefits from caching was conservative file-level analysis. Caching had no observable positive or negative impact. We obtained the same number of findings and the same number of timeouts. The analysis time was also comparable with and without caching. This is due to fact that conservative file-level analysis is contained and fast. Caching inherently adds some overhead and therefore did not benefit file level analysis overall.

5.4.2 Impact of Scope Customization

Our evaluation clarifies the advantages of customizing the tracking scope. Analyses using a file-level tracking scope ran faster compared to those with a package-level scope. The file-level scope helped the analyses complete within the given time limit. Conversely, analyses using a package-level tracking scope experienced more frequent timeouts, but they demonstrated notable improvements in recall across most scenarios.

File-level analysis outperformed package-level analysis in meeting the time limit. This observation holds for both the conservative and permissive data-flow actions, with or without caching. The "# TOs" columns in tables 2 and 3 show that rule execution rarely times out when a file-level tracking scope is in use. For example, the file-level analysis timed out on only 1 repository at the time limit of 5 minutes, while the package-level analysis timed out on 11 to 17 repositories, depending on the caching configuration. At the time limit of 30 minutes, the file-level analysis did not time out. The reduction in timeouts was due to the fact that file-level analysis needed to reason about a smaller call graph and therefore a smaller set of tainted program points, which ultimately led to faster analysis and fewer timeouts. For example, compared to conservative package-level analysis, total rule evaluation time of conservative file-level analysis sped up by more than 1.5×, as shown in rows 1 and row 4 of table 2. (Note that the comparison of rule evaluation time only took into account the analyses that did not time out.)

In most scenarios, especially when conservative data-flow actions are in use, using a package-level tracking scope would improve the recall. The improvement is more significant when caching is in place and when the time limit is longer. At a time limit of 5 minutes, the file-level analysis produced 314 findings regardless of the caching configuration, while the package-level analysis produced 343 findings (9% more) without caching, 406 findings (29% more) with a 10,000-item cache, and 495 findings (58%) with a 100,000-item cache. At a time limit of 30 minutes, the file-level analysis produced 339 findings, while the package-level analysis produced 529 findings (56%), 558 findings (65% more), and 799 findings (136% more), respectively. Since the package-level analysis was more expensive and needed to compute more tainted program points, caching can help to cut the cost and a longer time limit allowed the analysis to run longer, both of which enhanced the benefits of using a package-level tracking scope.

However, if the analysis used a combination of rule configuration that made it hard to scale properly, leading to many timeouts, then the improvement on recall using a package-level tracking scope would be minimal. In some extreme scenarios, the number of findings even dropped. If the time limit is short and caching is unavailable, analysis with a package-level tracking scope timed out on 17 repositories while the analysis with a file-level tracking scope only timed out on 1 repository. The former only produced 29 more findings (9% more) than the latter. When the analysis used permissive data-flow actions together with a package-level tracking scope, the number of findings dropped in some cases. We discuss this result further in section 5.4.3.

5.4.3 Impact of Data-flow Actions

Our experiments show that given the same caching configuration, analysis scope, and time limit, rules with permissive data-flow actions lead to more findings most of the time, compared to rules with conservative actions, but at the cost of more timeouts.

Results show that permissive data-flow actions can significantly improve the recall of the analysis with a file-level tracking scope. Rows 1 and 7 of table 2 show that the permissive analysis generated 401 findings: 87 (28%) more than the 314 findings generated by the conservative analysis. When the time limit is 30 minutes, the permissive analysis generated 764 findings: 125% more than the 339 findings generated by the conservative analysis. We randomly sampled 79 additional findings found by the permissive analysis and manually examined the source code. Although the analysis did generate more false-positives, it produced more valid findings as well (41% of the additional findings produced by the permissive actions were true-positives), effectively improving the recall. The conservative analysis was not able to find them because its data-flow actions did not capture the pass-through actions that represent APIs from libraries in most of the time, such as JSON-parsing libraries and URI-building libraries, or even sometimes internal methods that

are defined in the same package but in a different file. These APIs and method calls can admit data flow, and need to be explicitly specified as a part of the pass-through actions. Among the rest of the 79 additional findings, 45 (57%) were false-positives. In 2 cases our manual analysis could not draw a conclusion because the tainted data flowed into a method whose implementation was neither in the same repository nor from common libraries. When rules are applied on real world code bases where there are numerous APIs from third-party libraries that can admit data-flow, a permissive pass-through action that assumes that any unknown function can admit data-flow could indeed be one of the viable options to improve recall. However, we also observed that overly permissive pass-through actions made the analysis to make incorrect assumptions about data-flow, contributing to the vast majority of false alarms.

While analyses with permissive actions typically produced more findings, those using conservative actions consistently outperformed the permissive ones when it came to meeting time limits. Surprisingly, using conservative actions not only reduced timeouts but, in certain combinations of specifications within our experiment, also resulted in more findings than the permissive actions. At the package level when the time limit was 30 minutes and a large cache was in use, the conservative analysis generated 799 findings, 1.6% more than the 786 findings generated by the permissive analysis, as shown in rows 6 and 12 of table 3. This was due to the fact that the permissive analysis had 18 timeouts: 6× more than the conservative analysis, eventually resulting in fewer findings.

In general, we observed that the negative impact of more frequent timeouts, caused by the permissive actions, becomes evident in the package-level analyses, potentially offsetting the benefits of expanding the analysis scope from file-level to package-level. When the analysis used conservative actions, changing the analysis scope from file-level to package-level can always lead to more findings, as discussed in section 5.4.2. However, we noticed that the permissive package-level analysis could produce fewer findings than the permissive file-level analysis. For example, given a short time limit of 5 minutes, rows 9 and 12 of table 2 show that the permissive package-level analysis had 7.7× more timeouts than the permissive file-level analysis and 15% fewer findings. Similarly, given a longer time limit, the number of findings from package-level analysis at row 10 and 11 of table 3 is smaller than the number of findings coming from the corresponding file-level analysis at row 7 and row 8 in the same table. This negative impact was caused by the fact that the permissive pass-through/side-effecting/reading actions would produce a large program slice. The more tainted program points in this slice, the longer it took for the data-flow analysis to finish. In package-level analysis, overly permissive actions could cause the program slice to rapidly expand, leading to poor scalability and frequent timeouts in the overall analysis. Caching was helpful in alleviating the situation as it reduced the number of timeouts. However, the overall number of timeouts after the improvement from caching remained large.

Based on all the above observations, we determine that there are two takeaways for rule authors to optimize their rule configuration and execution.

1. **Avoid using overly permissive actions for a package-level analysis.** When package-level analysis is needed, instead of allowing any unknown functions to pass-through tainted data, rule authors should specify a more contained set of functions that can pass the tainted data through. This will slow the expansion of the program slice during the data-flow analysis, resulting in faster rule evaluation. To identify the required pass-through actions, rule authors can run the rules with permissive actions and conservative actions against test repositories in an experimental mode with a long time limit. After manually reviewing the additional findings from the permissive version of the rules, they should identify the APIs, e.g., from a JSON-parsing library, that can admit data-flow can add those to the pass-through actions. In this way, the recall of the rules increases and the precision remains high.
2. If rule authors do not want to spend time manually identifying accurate data-flow actions, but they still want to maximize recall, then they can **use both conservative and permissive actions in a greedy manner.** Given a certain time limit, the rules with conservative actions should be executed first. If these rules complete their analysis of the repository before the time limit is reached,

any remaining time can be spent executing the permissive version.

6 Related Work

Toman and Grossman [30] note caching as a widely used technique to make static analysis tractable [10, 12, 19, 20, 21, 23, 29], but limited to reanalysis of the same program or analysis of shared library code [18]. Prior work on analysis caching has generally keyed the cache on coarse-grained program components, such as functions or files. By contrast, we 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 [28], 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. [14], who focus on scaling any single analysis to large programs.

Unlike other rule-based static analysis languages such as, CodeQL [13], 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 offers codebase-wide data-flow and type-state capabilities, that is, deeper and more semantic analysis, unlike tools such as Semgrep [27].

Arzt et al. [11] present *FlowDroid*, which is a novel and highly precise static taint analysis for Android applications using context, flow, field and object-sensitivity to reduce the number of false alarms. Schubert et al. [26] present *Varalyzer* that performs effective static data-flow analysis of software product lines on real-world C code which allow developers to find bugs and vulnerabilities much earlier in the development process.

Toman and Grossman [30] 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. [25] discuss the importance of understanding analysis performance so that it can be tuned to perform well. Toman and Grossman [30] also note the use of tunable "knobs" to balance precision and performance [15, 16, 17]. 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. [25] 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 and data-flow actions of inter-procedural 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, 2× speedup thanks to caching, and 1.5× thanks to fine-grained data-flow specification.

References

[1] Amazon Web Services. Boto3 - the AWS SDK for Java, 2022. URL https://github.com/aws/aws-sdk-java.

[2] Amazon Web Services. Boto3 - the AWS SDK for Python, 2022. URL https://github.com/boto/boto3.

[3] Amazon Web Services. AWS SDK for Python (Boto3), 2022. URL https://aws.amazon.com/sdk-for-python/.

[4] Amazon Web Services. Missing pagination rule. https://docs.aws.amazon.com/codeguru/detector-library/java/missing-pagination/, 2022.

[5] Amazon Web Services. Batch request with unchecked failures rule. https://docs.aws.amazon.com/codeguru/detector-library/java/aws-unchecked-batch-failures/, 2022.

[6] Amazon Web Services. Inefficient polling of aws resource high rule. https://docs.aws.amazon.com/codeguru/detector-library/java/aws-polling-instead-of-waiter/, 2022.

[7] Amazon Web Services. Check uncaught exceptions high rule. https://docs.aws.

amazon.com/codeguru/detector-library/java/check-uncaught-exceptions/, 2022.

[8] Amazon Web Services. Use of a deprecated method rule. https://docs.aws.amazon.com/codeguru/detector-library/java/deprecated-method/, 2022.

[9] Amazon Web Services. What is Amazon CodeGuru Reviewer?, 2023. URL https://docs.aws.amazon.com/codeguru/latest/reviewer-ug/welcome.html.

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

[11] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 259–269. ACM, 2014. doi: 10.1145/2594291.2594299. URL https://doi.org/10.1145/2594291.2594299.

[12] Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 459–465. Springer, 2011. doi: 10.1007/978-3-642-20398-5_33. URL https://doi.org/10.1007/978-3-642-20398-5_33.

[13] GitHub, Inc. Codeql, 2019. URL https://codeql.github.com.

[14] Rong Gu, Zhiqiang Zuo, Xi Jiang, Han Yin, Zhaokang Wang, Linzhang Wang, Xuandong Li, and Yihua Huang. Towards efficient large-scale interprocedural program static analysis on distributed data-parallel computation. *IEEE Trans. Parallel Distributed Syst.*, 32(4):867–883, 2021. doi: 10.1109/TPDS.2020.3036190. URL https://doi.org/10.1109/TPDS.2020.3036190.

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

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

[17] Yoonseok Ko, Hongki Lee, Julian Dolby, and Sukyoung Ryu. Practically tunable static analysis framework for large-scale javascript applications (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 541–551. IEEE Computer Society, 2015. doi: 10.1109/ASE.2015.28. URL https://doi.org/10.1109/ASE.2015.28.

[18] Sulekha Kulkarni, Ravi Mangal, Xin Zhang, and Mayur Naik. Accelerating program analyses by cross-program training. In Eelco Visser and Yannis Smaragdakis, editors, *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*, pages 359–377. ACM, 2016. doi:

10.1145/2983990.2984023. URL https://doi.org/10.1145/2983990.2984023.

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

[20] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In Bertrand Meyer, Luciano Baresi, and Mira Mezini, editors, *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*, pages 554–564. ACM, 2013. doi: 10.1145/2491411.2501854. URL https://doi.org/10.1145/2491411.2501854.

[21] Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient incremental static analysis using path abstraction. In Stefania Gnesi and Arend Rensink, editors, *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*, volume 8411 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2014. doi: 10.1007/978-3-642-54804-8_9. URL https://doi.org/10.1007/978-3-642-54804-8_9.

[22] Rajdeep Mukherjee, Omer Tripp, Ben Liblit, and Michael Wilson. Static analysis for AWS best practices in python code. In Karim Ali and Jan Vitek, editors, *36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany*, volume 222 of *LIPIcs*, pages 14:1–14:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi: 10.4230/LIPIcs.ECOOP.2022.14. URL https://doi.org/10.4230/LIPIcs.ECOOP.2022.14.

[23] Lori L. Pollock and Mary Lou Soffa. An incremental version of iterative data flow analysis. *IEEE Trans. Software Eng.*, 15(12): 1537–1549, 1989. doi: 10.1109/32.58766. URL https://doi.org/10.1109/32.58766.

[24] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In Ron K. Cytron and Peter Lee, editors, *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61. ACM Press, 1995. doi: 10.1145/199448.199462. URL https://doi.org/10.1145/199448.199462.

[25] Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. Know your analysis: how instrumentation aids understanding static analysis. In Neville Grech and Thierry Lavoie, editors, *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*, pages 8–13. ACM, 2019. doi: 10.1145/3315568.3329965. URL https://doi.org/10.1145/3315568.3329965.

[26] Philipp Dominik Schubert, Paul Gazzillo, Zach Patterson, Julian Braha, Fabian Schiebel, Ben Hermann, Shiyi Wei, and Eric Bodden. Static data-flow analysis for software product lines in C. *Autom. Softw. Eng.*, 29(1):35, 2022. doi: 10.1007/s10515-022-00333-1. URL https://doi.org/10.1007/s10515-022-00333-1.

[27] Semgrep, Inc. Semgrep, 2020. URL https://semgrep.dev.

[28] Amazon Web Services. Codeguru rules. https://docs.aws.amazon.com/codeguru/detector-library/, 2024.

[29] Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *2001 International Conference on Software Maintenance, ICSM 2001, Florence, Italy, November 6-10, 2001*, pages 682–691. IEEE Computer Society, 2001. doi: 10.1109/ICSM.2001.972787. URL https://doi.org/10.1109/ICSM.2001.972787.

[30] John Toman and Dan Grossman. Taming the static analysis beast. In Benjamin S. Lerner, Rastislav Bodík, and Shriram Krishnamurthi, editors, *2nd Summit on Advances in*

Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA, volume 71 of *LIPIcs*, pages 18:1–18:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.SNAPL.2017.18. URL https://doi.org/10.4230/LIPIcs.SNAPL.2017.18.