A Language-agnostic Framework for Mining Static Analysis Rules from Code Changes

Sedick David Baker Effendi
Stellenbosch University

Berk Çirisci
CNRS, IRIF

Rajdeep Mukherjee, Hoan Anh Nguyen, Omer Tripp
Amazon Web Services

***Abstract*—Static analysis tools detect a wide range of code defects, including code quality issues, security vulnerabilities, operational risks, and best-practice violations. Creating and maintaining a set of high-quality static analysis rules that detect misuses of popular libraries and SDKs across multiple languages is challenging. One of the mechanisms for inferring static analysis rules is by leveraging frequently occurring bug-fix code changes in the wild that are committed by multiple developers and into different software repositories. The intuition is that code changes following a common pattern correspond to recurring mistakes, from which deriving best practices could likely be of high value and accepted by the community.**

Automating the process of mining and clustering code changes enables a scalable mechanism to source and generate best-practices rules. From a coverage standpoint, the rules are derived from real-world code changes, which ensures that popular libraries and application domains are accounted for.

In this paper, we present a language-agnostic framework for mining and clustering code changes from software repositories using a graph-based representation dubbed MU (μ). Unlike language-specific ASTs, the MU representation generalizes across languages by modeling programs at a higher semantic level, which enables grouping of code changes that are semantically similar yet syntactically distinct. We have mined a total of 62 high-quality static analysis rules across Java, JavaScript, and Python from less than 600 code change clusters. These cover multiple libraries, including the AWS Java and Python SDKs, as well as libraries like pandas, React, Android libraries, Json parsing libraries, and many more. These rules are integrated into a cloud-based static analyzer, *Amazon CodeGuru Reviewer*. Developers have accepted 73% of recommendations from these rules during code review, which signifies the value of these rules to help improve developer productivity, make code secure, and improve code hygiene.

***Index Terms*—static analysis, mining software repository, program synthesis, coding best practices, clustering**

I. Introduction

Code changes contain rich information about bug fixes and common mistakes. Hoisting recurring fix patterns into automated and reusable analysis rules is of value to the wider developer community. These manifest as "bug fix" code changes in the software change history, where the "before" snapshot is the buggy code and the "after" snapshot is the fixed code. Changes committed by multiple developers across different software repositories are more likely to lead to best-practice rules that have high value and high action rate. From a coverage standpoint, code changes from the wild may use a wide variety of open-source libraries from various domains.

Recent efforts to automatically mine related code changes – i.e., change patterns – from open-source projects have proven useful in identifying candidates for new rules for best practices and code vulnerabilities [1, 2, 3, 4]. However, these existing approaches have several limitations: a) they do not support multiple languages, b) the mined change patterns are syntactic, and/or c) the rules are restricted to specific domains, such as cryptography APIs [4] or Python machine learning (ML) systems [3].

In this paper, we present a framework for mining static analysis rules from code changes using a language-agnostic code change extraction and clustering framework. The mined rules are integrated into a cloud-based static analyzer, *Amazon CodeGuru Reviewer*. Our presented solution has several principal advantages compared to existing approaches: 1) it generalizes to multiple languages due to the underlying semantic code representation, 2) it infers static analysis rules from a wide variety of domains/SDKs, such as AWS, Python ML, security and so on, due to hierarchical clustering of code changes, and 3) the semantic representation of code changes enables critical downstream tasks, in particular, automatic synthesis of rules from code change clusters [5], which greatly optimizes human effort in creating new rules.

Our code change extraction and clustering framework supports Java, JavaScript and Python. It translates ASTs parsed from the source code written in different languages into a single semantic Intermediate Representation (IR) dubbed MU [1]. A MU graph roughly corresponds to a single function's data-flow graph overlaid with a control-flow (not control-dependence) graph (CFG). As in prior work that used similar representations [6, 7], the MU representation is useful for finding API misuse defects where both the data flowing into an operation and the order of operations are important.

Use of a semantic IR enables us to perform origin analysis and program differencing directly atop the representation. This makes our framework language-agnostic and easily extensible to other languages. Furthermore, the availability of richer type information in MU graphs (either natively or using advanced type inference) improves the quality of various phases of the framework, including graph differencing and clustering. The semantic properties of code changes guide the grouping of these changes into domain-specific hierarchical groups, as shown in Figure 3. We cluster the MU representations of code changes in two phases: (1) we map each code change to a set

[1] "MU" originally stood for "misuse", and is pronounced as the name of the Greek letter μ.

of APIs that are added, deleted, or replaced in the change; and (2) for each API, we cluster the mapped code changes using a bottom-up clustering algorithm. The hierarchical grouping of code changes enables several downstream tasks, primarily automatic synthesis of static analysis rules from domain-specific code examples [5], or training ML models on domain-specific bug-fix pairs for automated program repair.

The main contributions of this paper are as follows:

1) A language-agnostic framework for code change extraction and clustering,
2) A novel type inference technique that uses a staged approach for type recovery, and
3) A set of high-value static analysis detectors mined from code change clusters.

II. Reated Work

Many approaches have been proposed to mine code change patterns from the histories of software systems. Most of them focus on Java. Nguyen *et al.* [8] conducted a large scale empirical study to confirm the repetitiveness of changes. They also reported that the repetitiveness is inversely proportional to the size of the changes, and building a database of repeated changes could be beneficial to change recommendation and automated program repair. In another study, Barr *et al.* [9] concluded that changes from previous commits could be used to compose new changes. They laid the empirical foundation for change pattern mining, however, their goals are not to find semantic change patterns.

Several approaches learn systematic editing to support porting [10, 11, 12] or automating similar changes [13]. Their approaches build the syntax-based edit scripts from changes in the learning dataset along with their surrounding contexts, then identify the locations and apply the suitable scripts to transform the code in the target systems. Negara *et al.* [14] aim to detect in-the-wild, high-level code change patterns from sequences of fine-grained syntactic edits recorded from IDEs using frequent-itembag mining. In comparison, their representations of changes are syntax-based, while ours is program dependence graph-based. Nguyen *et al.* [2] have shown that the latter outperformed the former on the quality of the mined change patterns. To improve the quality, [2] mines fine-grained code change patterns using a frequent subgraph mining technique on a graph-based representation that captures both control and data flows between the changed code elements. Our approach also uses a semantic representation. However, our mining algorithm uses a hierarchical clustering technique, which is more light-weight than the frequent subgraph mining. Moreover, our approach is agnostic to the target code's programming languages thanks to our code change representation using our intermediate representation, MU graphs, while the above approaches only support Java.

There have been few change pattern mining approaches for dynamic languages such as Python and JavaScript. Inspired by Defects4J [15], Widyasari *et al.* [16] and Gyimesi *et al.* [17] built large scale benchmarks of bug & fix pairs for Python and JavaScript, respectively. These benchmarks can be used for testing and debugging, but are not suitable for mining change patterns. [18, 19] mine Python fine-grained code change patterns from commits of GitHub open source repositories. BugAID [20] mines JavaScript bug patterns from a large number of commits and focus on common re-occurring across multiple projects. However, their change patterns are captured at syntax level. Similarly to approaches for Java, these also support a single language.

The closest related work to ours is from Dilhara *et al.* [3]. The change representation in their approach also captures the control and data flows in programs. In addition, it supports multiple languages, Java and Python, at the time of this writing, and can be easily extended to support other languages. The difference is that their approach supports multiple languages at *syntax level* in which it transforms ASTs from all languages to Java-like ASTs while ours faithfully translates ASTs from different languages into a single semantic intermediate representation, MU graphs. More importantly, the program differencing step in [3] also happens at syntax level whose results are used to derive the semantic change graphs while ours directly compares semantic graphs. This would result in higher quality differencing results and avoid including refactoring changes such as renaming, temporary variable introducing and switching to syntactic sugars. Another difference is that our mining algorithm uses a hierarchical clustering technique, which is more light-weight than the frequent subgraph mining that [3] inherits from [2].

Thanks to the use of domains at the top level of our hierarchical clustering, our approach can be used to mine generic change patterns as well as domain-specific ones such as security [21], API misuses [22, 23], compilation errors [24], and machine learning systems [3], etc.

III. Change Extraction and Clustering Workflow

The following section describes the overall workflow of the code change extraction and clustering as shown in Figure 1. In Section IV-A we discuss the origin analysis as well as the entity mapping algorithms used to extract pairs of procedures between two versions of code. Section IV-B describes how we formulate the two graphs into an Integer Linear Programming (ILP) optimization problem from which we can extract which entities within a MU graph pair are changed. Section V then describes our graph-pair featurization technique to produce feature vectors used by our clustering algorithm to group code changes with similar semantic modifications in an effort to find generalized best practice improvements.

IV. Code Change Extraction

In this section, we describe the design of our change extraction steps. Section IV-A describes the origin analysis process, where the goal is to match as many procedures as possible between two versions of the code. Section IV-B describes extracting differences between the mapped procedures. Finally, Section IV-C describes our novel solution for handling missing type information, mainly for dynamically typed languages, such as JavaScript and Python. Each phase operates over

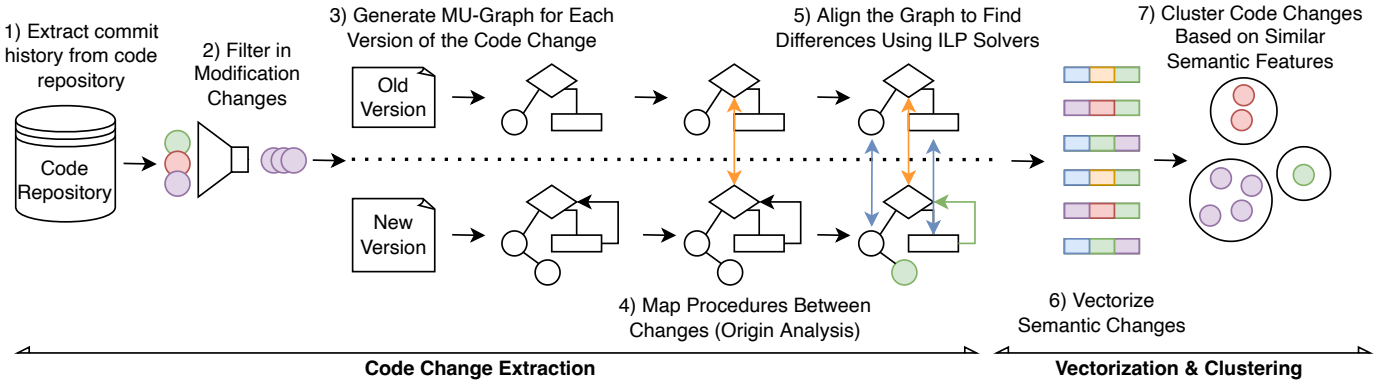

Fig. 1: The code change extraction and clustering workflow.

the MU representation, which makes the mining framework language-agnostic.

MU graphs are our in-house IR used to develop detectors in CodeGuru services. Different from most existing IRs and program analysis frameworks, MU graphs can be built from source code of changed files without requiring the projects under analysis to be complete and compilable. This makes semantic code change mining practical and scalable to tens of thousands of repositories and millions of changes. To build MU graphs for a programming language, we first use a suitable front-end to parse source code into ASTs and then translate them into MU graphs. This allows us to make use of the state-of-the-art front ends with well-developed support for name and type resolution such as Microsoft Pyright for Python and Google Closure Compiler for JavaScript. Building our own IR from ASTs also gives us more control on what information from the source code can be kept and used in downstream applications such as source line numbers, custom annotations and comments. It also gives us more control on the evolution of the IR with respect to supporting new languages as well as new versions of already-supported ones.

A. Origin Analysis

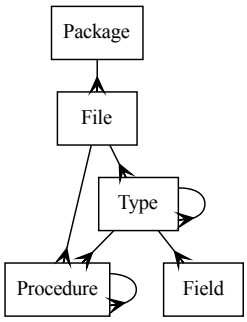

Fig. 2: A language-agnostic abstraction of the entity hierarchy.

Figure 2 shows the hierarchy of the program entities in MU graphs. For mapping the entities between the two code versions, we begin from the package level downwards, mapping parents and comparing children recursively. Entities such as files, type declarations, and procedures are ranked based on the signature information and children similarity whereas fields and procedures additionally take into account initialization and body information, respectively. The idea is similar in spirit to [25]. However, our origin analysis operates on a language-agnostic semantic intermediate representation while [25] operates on ASTs and is specific for Java. For the purpose of differentiating between procedures from a language-agnostic perspective we use the following definitions. A *method* is defined as a member of a type declaration, a *function* is defined within a file or procedure but outside of an expression, and a *lambda* is similar to a function but is always defined as part of an expression.

We use Listing 1 as our running example throughout the paper[2]. This example uses AWS S3 service to upload an object over the network. The buggy code allowed unrestricted file uploading, which could lead to S3 quotas being exceeded. The fix added the size monitoring to the file upload event, which interrupts the process if a maximum allocated usage is exceeded. Monitoring a file upload progress with `AWS.S3.ManagedUpload` can be identified as a best practice, which is demonstrated by the code change in Listing 1.

```
+ var maxBytes = 20000000;
var s3 = new AWS.S3({ ... });
var s3ManagedUpload = s3.upload(...)
s3ManagedUpload.on(
  'httpUploadProgress',
-  function (event) {
+  (event) => {
+  if (event.loaded > maxBytes) {
+    console.log('UPLOAD EXCEEDED MAX BYTES!');
+    s3ManagedUpload.abort();
+  } else {
     console.log(...);
+  }
});
```

Listing 1: A JavaScript example introducing upload quota monitoring preventing excessive uploads.

1) Algorithm overview: The origin analysis maps program entities from the root downwards in a breadth-first manner, as described by Figure 2. In the case of multiple candidates, the similarity scores are ranked, and the most similar candidate is chosen, with the score giving more weight to name and signature information (e.g. file path, method name, parameter types, etc.) over body information (e.g. children entities). We describe this process in detail in Algorithm 1.

The heuristics used when calculating the similarity score depend on the entities being compared. These scores are calculated by the functions HEADSIMILARITY and BODYSIMILARITY, respectively, and are described in detail under Section

[2]The full code change can be found on GitHub.

IV-A2. Our file similarity check uses a name-heuristic [26] e.g. (i) match pairs of files based on exact full path matches, (ii) match any leftover files based on file name matches.

Algorithm 1 Given two sets of entities M and N, the algorithm will attempt to create a mapping $E(M) = N$ based on the similarity heuristic in Algorithm 2.

```
1:  procedure MappingFunction(M, N)
2:      E ← ∅                          ▷ Stores mapping information
3:      P ← ∅                          ▷ Entity pairings and their scores
4:      for all m in M; n in N do
5:          (isSim, s) ← IsSimilar(m, n)
6:          if isSim then
7:              P ← P ∪ (s, m, n)
8:          end if
9:      end for
10:     SortByScoreDesc(P)
11:     for all (s, m, n) in P do
12:         if E(m) = ∅ then
13:             E(m) ← n               ▷ Map the highest scoring pair
14:         end if
15:     end for
16:     return E
17: end procedure
```

Algorithm 1 is used for scoring similarities for type declarations and procedures. The input for the mapping function for child nodes will be the output of the preceding mapping functions, but it is possible that child entities will be assigned new parent entities within a change and unmapped entities will be handled later. Thus, type declarations and procedures under two mapped containing entities will be considered in an initial pass and any leftover child entities are handled as a special case. The latter is often how fixes that may include some refactoring between parent entities are able to be mapped.

For ranked entities, we only compare candidate pairings that are considered similar enough based on various thresholds. Algorithm 2 describes the procedure that determines whether two entities are within a threshold where they could be considered a pair. Algorithm 2 returns a tuple consisting of a Boolean to describe if the similarity score passes a similarity threshold, and a similarity score. There are four main hyperparameters that concern the similarity threshold namely; α, β, δ, and ϵ. Here, α and β describe "moderate" and "high" head similarities, respectively, whereas δ and ϵ describe "moderate" and "high" body or content similarities, respectively. There are two hyperparameters that concern how the score is aggregated: τ adjusts how much the head similarity is weighted, whereas ρ adjusts the body similarity weighting.

2) Measuring similarities: This section describes how we measure the similarity between entity "head" and "body" information. That is, we describe how the HeadSimilarity and BodySimilarity procedures from Section IV-A1 work. Both functions return a similarity score of $s \in [0, 1]$ from low to high similarity.

Algorithm 2 Given two entities m and n, the algorithm returns a tuple where the first element is a `boolean` that is `true` if two entities are similar enough to be considered as a pair and the second element is the similarity score. AreParentsMapped uses E to check if the parent entities of m and n are mapped.

```
1:  procedure IsSimilar(m, n)
2:      x ← HeadSimilarity(m, n)
3:      y ← BodySimilarity(m, n)
4:      s ← τx + ρy      ▷ Some aggregation of the similarity
5:      isSim ← (AreParentsMapped(m, n) and x ≥ α)
6:              or (x >= 0 and y >= ϵ)
7:              or (x >= β)
8:              or (x >= α and y >= δ)
9:      return (isSim, s)
10: end procedure
```

HeadSimilarity (Algorithm 3) will typically compare information such as, in the case of a procedure, the name, identifiers, parameter, or return type. For entities closer to the root of the entity hierarchy, such as files and type declarations, we can weigh name and package information more than content. One challenge is to remain robust to small changes in name or type information. However, in the case where we do have an exact name and signature match, we mark this score as `1.0` and terminate immediately. For every other case, we use the *longest common subsequence* (LCS) on vectors generated from head information, marked as DoLCS in Algorithm 3.

In the case of a type declaration, field, or method identifiers we generate the feature vector by splitting the package path and identifier by its corresponding notation (e.g. camel case, snake case, etc.) and also use special characters or numbers in this split function. To avoid redundant matches on getters and setters, any "get" or "set" prefix is removed. An example of using FullNameToVec from Algorithm 3 would be, given `a.b.getFooBar12` or `a.b.foo_bar12`, return `[a, b, foo, Bar, 12]`.

In the case ParamToVec and ReturnTypeToVec we prioritize the type information. In the case of missing or unrecoverable type information for parameters, we use parameter names but a missing return type would be labelled "unknown". Similarly, we then generate a feature vector of these, e.g. `def foo(x: int, y) -> float` given to ParamToVec would return `[int, y]`.

BodySimilarity is calculated in a recursive manner depending on the starting entity. The lowest entity in the hierarchy that we consider in the origin analysis are procedures, which are compared using feature vectors created from node information and are compared using the weighted Jaccard similarity [27]. The features are created from a combination of identifier, type, and (in the case of procedure calls) argument information. If a set of procedures are the "body" of some parent entity, the permutations of procedure pair combinations are scored and ranked between two parents and the best

Algorithm 3 Given two entities m and n the algorithm will calculate the entity head similarity. Note that "full name" includes package, type, identifier and parameter information.

procedure HEADSIMILARITY(m, n)
  if m.name $= n$.name **then**
    if m.fullName $= n$.fullName **then**
      return 1.0 ▷ Exact name match
    else ▷ Parameter match
      $P_m \leftarrow$ PARAMTOVEC(m)
      $P_n \leftarrow$ PARAMTOVEC(n)
      return $\frac{\text{DOLCS}(P_m, P_n)}{\text{MAX}(|P_m|, |P_n|)}$
    end if
  else ▷ Attempt partial full name and return match
    $F_m \leftarrow$ FULLNAMETOVEC(m)
    $F_n \leftarrow$ FULLNAMETOVEC(n)
    $V_F \leftarrow \frac{\text{DOLCS}(F_m, F_n)}{\text{MAX}(|F_m|, |F_n|)}$
    $R_m \leftarrow$ RETURNTYPETOVEC(m)
    $R_n \leftarrow$ RETURNTYPETOVEC(n)
    $V_R \leftarrow \frac{\text{DOLCS}(R_m, R_n)}{\text{MAX}(|R_m|, |R_n|)}$
    return $\frac{V_F + V_R}{2}$
  end if
end procedure

permutation is given as the candidate body similarity score.

An example of this propagation would be comparing two type declarations. Type declarations would compare the similarity of children entities. Following Figure 2, this would be fields, procedures, and sub-types. For each of the child entities, a similarity score is calculated for the best candidate matches by starting at the leaf-level. In this example, the recursion terminates at fields and procedures, where a field initializer expression (if present) is considered as the "body" and processed as a procedure vector. As the recursion terminates, the similarity score is aggregated and passed upward to give a representative body similarity score.

3) Handling anonymous procedures: Anonymous procedures such as unnamed lambdas are given an artificial identifier. This identifier is either the name of the variable or object property it is assigned to or the name of the procedure it is being passed as an argument towards. The mapping output from Listing 1 describes the correct mapping of the lambda under the mapped parent file. Part of the signature information is that this lambda is an argument to `s3ManagedUpload::on` which helps map the two procedures.

B. Code Differencing

This code differencing phase maps the nodes between two versions of a modified procedure which is then leveraged by our clustering phase in Section V. While code differencing tools such as GumTree [28] and Shift AST [29] already exist, they operate on ASTs, while our framework relies on MU graphs, an IR at semantic level, to be language-agnostic. MU graph-based differencing would help improve the quality of the code differencing and avoid reporting refactoring changes such as renaming, temporary variable introduction, etc.

Our code differencing accepts mapped procedures in the form of pairs of program dependence graphs (PDG) [30] generated from their respective MU graphs. This effectively means that we perform graph differencing, where we can generate additional constraints based on node and edge information. This problem along with the constraints can be expressed as a generic ILP optimization problem [31] and solved by an ILP solver such as FICO Xpress [32] as is done in our case.

ILP problems can be reduced to 3-SAT and, thus, shown to be NP-complete, so we introduce the following optimizations. If we can identify whether two graphs are isomorphic, there is no need to solve the associated ILP optimization problem. While algorithms to determine graph isomorphism are inefficient in themselves, various polynomial time algorithms for trees are known [33, 34, 35]. We express MU graphs as rooted tree structures by removing loops in conditional structures. The first optimization is where we assign Knuth tuples to each tree and compare them, which allows us to decide if our graphs are isomorphic with constant complexity [36]. If two graphs are not isomorphic but we can identify subgraphs within the two graphs which are isomorphic, we can exclude these from the ILP problem to reduce the search space.

C. Layered Dynamic Language Type Recovery

Type information is necessary for performing semantic similarity comparison between code. To this end, we use a staged type resolution strategy that makes best effort type inference. The first layer relies on the compiler's type inference capabilities. This is effective at inferring primitives and language SDK types, or types of dependencies whose source code or type stubs are provided with the project under analysis. For example, we use Eclipse JDT compiler for Java, Pyright for Python, and Google Closure compiler for JavaScript.

The second layer relies on data-flow-based type hint propagation [37]. It derives hints of the type information from statements that `import` external modules into the current module and propagate them forward through the data flow. For example, an `import` declaration in Java would provide the fully-qualified name of a simple type used in the declaration of a variable in the file. Similarly, a `require` statement in JavaScript would provide the name of the module containing the imported entity. The entity type, i.e., `class` or `function`, could be inferred from how it is used in the current module, i.e., as a `function` if it is used as a callee of a function call or a `class` if it is used to create a new class instance. This technique can recover type definitions made interprocedurally within the scope of the entire application.

If the previous two layers are unable to infer the type for a given name, we then use a data-driven type inference technique based on a collected corpus of API information for popular third-party libraries. This final technique will suggest a *most-likely* type with a confidence score, which we can then accept based on supplied confidence thresholds.

Data-driven type inferencing: We use a data-driven technique to infer type information from mined API documentation similar to what is found in [38]. We start with an initial seed of

mined API specification from SDKs, such as, AWS SDKs, or popular libraries in Java, Python and JavaScript. We iteratively add new APIs and their types which fail to resolve during our experiments, by automatically mining from the corresponding library documentation or SDKs. This iteratively improves the corpus based on the input dataset.

We start with an oracle, being a corpus of types and APIs, mined from documentation to form triplets like {domain, type, API}, as illustrated in Figure 3. For every node in the MU graph that has an unknown type, we track all method invocations and check if they occur in the corpus. If our corpus returns a unique type, we leverage that type information from our corpus. Otherwise, if the API reference is too generic, and our corpus suggest multiple types, we use a ranking function to get the most relevant type, which is described next.

When considering the receiver name, we use common naming conventions to deduce the potential type. We often find that developers describe the identifier based on the type name as either an abbreviation, the name itself, or some combination of the two. Examples of such are `s3Bucket`, `ddb`, or `currentDate` for objects of type `AWS.S3`, `AWS.DynamoDB`, and `Date` respectively. Using string similarity techniques such as the LCS score used in Section IV-A, case transforms (camel-case, snake-case, etc.), and abbreviations from types in the corpus, we accumulate a list of matches to add to the ranking of likely types.

V. Clustering

If a code change induces a perfect clustering, then the clustering algorithm outputs an exact cluster. However, in practice, a code change may contain unrelated changes not related to the actual interesting change. Additionally, code changes may be syntactically different but semantically equivalent. For example, error handling in code can be performed in different ways but they are similar at a semantic level, for example, – using a conditional-loop (while loop or for loop), using an if-statement, or by simply logging the output. These properties of code changes makes it hard to achieve perfect grouping.

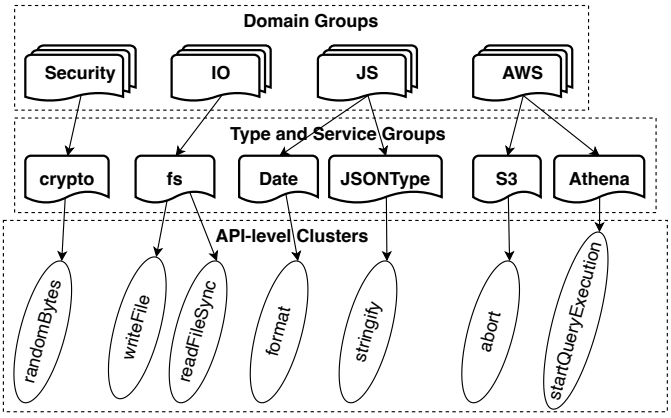

Fig. 3: The *{domain, library, type, API}* hierarchy for grouping change graph pairs. This illustrates the first stage of the clustering where change graph pairs are grouped together based on where the main API of the code change fits in the hierarchy. These groupings, or *API-level clusters*, are then given as input to the agglomerative clustering algorithm.

Prior work on hierarchical code change clustering, such as Getafix [23], summarizes fix patterns into a hierarchy ranging from general to specific patterns. Compared to the rules in Getafix [23], our technique is driven by data, not top-down categories. Where Getafix is a bug-fixing tool that suggests fixes for only six specific bug categories, we learn rules from "ad-hoc" clusters, thus providing higher coverage and diversity with the rules we generate, across multiple languages, categories, libraries and SDKs. DiffCode [21] presented a clustering algorithm that uses commonalities between semantic code changes to infer security rules pertaining to Java Crypto APIs. What follows is the description of our clustering framework.

A. Hierarchical Grouping

The hierarchical grouping of the code changes is performed by leveraging the node mapping information, {*added, removed, mapped*}, corresponding to the APIs in the code change, where the node mappings are obtained from the code differencing phase (see Section IV-B). For code changes that contain multiple APIs, we heuristically determine the *main* API in the change by considering the: (1) node centrality (number of transitive edges induced by the API node), (2) *invariant* call pairs (API nodes with the same API name that are mapped between before code and after code) or use an *added*, *removed*, or *changed* API node in absence of an invariant API, and (3) *domain* preference (use prior knowledge of the preferred domain, such as library/SDK if any). An example of hierarchical grouping is illustrated in Figure 3. It groups code changes into a hierarchy, *{domain, library, type, API}*, by identifying the main APIs of the change. The properties of the main API gives the general domain (e.g. AWS, Apache, Android, Machine Learning, Security, JavaScript standard library, etc.), the services or library (AWS.S3, AWS.DynamoDB, Java.lang, pandas, Crypto, etc.), type (AWS.S3.ObjectMetadata, AWS.S3.PutObjectRequest, pandas.DataFrame, java.util.HashMap, etc.), and finally the name of the main API within the change. The hierarchy can be inferred completely or partially, depending on the extent of the type information available during clustering. For example, our example in Listing 1 would be grouped under the hierarchy, *AWS* $\rightarrow$ *S3* $\rightarrow$ *abort*, where the API node *abort* is ADDED. The hierarchical grouping of code changes are beneficial for several downstream analyses such as, automatic synthesis of static analysis rules from domain-specific code examples [5], review domain-specific clusters by human reviewers for inferring static analysis rules VI, or to train a machine learning model on domain-specific bug-fix pairs for automated program repair.

B. Bottom-up Clustering

Given a set of code changes, they are first grouped by into their corresponding hierarchical API groups. Once the hierarchies are determined, the code changes are clustered based on their semantic similarity using agglomerative hierarchical clustering [39, 40]. Compared to other classical clustering algorithm, such as, the divisive analysis [41], or DBSCAN [42], we found that the agglomerative clustering favors smaller clusters, which works well with the size of the API groups, that are obtained from V-A. While an implementation for k-modes clustering with weighted attributes exists [43], we would need to dynamically calculate an optimal value for k for each API grouping, more accurately if a guaranteed "knee" or "elbow" was present in each. With hierarchical clustering, we can use our domain knowledge of the code change features and kinds of similarities and differences we expect, to provide a cut-off height that generalizes well enough across API groups.

The following steps are performed on each change graph to precisely capture the semantic transformations of the change. Recall that a change graph contains two MU graphs (a MU graph before the change and a MU graph after the change) and mappings between their nodes, if any.

1) Slicing: Program slicing is performed on the MU graphs by considering the main APIs obtained from hierarchical grouping, as the anchor nodes. This prunes out the sub-graphs that are disjoint from the actual change, while preserving the semantic transformation of the code change.

2) Featurization: The resulting change graphs are featurized using semantic predicates, which gives the feature vectors. The featurization of a change graph is performed using two sets of predicates: (1) node-level predicates and (2) graph-level predicates. *Node-level* predicates capture node properties such as, method-calls, the receiver type of an API, conditional checks, caught exceptions, context predicates (such as try-catch, while, for, etc), names of data nodes, and assignment predicates to capture assignments. These predicates are mined from the underlying MU graph using dedicated miners. Similar to node-level predicates, *graph-level* predicates are mined from the MU graph, and these predicates capture second-order properties of the MU graph, including transitive data-flow, and downstream conditional checks (for example, to determine if the return value of an API call is used downstream or is directly returned from the function). Finally, features are prefixed with whether they are present in the code before the change or after the change to help represent the temporal dimension of the changes. Code changes that agree on the same set of properties or features (node-level or graph-level) are clustered together. The accuracy of our clustering algorithm is driven by precisely representing each code change with a set of semantic predicates that captures node- and graph-level properties. The following are a few features using our example in Listing 1. Here, the *ctxt-type* predicate gives the enclosing context *{IF}*, the *rcvr-type* predicate gives the receiver type of the API call, and *mthd-call* gives the name of the method call.

```
after:log() :> ctxt-type: [IF]
after:abort() :> rcvr-type: .*aws-sdk.S3.*
after:abort() :> ctxt-type: [IF]
after:abort() :> mthd-call: .*abort.*
```

3) Compute Similarity: The Jaccard index [44] is used to measure the overlap between elements within different sets, while the Jaccard distance [45] projects the dissimilarity of these sets into a measurable space. The total number of possible code change features in our dataset is large, which implies high dimensionality if one were to use techniques to convert this categorical data to comply with the input of the above-mentioned distance metrics, such as one-hot-encoding. This does not scale well with Euclidean distance, but in the case of high dimensionality one could use cosine distance [46] or use dimensionality reduction [47]. Therefore, despite these possible solutions, Euclidean and cosine distance fails to effectively capture the distance between categorical and mixed data due to the distortion resulting from these transformations [48].

The agglomerative clustering algorithm uses the weighted Jaccard distance metric to measure vector dissimilarity. The weights are computed for each node in the MU graphs by considering the number of transitive edges induced by that node. These weights are associated with the corresponding features or predicates mined for that node. The difference of the features (diff-features) obtained from a change graph represents the semantic transformations of the change, that is, it contains the set of weighted predicates corresponding to the changed nodes or added nodes, or deleted nodes. Given a pair of change graphs, {G1, G2}, the Jaccard distance is computed from the diff-features of G1 and G2. If the distance metrics satisfy a certain threshold, then the graphs are clustered together or else, they form their respective clusters.

4) Determining cluster quality: The clustering algorithm is designed in a way such that 1) the inter-clusters distance is maximized while 2) intra-cluster distance is minimized. The accuracy of the clustering algorithm was measured offline — through "shadow review". From the shadow review feedback, we measure the *homogeneity score* that gives the measure of the similarity of the code changes in a cluster, and *completeness* that gives the measure of how much similar code changes are put together by the clustering algorithm.

C. Cluster Selection Process

While methods such as Elbow, Silhouette, and Gap statistic are generally used to determine the optimal number of clusters, our clustering process is two-step where API-level groups, once clustered, may give different answers when using each of these measures. Since the agglomerative clustering is run once we have API-level groups, these groups should already be similar in terms of usages of the main API (see Section VA). In order to identify unique clusters, we aim for tight cohesiveness and thus opt for a very short cutoff height. These clusters are then ranked based on few criterions – (a) number of code changes (3 or more), (b) Jaccard distance metric, (c) code changes from 2 or more repositories or 2 or more commit IDs to ensure that only clusters formed from a diverse set of repositories or commits are considered.

VI. Evaluation

In this section, we report on experiments. Our experiments are guided by the following research hypotheses:

H1: Language-agnostic code change extraction and clustering atop the MU representation allows significant reuse of multiple components of the framework including origin analysis, graph differencing, and hierarchical clustering.

H2: A robust type inference strategy is critical for identifying the semantic nature of the code changes.

H3: Software defects that repeat themselves across different codebase are typically remediated in similar if not identical ways. Lifting common mistakes and their corresponding fixes to the level of reusable automated rules, thus, enables early detection of the same bug patterns.

A. Input Dataset

We have obtained code changes from GitHub packages that have licenses, Apache or MIT, and popularity of at least 4 stars. This led to a dataset of 27K Java repositories, 8K JavaScript repositories and 25K Python repositories, which consists of 1.8M, 1.1M and 1.6M code changes, respectively. We used the GitHub search API to identify the repositories that met our selection criteria programmatically, namely by programming language, license and rating.

B. Experimental Setting

The workflow presented in Figure 1 was run on an Amazon EC2 machine with 48 cores, 384GB of memory, and 2 hard drives of size 1TB each. Our experiments utilized 300GB of heap memory, a stack size of 256MB, and 4 threads. On top of this hardware configuration, we had to enforce several constraints to ensure timely termination of the workflow. ILP optimization problems for graph differencing (Section IV-B) scale poorly with the size of the input MU graph pair. As our implementation is multithreaded, there are times when each thread requires mapping large amounts of heap space to solve its respective ILP optimization problem (graphs having 500+ nodes and 800+ edges) at the same time, thereby leading to memory exhaustion. To address this issue, our first constraint was to limit the size of the graphs handed to the ILP solver, based on the average node count of 400 nodes and edge count of 600 edges, between the two MU graphs being compared. Our second limit is on the thread pool size of 4 threads, which, together with the ILP solver limit, gives us an idea of the maximum amount of memory our process could demand at any point in time. Third, we constrained the number of change graphs per git commit to 50 at most. This helps discard commits with many change graphs, which typically correspond to global code refactorings. Through a combination of thread count and graph size limits, we were able to guarantee timely termination of our change extraction.

C. Effort to support new languages

The effort to extend the framework from one language to another is minimal, due to the high degree of reusability of the core algorithms over MU graphs for code change representation across languages, including origin analysis, graph differencing, and hierarchical clustering. This validates our research hypothesis, *H1*. The only major addition was done to the entity hierarchy in the origin analysis, where we have added support for Python and JavaScript, to the existing support for Java. That is, in Java, method is the only kind of procedures which can only be declared as a member of a type, whereas in Python and JavaScript, there also exists functions which can be defined under files, methods or other functions.

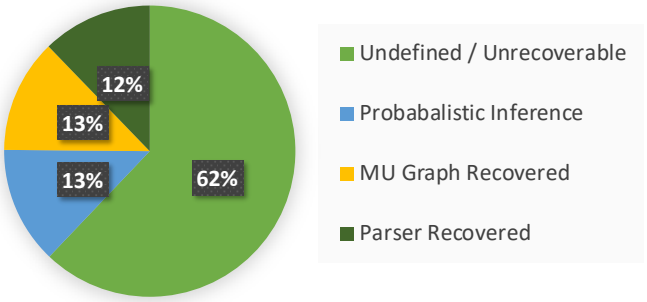

Fig. 4: Result of the layered type recovery.

D. Type Recovery

Figure 4 illustrates the distribution of the type information recovered in JavaScript programs using our novel type inference strategy. While a large majority of type information is unknown or undefined, our approach is able to recover about 26% additional type information we otherwise would not have. The significant boost in the type information is possible due to the use of the type stubs and smarter type recovery heuristic system used during MU graph construction. The type stubs are automatically curated by mining API signatures from popular Java libraries, JavaScript frameworks and Python modules, including AWS SDKs. For example, for Java, we passed the JAR files containing those libraries as the dependencies to the Eclipse JDT compiler to help it resolve the types. For Python, we used third party type stubs [49], called boto3-stubs, along with Pyright to improve the type information for Python applications that use the AWS Python SDK. For type recovery in JavaScript programs that use the AWS SDK, we built a type stub by mining the API signatures from the AWS JavaScript SDK. In addition to AWS, we built domain specific type stubs for popular libraries in various other non-AWS domains. The availability of type information enables us to cluster more code changes in reviewable API groups with higher cluster homogeneity compared to attempting to cluster a large group of otherwise unrelated code changes, which validates our hypothesis, *H2*.

The 62% unrecoverable types indicates room for improvement. One such improvement could be adding type information around the API parameter and return types. Although continued web-scraping effort to increase the size of the API corpus should also improve the type inference yield in future, a larger corpus with more fields may impact the scalability

TABLE I: Acceptance rate and frequency of rules.

Language	Total	SDK/Libraries	Acceptance	Frequency
Java	42	AWS, Apache Android, Gson, Java FileSystem	73.8%	0.75%
Python	12	AWS, Python3 best practices, hashlib, socket, math	76.4%	0.62%
JavaScript	8	AWS, Crypto, jQuery, React	72.8%	0.83%

of this technique. A learning-based approach may be worth exploring to address this potential issue.

E. Manual Review of Selected Clusters

Top-ranked clusters are manually reviewed by domain experts, including security, privacy, programming languages, machine learning, and so on. This is a gating requirement for the rules due to the clusters to be integrated into the product, where we also review the detections from these rules as they progress through our deployment stages, then monitor user feedback and action once the rules are launched. One way to measure the accuracy of the presented approach is by measuring the *yield*, which is the ratio of the number of mined rules over the number of clusters. The yield varies between 8 - 11%. For example, we have randomly picked 418 top-ranked Java clusters and manually reviewed them among a team of expert Java practitioners. The task of the review process is to analyze the conforming and non-conforming code in these clusters, and suggest any potential static analysis rules. On average, each reviewer spends 5 minutes to review a cluster. This process resulted in 42 static analysis rules from 418 clusters for Java, with the yield of 10%. For JavaScript, we picked 70 clusters, which were reviewed by expert JavaScript practitioners, and resulted in 8 JavaScript rules with the yield of 11%. For Python, 12 Python rules were inferred from 147 clusters with the yield of 8%.

F. Rule Evaluation from Code Review Feedback

An Amazon CodeGuru action can be triggered by a pull request (PR), push, or scheduled run of the CI/CD pipeline. Among its available integration points, CodeGuru Reviewer can leave comments on PRs. In this setup, the developer can provide free-form textual feedback and/or a rating (either thumbs-up/down or useful/not useful/not sure, depending on the code review tool). We treat "thumbs-up" and "useful" ratings as positive, and the rest as negative. We map textual feedback sans rating to positive/negative using a sentiment analysis model. Furthermore, we compute acceptance as the proportion of positive feedback points out of all feedback points.

Table I gives the acceptance rate and frequency of the mined rules per language. The *acceptance* rate metrics are used as an indication of whether developers have found a given rule's review comments useful. Column 3 in Table I shows the acceptance rate for the static analysis rules for a time period of 8 weeks. During this time period, CodeGuru analyzed 80K+ PRs, and published 20K+ recommendations, and approximately 2K feedback points. Developers have accepted over 73.8% of the recommendations made by the Java rules, 76.4% of the recommendations from Python rules, and 72.8% of the recommendations from the JavaScript rules. We have analyzed the negative feedback points (roughly 27%, or 540 responses, out of the overall feedback). These break into several modes: (1) addressing the CodeGuru Reviewer recommendation would de-focus the PR; (2) the recommendation is applicable, but can only be addressed by upgrading to the latest AWS Python SDK; (3) code quality is not a concern, since the codebase is experimental or the code in question is for testing; or (4) the detection is incorrect (false positive), which accounts for less than 3% of the cases.

The high acceptance rate suggests that the static analysis rules inferred from code change clusters helps detect similar repetitive bugs in new projects when these rules are run in production environments, which validates our hypothesis, *H3*.

Column 4 in Table I gives the frequency of the CodeGuru rules per language. The frequency of a rule is given by the percentage of the recommendations made by that rule compared to the total number of recommendations.

G. Mined Rules

A total of 62 static analysis rules are inferred from the code change clusters. All these rules have met our accuracy bar of 70% actionable findings, which is measured offline — through "shadow review" — ahead of rule deployment, then monitored and validated in production through developer feedback on the comments CodeGuru leaves on pull requests. These rules are integrated into Amazon CodeGuru service and deployed across multiple cloud regions. All the CodeGuru rules are available online, in the CodeGuru Detector Library. For brevity, we present few representative rules in Table II. Column 1-4 give the *language* of the source code that the rule provides a recommendation on, *category* of the rule that includes the Common Weakness Enumeration (CWE) for security rules, *domain* the rule belongs to, and the *main API(s)* of the code changes, respectively. Column 5 gives the description of the rule along with the reference to the corresponding rule in the CodeGuru Detector Library [72] or API reference document, where applicable. The incremental process to generate additional rules from change clusters reduces to dataset selection. Examples of datasets would be commits involving a new AWS SDK version or popular third-party library, or code changes in a new programming languages that CodeGuru Reviewer is adding support for. Once the dataset selection is complete, we apply the procedure described in Sections III, IV and V.

H. Impact of Mined Rules

The mined rules are considered high-value because they detect high-fidelity bugs in code. From our conversations with developers, the textual feedback they provided, and our own review of some of the detections, we have identified few main factors that contribute to the usefulness of these rules.

Missed features. SDK (new) features are sometimes missed by developers. Pagination, retry and error handling are examples of such features, where developers not familiar with these

TABLE II: Representative CodeGuru Rules Inferred from Code Change Clusters.

Language	Category	Domain	API	Rule Description
Java	security (CWE-429)	AWS/S3	ObjectMetadata:: setContentLength	Detects if an object of type stream is uploaded to Amazon S3 without setting the content length of the object.[50]
Java	security (CWE-19)	AWS/S3	AmazonS3::putObject	Avoids a ResetException when uploading objects to S3 using a stream by setting the read limit using setReadLimit.[51]
Java	code quality	Android	getSupport, ActionBar	The method "getSupportActionBar" returns "null" if the Android activity does not have an action bar. One must null-check the value returned by "getSupportActionBar", if the action bar is not explicitly set by a "setSupportActionBar" call.[52]
Java	code quality	AWS/S3	AmazonS3:: getCachedResponseMetadata	Detects null dereferencing error if the code is accessing the cached response metadata using the method getCachedResponseMetadata without performing a null check on the metadata.[53]
Java	code quality	Lang	Map::entrySet	Detects use of map iteration on keys and querying for their respective values, and suggest to use a more efficient iteration on map entries.[54]
Java	code quality	AWS/ DynamoDB	AmazonDynamoDB::getItem	AWS DynamoDB's GetItemResult.getItem can be null if the item is not found in the database. DynamoDB.getItem does not throw an exception when a returned item is null, so we recommend checking if the value before accessing it.[55]
Java	code quality	java.util.concurrent	ConcurrentHashMap::get	Oversynchronization with ConcurrentHashMap or ConcurrentLinkedQueue can reduce program performance. An oversynchronization happens when unnecessary synchronization is used that prevents parallel execution.[56]
Java	code quality	Gson	Gson::fromJson	Detects code that deserializes a list of JSON items by iterating in a loop. Instead, one can directly deserialize into a list by specifying the correct parameterized type using the "TypeToken" class.[57]
Java	security	android.content .Context	Context::startActivity	When launching an Android activity with an implicit intent, one must check if an application that can receive the intent exists on the device. A missing check can cause an application crash.[58]
Java	code quality	Android	LayoutInflater	It is recommended to use "LayoutInflater" to inflate views rather than inflating them programmatically, especially if the layout is complex. Defining the layout and inflating it is easier than creating the layout completely in code.[59]
Java	security	android.content .Context	Context:: getContentResolver().query(..)	It is recommended to check if the cursor pointing to the result of a database operation is empty. If a check on the value returned by moveToFirst is missing, subsequent database read operations can cause your application to crash.[60]
Java	code quality	java.util.concurrent	ExecutorService:: shutdownNow(..)	Sudden service shutdown might prevent a graceful termination of threads. This can make the code harder to debug.[61]
Java	code quality	java.util	Maps:: newHashMapWithExpectedSize	Detects code that uses inefficient APIs which might impact the performance.[62]
Python	security	boto3	boto3.client	Recreating AWS clients from scratch in each Lambda function invocation is expensive and can lead to availability risks. Clients should be cached across invocations.[63]
Python	Cryptography	secrets	secrets.token_bytes	Detects algorithms with known weaknesses, certain padding modes, lack of integrity checks, insufficiently large key sizes, and insecure combinations of the aforementioned.[64]
Python	code quality	AWS/SNS	subscribe	The Amazon SNS subscribe operation by default returns either the subscription ARN or the phrase: PENDING CONFIRMATION. In order to always return the subscription ARN, it is recommended to set the ReturnSubscriptionArn argument to True.[65]
Python	code quality	hashlib	hashlib.new	Detects the usage of the new() from the hashlib module and recommend using hashlib constructors instead.[66]
Python	security	socket	create_connection	Not setting the connection timeout parameter in Python Socket can result in blocking socket mode. In blocking mode, operations block until complete or the system returns an error.[67]
Python	security (CWE-502)	jsonpickle	jsonpickle.decode	Detects deserialization of untrusted or potentially malformed data which can be exploited for denial of service or to induce running untrusted code.[68]
JavaScript	security (CWE-310, CWE-311)	AWS/KMS	reEncrypt	Checks whether the 'reEncrypt' API is used instead of decrypting data and immediately encrypting it again.[69]
JavaScript	security	Crypto	generateKeyPair, generateKeyPairSync, createECDH	Detects if a code is generating a key that is less than 256 bytes (2048 bits) for an HMAC. Cryptographic systems require a sufficient key size to be robust against brute force attacks. It is recommended to use keys that are larger than 256 bytes.[70]
JavaScript	security	http	http.Agent()	Detects usage of "http.Agent()" function to configure connections to transmit data in clear text, and recommend to use "https.Agent()" instead to transfer data in an encrypted form.[71]

built-in capabilities sometimes implement "manual" mechanisms instead. Another example is manual polling versus the recommended use of the waiter utility.

Missed expectations. Developers sometimes assume, rather than verify, the functionality of a given API or the role of a given parameter. An example is the `QueryResponse::hasItems` method, whose (`boolean`) return value is sometimes incorrectly interpreted as the response containing a non-empty collection of items, whereas, in fact, meant that the response defines an `Items` property. To make sure whether any items are contained in the response, the developer needs to also check `Items::isEmpty`. Mistakes like this can lead to large-scale operational failures.

VII. Conclusion

In this paper, we presented a framework for learning static analysis rules from code changes using a language-agnostic code change extraction and clustering framework. These rules are deployed in a cloud-based static analyzer, *Amazon CodeGuru*. Unlike other mining approaches, our presented solution scales to multiple languages due to the underlying MU-based semantic representation of code changes, enables robust type resolution which can be leveraged by downstream tasks such as clustering or automatic rule synthesis engine, and infers domain specific semantic rules due to hierarchical clustering. We have minded a total of 62 static analysis rules across multiple languages such as Java, Python, and JavaScript. These rules have an overall acceptance rate of 73% during code review phase. In the future, we plan to infer static analysis rules for more languages using our approach, and improve the type resolution strategy for better yield.

References

[1] A. Sven, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static api-misuse detection," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, pp. 265–275.

[2] H. A. Nguyen, T. N. Nguyen, D. Dig, S. Nguyen, H. Tran, and M. Hilton, "Graph-based mining of in-the-wild, fine-grained, semantic code change patterns," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 819–830.

[3] M. Dilhara, A. Ketkar, N. Sannidhi, and D. Dig, "Discovering repetitive code changes in python ml systems," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 736–748.

[4] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring crypto api rules from code changes," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 450–464. [Online]. Available: https://doi.org/10.1145/3192366.3192403

[5] P. Garg and S. S. SHS, "Example-based synthesis of static analysis rules," *Proc. ACM Program. Lang.*, no. OOPSLA, 2022. To appear. [Online]. Available: https://arxiv.org/pdf/2204.08643.pdf

[6] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "Investigating next steps in static API-misuse detection," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, M. D. Storey, B. Adams, and S. Haiduc, Eds. IEEE / ACM, 2019, pp. 265–275. [Online]. Available: https://doi.org/10.1109/MSR.2019.00053

[7] ——, "A systematic evaluation of static API-misuse detectors," *IEEE Trans. Software Eng.*, vol. 45, no. 12, pp. 1170–1188, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2018.2827384

[8] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 180–190.

[9] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 306–317.

[10] N. Meng, M. Kim, and K. S. McKinley, "Sydit: Creating and applying a program transformation from an example," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. ACM, 2011, pp. 440–443.

[11] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 53:1–53:11.

[12] B. Ray, C. Wiley, and M. Kim, "REPERTOIRE: a cross-system porting analysis tool for forked software projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 8:1–8:4.

[13] N. Meng, M. Kim, and K. S. McKinley, "LASE: locating and applying systematic edits by learning from examples," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, pp. 502–511.

[14] S. Negara, M. Codoban, D. Dig, and R. E. Johnson, "Mining fine-grained code changes to detect unknown change patterns," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 803–813. [Online]. Available: https://doi.org/10.1145/2568225.2568317

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser.

ISSTA 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 437–440. [Online]. Available: https://doi.org/10.1145/2610384.2628055

[16] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1556–1560. [Online]. Available: https://doi.org/10.1145/3368089.3417943

[17] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszedes, R. Ferenc, and A. Mesbah, "Bugsjs: a benchmark of javascript bugs," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 90–101.

[18] W. Lin, Z. Chen, W. Ma, L. Chen, L. Xu, and B. Xu, "An empirical study on the characteristics of python fine-grained source code change types," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 188–199.

[19] Y. Yang, T. He, Y. Feng, S. Liu, and B. Xu, "Mining python fix patterns via analyzing fine-grained source code changes," *Empirical Softw. Engg.*, vol. 27, no. 2, mar 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10087-1

[20] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in javascript," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 144–156. [Online]. Available: https://doi.org/10.1145/2950290.2950308

[21] R. Paletov, P. Tsankov, V. Raychev, and M. Vechev, "Inferring crypto api rules from code changes," *SIGPLAN Not.*, vol. 53, no. 4, p. 450–464, jun 2018. [Online]. Available: https://doi.org/10.1145/3296979.3192403

[22] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, vol. 45, no. 12, pp. 1170–1188, 2018.

[23] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360585

[24] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian, "Deepdelta: Learning to repair compilation errors," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 925–936. [Online]. Available: https://doi.org/10.1145/3338906.3340455

[25] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 302–321. [Online]. Available: https://doi.org/10.1145/1869459.1869486

[26] D. Steidl, B. Hummel, and E. Juergens, "Incremental origin analysis of source code files," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 42–51.

[27] S. Ioffe, "Improved consistent sampling, weighted minhash and l1 sketching," in *2010 IEEE international conference on data mining*. IEEE, 2010, pp. 246–255.

[28] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[29] I. Shake Security, "Shift ast," https://shift-ast.org, accessed: 2022-08-24.

[30] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[31] S. E. Elmaghraby, "Operations research," in *Encyclopedia of Physical Science and Technology (Third Edition)*, 2003, ch. 3, pp. 193–218.

[32] P. Belotti, Z. Csizmadia, S. Heipcke, and S. Lannez, "Robust optimization with fico tm xpress," 2014.

[33] A. V. Aho and J. E. Hopcroft, *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[34] L. Babai and E. M. Luks, "Canonical labeling of graphs," in *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, 1983, pp. 171–183.

[35] L. Babai, "Graph isomorphism in quasipolynomial time," in *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, 2016, pp. 684–697.

[36] A. Smal, "Explanation for 'tree isomorphism'talk," 2008.

[37] R. Mukherjee, O. Tripp, B. Liblit, and M. Wilson, "Static analysis for aws best practices in python code," *arXiv preprint arXiv:2205.04432*, 2022.

[38] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 643–652.

[39] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[40] D. Eppstein, "Fast hierarchical clustering and other applications of dynamic closest pairs," *Journal of Experimental Algorithmics (JEA)*, vol. 5, pp. 1–es, 2000.

[41] W. T. Williams and J. M. Lambert, "Multivariate methods

in plant ecology: I. association-analysis in plant communities," in *The Journal of Ecology*, 1959, pp. 83–101.
[42] M. Ester, H. P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *kdd*, vol. 96, 1996, pp. 226–231.
[43] Z. He, X. Xu, and S. Deng, "Attribute value weighting in k-modes clustering. expert systems with applications," in *Expert Systems with Applications,*, 2011, pp. 15 365–15 369.
[44] P. Jaccard, "The distribution of the flora in the alpine zone," in *New phytologist*, 1912, pp. 37–50.
[45] M. Levandowsky and D. Winter, "Distance between sets," in *Nature*, 1971, pp. 34–35.
[46] Y. Qian, F. Li, J. Liang, B. Liu, and C. Dang, "Space structure and clustering of categorical data," in *IEEE transactions on neural networks and learning systems*, 2015, pp. 2047–2059.
[47] C. O. S. Sorzano, J. Vargas, and A. P. Montano, "A survey of dimensionality reduction techniques," in *arXiv preprint arXiv:1403.2877*, 2014.
[48] T. R. dos Santos and L. E. Zárate, "Categorical data clustering: What similarity measure to recommend?" in *Expert Systems with Applications*, 2015, pp. 1247–1260.
[49] V. Emelianov. mypy_boto3_builder: Type annotations builder for boto3 compatible with VSCode, PyCharm, Emacs, Sublime Text, pyright and mypy. [Online]. Available: https://vemel.github.io/mypy_boto3_builder/
[50] "CodeGuru Rule: Unchecked s3 object metadata content length," https://docs.aws.amazon.com/codeguru/detector-library/java/s3-object-metadata-content-length-check/.
[51] "CodeGuru Rule: Avoid reset exception in Amazon S3," https://docs.aws.amazon.com/codeguru/detector-library/java/avoid-reset-exception-rule/.
[52] "CodeGuru Rule: Missing null check on returned support action bar," https://developer.android.com/reference/androidx/appcompat/app/AppCompatActivity#getSupportActionBar().
[53] "CodeGuru Rule: Missing null check for cache response metadata," https://docs.aws.amazon.com/codeguru/detector-library/java/null-check-cache-response-metadata/.
[54] "CodeGuru Rule: Inefficient map entry iteration," https://docs.aws.amazon.com/codeguru/detector-library/java/iterate-on-map-entries/.
[55] "CodeGuru Rule: AWS DynamoDB getItem output is not null checked," https://docs.aws.amazon.com/codeguru/detector-library/java/aws-dynamodb-getitem-null-check/.
[56] "CodeGuru Rule: Oversynchronization," https://docs.aws.amazon.com/codeguru/detector-library/java/concurrency-over-synchronization/.
[57] "CodeGuru Rule: Simplifiable code," https://docs.aws.amazon.com/codeguru/detector-library/java/simplifiable-code/.
[58] "CodeGuru Rule: Missing check when launching an Android activity with an implicit intent," https://docs.aws.amazon.com/codeguru/detector-library/java/missing-check-on-android-startactivity/.
[59] "CodeGuru Rule: Low maintainability with old Android features," https://docs.aws.amazon.com/codeguru/detector-library/java/old-android-features/.
[60] "CodeGuru Rule: Missing check on the value returned by moveToFirst API," https://docs.aws.amazon.com/codeguru/detector-library/java/output-ignored-on-movetofirst-operation/.
[61] "CodeGuru Rule: Improper service shutdown," https://docs.aws.amazon.com/codeguru/detector-library/java/sudden-service-shutdown/.
[62] "CodeGuru Rule: Use of inefficient APIs," https://docs.aws.amazon.com/codeguru/detector-library/java/inefficient-apis/.
[63] "CodeGuru Rule: AWS client not reused in a Lambda function," https://docs.aws.amazon.com/codeguru/detector-library/python/lambda-client-reuse/.
[64] "CodeGuru Rule: Insecure cryptography," https://docs.aws.amazon.com/codeguru/detector-library/python/insecure-cryptography/.
[65] "CodeGuru Rule: Set SNS return subscription ARN," https://docs.aws.amazon.com/codeguru/detector-library/python/sns-set-return-subscription-arn/.
[66] "CodeGuru Rule: Inefficient new method from hashlib," https://docs.aws.amazon.com/codeguru/detector-library/python/hashlib-constructor/.
[67] "CodeGuru Rule: Socket connection timeout," https://docs.aws.amazon.com/codeguru/detector-library/python/socket-connection-timeout/.
[68] "CodeGuru Rule: Deserialization of untrusted object," https://docs.aws.amazon.com/codeguru/detector-library/python/untrusted-deserialization/.
[69] "CodeGuru Rule: Client-side KMS reencryption," https://docs.aws.amazon.com/codeguru/detector-library/java/aws-kms-reencryption/.
[70] "CodeGuru Rule: Inadequate encryption strength," https://cwe.mitre.org/data/definitions/326.html.
[71] "CodeGuru Rule: Cleartext transmission of sensitive information," https://cwe.mitre.org/data/definitions/319.html.
[72] "CodeGuru Rules," https://docs.aws.amazon.com/codeguru/detector-library/.