Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler

ABSTRACT

In this work we apply anomaly detection to source code and bytecode to facilitate development of a programming language and its compiler. We define anomaly as a code fragment that is different from typical code written in a particular programming language. Identifying such code fragments is beneficial to both language developers and end users, since anomalies may indicate potential issues with the compiler or with runtime performance. Moreover, anomalies could correspond to problems in language design. For this study, we choose Kotlin as the target programming language. We outline and discuss approaches to obtaining vector representations of source code and bytecode and to detection of anomalies across vectorized code snippets. The paper presents a method that aims to detect two types of anomalies: syntax tree anomalies and so-called compiler-induced anomalies that arise only in the compiled bytecode. We describe several experiments that employ different combinations of vectorizaton and anomaly detection techniques, and discuss types of detected anomalies and their usefulness for language developers. We demonstrate that the extracted anomalies and the underlying extraction technique provide additional value for language development.

ACM Reference Format:

. 2020. Using Large-Scale Anomaly Detection on Code to Improve Kotlin Compiler. In *Proceedings of The 17th International Conference on Mining Software Repositories (MSR 2020).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Anomaly detection techniques [18] have been successfully applied to a variety of practical tasks in many areas. These techniques help to detect cyberattacks [32, 35], identify pathologies in medical images [6], and detect traces of fraudulent activities in financial data [4].

In software engineering, anomaly detection is widely applied to finding bugs [17], security issues [14], architectural design flaws [28], workflow errors [15], synchronization errors in concurrent programs [36], and other anomalous patterns in code or other software artifacts. Definitions of an anomaly vary from study to study and are imposed by the exact task in each case.

In this study, we propose a new area of application for anomaly detection — finding issues in programming language compilers. We define *code anomalies* as fragments of code that are not typical within the community or an ecosystem of a given programming language, or machine code that is uncharacteristic within the range of output produced by the compiler. Such code could be useful to both users and developers of the language: for example, anomalous, yet actually existing, code snippets can highlight flaws in the language design or indicate problems in the performance of programs, problems in code generation, compiler optimizations, type inference, or data flow analysis, which turns them into a valuable material for compiler tests.

The task of identifying code anomalies at a scale of a language ecosystem consists in the following steps: (1) retrieve large corpora of source code and compiled machine code that are representative of the whole range of conventional coding practices and compiler output, respectively; (2) transform source code and machine code into a vectorized form that is digestible by anomaly detection algorithms; (3) run anomaly detection across vectorized data; (4) process the output of anomaly detection algorithms to identify, interpret, and classify meaningful anomalies.

Established industry standard languages, such as Java and C++, are not the most feasible targets for detecting anomalies at the scale of a language ecosystem. While large amounts of open source code in these languages are publicly available, mainstream compilers are very well-tested and stable, which makes it hard to identify unknown compiler issues through anomaly detection. At the same time, it is essential that the target language has a significant and diverse community of users.

Considering these arguments, we choose Kotlin [2] and its ecosystem as a target for this study. Since the language is relatively young, its compiler might still contain bugs and performance issues, which makes anomaly detection more likely to provide actionable insights for the language development team. At the same time, Kotlin is one of the most rapidly developing languages with an actively growing community and a significant ecosystem of diverse open source projects [3]. Finally, the development team of Kotlin is easy to reach via a public issue tracker [1], which makes it easy to communicate potential findings.

The contribution of this paper is threefold:

- A method for finding code anomalies for a chosen programming language that is based on vectorizing a large code corpus with code embedding techniques and then applying anomaly detection algorithms on this vector data.
- A set of tools implementing the proposed method as well as the dataset containing more than 4 million unique Kotlin functions collected from Github and the bytecode for a part of them (more than 41,000 compiled classes).
- The evaluation of the proposed method on the collected dataset, which resulted in discovering several dozens of code fragments that were considered useful by the Kotlin compiler

MSR 2020, 25–26 May 2020, Seoul, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

team and that were included into the Kotlin compiler test infrastructure.

The rest of the paper is organized as follows. In Section 2, we discuss existing studies and tools that apply the anomaly detection techniques to programs. Section 3 provides an overview of methods and techniques that are potentially applicable to finding code anomalies. Section 4 presents our proposed approach to anomaly detection, and describes the collected dataset as well as steps of the processing pipeline. Section 5 outlines our approach to evaluation of viability of the proposed method, as well as significance and practical value of the detected anomalies. Section 6 discusses possible threats to validity of our study. Section 7 presents the obtained results and describes the most interesting types of discovered anomalies. Section 8 concludes the paper by summarizing our results and providing possible directions for future work.

2 RELATED WORK

Several papers and tools exist that are aiming to search for anomalies in programs. All of them introduce their own definitions of code anomalies, so their goals, methods and results also differ.

The GrouMiner tool [26] was developed to detect anomalous patterns in object interaction in Java programs. The approach is based on modeling object interaction with a directed acyclic graph, the nodes of which are constructor and methods calls as well as fields references, while the edges represent dependencies between them. The tool performs a static code analysis: the source code is parsed into an abstract syntax tree and an object usage graph is built. Graph-based anomaly detection methods are used to detect unusual method calls and other atypical areas of the control flow graph.

A somewhat similar idea is presented in [40]: the authors propose mining usage models for all objects from source code as sequences of their method calls. If an abnormal usage pattern emerges in some code fragment, it is treated as an anomaly and a defect candidate. This approach is also based on static code analysis and employs graph-based anomaly detection techniques.

Undoubtedly, object interaction anomalies are important, but they represent only one possible anomaly type, and there could be many others, for example, atypical usage of some language constructs (not involving object interaction of any kind).

The DIDUCE tool [17] for Java programs is based on dynamic code analysis: it runs a program and stores values for each expression found in it. It tries to induce invariants for these expressions, starting from the strictest ones and weakening them as new values are encountered. When these rules are violated, meaning that some expression gets a value that significantly differs from all previous values of this expression, this is reported as an anomaly. Papers [34] and [14] also use dynamic code analysis but collect and analyze traces of system calls.

Several papers [24, 28] define code anomalies as "code smells" — specific code patterns indicating possible architectural flaws. An example of such pattern is a Feature Envy smell that arises when a method interacts with other classes more than with methods and fields of its own class. Finding such code fragments in a project usually helps to improve its design.

All these approaches are helpful if one is trying to find logical errors or architectural issues in programs and therefore are targeting programming language users, not its developers. It's also worth noting that dynamic analysis algorithms don't seem like a good fit for our task since we are looking at a potentially very large code base, and running all this code does not seem feasible. Projects might have all kinds of strange dependencies, and it would require a lot of human effort to understand how all of them are supposed to be compiled and run.

In the domain of programming language development, code anomalies, thanks to being atypical code, while having been actually implemented by someone, can be used as a source of data for fuzz testing techniques, which have proven to be valuable for finding tricky issues in compilers [41].

3 DETECTION OF CODE ANOMALIES

3.1 Anomaly detection techniques

In data mining, anomalies are defined as deviations of the observed behavior from the expected behavior and are divided into three types [9]:

(1) *Point anomalies* occur when a single data instance is considered abnormal compared to other data.
(2) *Contextual anomalies* occur when a single data object is anomalous in a specific context but not otherwise.
(3) *Collective anomalies* occur when linked objects are observed against other objects as an anomaly.

In this study, we focus solely on point anomalies, but other anomaly types are also worth investigating and might lead to interesting results.

Our task implies that a dataset is unlabeled and we have no examples of code anomalies, which leads us to unsupervised anomaly detection methods [18], such as Local Outlier Factor [7] and Isolation Forest [22, 23]. Several clustering methods [25] could be useful for our task. Clustering methods such as DBSCAN [13], ROCK [16], or SNN [12] do not require each object to be a part of a cluster, and we can see out-of-cluster objects as anomalies. One-class SVM [33] is a classification algorithm, but it assumes the presence of only one class, so the method could be used to detect outliers in unlabeled data.

Neural autoencoder [11] can also serve as an anomaly detection method. Autoencoder is tasked with reconstruction of a data point through an intermediate representation. After decoder is trained on a set of datapoints, we can identify anomalous points by measuring reconstruction loss, with the assumption that the loss will be higher for outliers.

Statistical methods [30] for anomaly detection measure the compliance of data to a specific probability distribution. The degree of anomalous behavior is the magnitude of the object's deviation from the distribution. For some methods in this group, an initial assumption about the distribution of data points is required, others may calculate a likely distribution that can generate the observed dataset.

3.2 Source code embedding

All of the anomaly detection algorithms, like most machine learning techniques, require embedding of analyzed objects in numeric vector space. To do so, first of all, we need to determine the level of structural units that will undergo analysis. It could vary from individual tokens and functions to files or even entire projects. Individual tokens and lines of code are highly dependent on their context and will not capture any significant anomalies. Functions are good candidates because most of them contain at least several lines of code that might form an anomaly and are isolated enough to represent a single operation within a class. Classes are also fitting for finding anomalies, especially in inheritance, properties use, class type parameters, etc. Using files as structural units would only allow us to detect anomalies in top-level constructions, of which there are not many and which rarely form anomalous code. Though analyzing entire projects as structural units of code may reveal some general patterns, these patterns will vary from one subject area to another and may not be informative.

Second, we need to define the level of code representation. It could be the source code itself (as text or token sets) or representations at different compilation stages. In case of Kotlin, we have access to following representations: a syntax tree; an intermediate representation for a specific compiler back-end (IR); JVM bytecode, LLVM bitcode or Javascript code generated by the compiler. Different representations require different embedding techniques.

We can treat source code as text, which will allow the use of various natural language processing (NLP) features, such as bag-of-words or N-grams [19, 39]. These features tend to capture the semantics of functions and variables names but tend to ignore the program's overall structure. NLP techniques could also be applied to bytecode since bytecode representation has a similar linear structure.

Representing a code fragment as a syntax tree introduces a wide range of embedding techniques divided into explicit and implicit approaches. Explicit approaches construct the vector from software metrics values [8, 27] capturing lexical, syntactical, architectural and other properties. The resulting vectors are quite easy to build and comprehend: observing these values may provide a clear understanding of why this particular code fragment is considered to be anomalous. However, it is always a challenge to choose which metrics to include (for example, [38] describes almost 300 metrics), especially when we are trying to search for anomalies of an unknown nature. Implicit approaches use features such as N-grams of token types, syntax trees hashing [10, 20], syntax trees encoding [29], the latent vector of a neural network autoencoder [37] and other distributed code representations [5]. These vectors usually lack in interpretability but have proven themselves to be capable of capturing complex code properties, including semantic dependencies.

Described methods are also applicable to other available representations of Kotlin source code mentioned before: a list of LLVM bitcode instructions, a Javascript syntax tree, and an intermediate representation (IR).

4 THE PROPOSED METHOD

In this study, we perform a search for code anomalies in Kotlin programs that are written specifically for the JVM, since it comprises the majority of code written in Kotlin. We analyze these programs in the form of syntax trees and sequences of bytecode instructions (other representations mentioned above are available only for Javascript and LLVM platforms).

Our goal is to detect two types of anomalies: syntax tree anomalies and compiler-induced anomalies. A *syntax tree anomaly* is a code fragment that is written in some way that is not typical for the programming language community. It could have abnormal complexity, be composed of sophisticated code constructs or in any other way differs from the rest of the code written in this language.

A *compiler-induced anomaly* is a code fragment that is not an anomaly in the syntax tree form but is an anomaly in the bytecode or vice versa. We choose to call them compiler-induced because their anomalous nature is revealed only after their bytecode is obtained and analyzed. We should note, though, that this wording does not imply any negative connotation: for instance, if the compiler does a good optimizing job and turns anomalous source code into non-anomalous bytecode, this is still an unexpected behavior and, therefore, an anomaly.

4.1 Dataset

4.1.1 Source code and syntax trees. To collect a large dataset, we cloned GitHub repositories that were created before March 2018, stated Kotlin as their main language, and were not forks of some other projects. That resulted in 47,751 repositories containing 932,548 source files with 4,044,790 unique functions. The collected source code was transformed into syntax trees using the parser module of the Kotlin compiler.

For the syntax trees code representation, we decided to use functions as code units. This allows us to simplify further analysis of detected anomalies, whether it is an expert assessment or a performance test. A function is an isolated unit of code, which is convenient if an anomalous fragment is to be analyzed by a compiler test because we only need to load the classes used in the function and to provide the input data it requires. Also, it will be easier for experts to analyze the function's code due to its isolation.

4.1.2 Bytecode. To detect compiler-induced anomalies we require both syntax tree and bytecode representations. GitHub supports publication of the project's builds packages which allows the direct collection of bytecode. Even though the dataset we collect in this way is relatively small, that kind of extraction approach is highly convenient because it allows us to collect the bytecode produced by its developers instead of figuring out the correct environment for each project and building them ourselves. We managed to obtain 41,226 compiled class files, which we then transformed into lists of JVM instructions.

Anomaly detection at functions level for bytecode representation is not as efficient as in the case of syntax trees because many Kotlin syntax constructions create additional code in bytecode representation. For example, lambdas (anonymous functions) are transformed into dummy functions. If an algorithm finds a bytecode anomaly related to such dummy function, it will be difficult to match a set

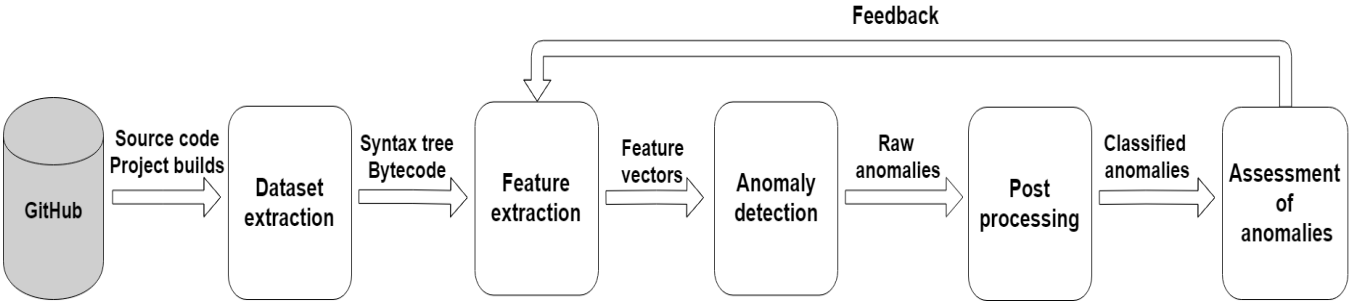

Figure 1: Overview of the proposed method for extraction and assessment of anomalies

of bytecode instructions to the source code. This directed us to choosing a class as a unit of evaluation for bytecode representation.

4.2 Anomaly detection pipeline

To detect and evaluate code anomalies, we propose the pipeline shown in Figure 1. It takes source code and project builds that are collected from GitHub as an input and converts them into syntax trees and JVM instructions lists using the dataset extraction module. Features are then extracted and anomaly detection techniques are applied. On the post-processing step, detected anomalies are classified according to their type. Finally, classes of anomalies are presented for expert evaluation. Then the pipeline's steps are adjusted for the next iteration with provided feedback.

The rest of this section discusses each step of anomaly detection in more detail. We describe assessment of individual anomalies in section 5.2.

4.2.1 Feature extraction. We use software metrics and N-grams extraction to embed the syntax tree representation of the code. For the bytecode representation, only implicit feature extraction via N-grams is used. The bytecode has a linear structure and is rarely analyzed by humans, and fewer metrics are known for it.

Software metrics are divided into 4 groups:

- general code metrics: the number of lines of code, the number of nodes and the height of the syntax tree, etc.
- structural metrics: nesting depth, cyclomatic complexity, number of branches in "when" expression, etc.
- external metrics of Kotlin functions: the formal arguments number, type parameters, annotations, the presence of a suspend modifier, etc.
- the number of particular language elements: expressions, operators, keywords, function calls, string patterns, etc.

Each function is encoded by a vector that contains the values of these metrics.

The N-grams extraction is performed as follows: the algorithm traverses a syntax tree and generates all unigrams, bigrams, ..., N-grams from connected nodes and adds a counter with value 1 for a new N-gram or increments the counter for an existing N-gram. For efficiency reasons, only parent-child relations are used to build N-grams, since considering all other relations (e.g., between sibling nodes) results in a vast number of N-grams, which will hinder the algorithm performance.

Both approaches result in a set of feature vectors. In the case of explicit representation, it is a k-dimensional vector where k is the number of metrics used. In the case of N-grams representation, it is a sparse vector where values represent counters of all N-grams met.

For compiler-induced anomalies detection, we have to process bytecode instructions sequences. N-grams are extracted incrementally from the sequence using a fixed size window. Within this window the extraction module generates all possible unigrams, bigrams, ..., N-grams, then moves the window one instruction ahead and repeats the procedure. Figure 2 presents an example of such extraction for $N = 3$.

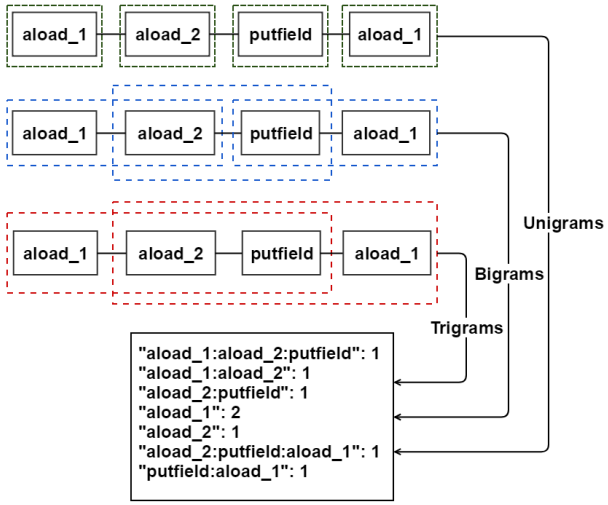

Figure 2: Extraction of N-grams from sequences of bytecode instructions

4.2.2 Anomaly detection. In the first phase of this study, to assess general viability of the idea of anomaly detection in syntax trees, we settled on relatively simple algorithms for anomaly detection — Local Outlier Factor [7] and Isolation Forest [22]. We chose these algorithms because they are relatively easy to implement and trial, compared to more complex approaches.

Local Outlier Factor is based on the idea of calculating an anomaly score for each object based on its distance to k nearest neighbors in the dataset. More precisely, the local density of a data point is considered: a coefficient of the point's reachability to its k nearest neighbors. The anomaly score of a point is calculated as the ratio of local density to the mean local density in the region of a point.

Isolation Forest employs the principle of random forest to separate outlier objects from the rest of the data. Each isolating tree is constructed in the following way: at the current node, the algorithm randomly selects a feature and a value to split the data into child nodes. The process continues until all elements are separated from each other. The normality measure for an object is introduced as the average length from the root to the object's position in all isolation trees. Thus, the earlier the object is separated from the sample by isolation trees, the smaller its measure of normality will be. Figure 3 shows an example of isolation trees. Anomalous objects are marked in black, while "normal" objects are marked in gray.

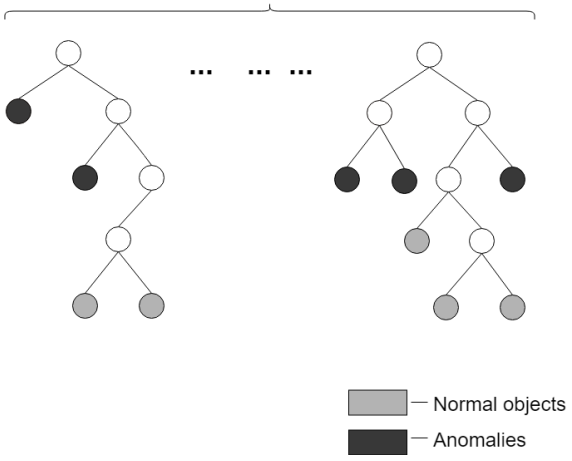

Figure 3: Detecting anomalies with Isolation Forest

Aforementioned algorithms are more efficient if the analyzed objects are represented by vectors of low dimensionality. When working with high-dimensional data, calculations might require too much time, hence it is necessary to fine-tune the algorithm parameters. For instance, a good choice of parameters for the Isolation Forest can sufficiently reduce processing time and memory usage without any noticeable changes in detection accuracy [23].

A popular approach to anomaly detection in high-dimensional datasets is a neural autoencoder [31], a popular type of neural networks. An autoencoder isn't so resource-intensive comparing to the Local Outlier Factor and Isolation Forest, which is crucial when we analyze the N-grams representation dataset since each feature vector contains thousands of values.

Figure 4 shows a basic autoencoder architecture that consists of input and output layers of the same size and a hidden layer that is significantly smaller. During the learning process, autoencoder trains to return the same vector as it receives as input. One of the results of this training is that we then have hidden layer values that comprise a vector representation (or embedding) of the input data. Another result is that if the trained autoencoder fails to reconstruct a data point, this data point may be an outlier to the dataset. Therefore, we can use the recovery error, i.e. the distance between input and output vectors as an anomaly score.

We propose the following as a way to detect compiler-induced anomalies: first, we calculate autoencoder anomaly scores both for syntax tree and bytecode representations of the same code fragment. If these scores differ by more than a specified threshold, we consider this fragment to be a compiler-induced anomaly.

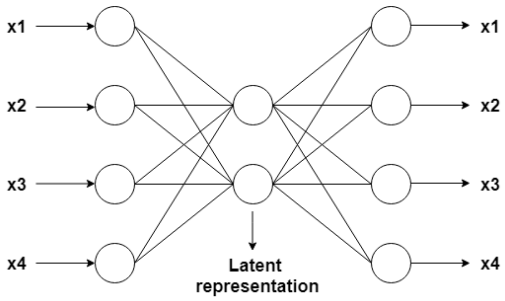

Figure 4: Architecture of a neural autoencoder

4.2.3 Anomalies classification. We grouped anomalies into classes for further evaluation. Each anomaly was manually labeled with a set of tags. We performed classification manually to get the most reliable result possible.

5 EVALUATION

To assess the viability of our approach to identification of anomalies, we conduct an evaluation of significance and practical value of anomalies extracted from the dataset. In this section, we describe the process of building the evaluation set (Section 5.1), our approach to assessment of importance of individual anomalies (Section 5.2), and the results of this assessment (Section 5.3).

5.1 Building the evaluation set

In this section we describe the process of building the set of anomalies for expert assessment.

5.1.1 Extracting anomalies from explicit source code representations. As the first step, to build explicit representation vectors for syntax trees in the dataset, we calculated 51 software metrics for each code fragment (49 quantitative and 2 binary). We scaled the quantitative attributes to the mean of 0 and the variance of 1, and reduced dimensionality of metric vectors from 51 to 20 via Principal Component Analysis. The number of principal components was chosen manually as a compromise between the time of training the models and the persistence of the explained variance (0.8).

We ran Local Outlier Factor and Isolation Forest outlier detection algorithms (Section 4.2.2) against the vectorized snippets. Both of these algorithms have multiple hyperparameters. The first notable parameter is a contamination parameter, influencing the fraction of code fragments that should be marked as potential anomalies. Since the aim of the study is to find a set of anomalies that could be reviewed by experts, we chose the contamination parameter so that the classifier marked 0.01% of the dataset, or approximately

400 out of 4 million collected functions. Specific parameter values were 0.0001 for Isolation Forest and 0.001 for Local Outlier Factor.

Other notable parameters of the outlier detection algorithms are the number of neighbors (*n_neighbors*) for the Local Outlier Factor, which we set to 20, and the number of trees built (*n_estimators*) for the Isolation Forest, which we set to 200. We settled on these values after several rounds of experiments and manual assessment of the resulting anomalies. As a result, these outlier detection algorithms have extracted 322 unique anomalies that we considered worthy of the attention of the Kotlin compiler team, from explicit representations of code snippets in the dataset.

5.1.2 Extracting anomalies from implicit representations. To build implicit representation vectors for both syntax trees and bytecode, we extracted unigrams, bigrams, and trigrams from individual instruction sets. The extraction yielded 1,708,022 unique N-grams for syntax trees and 110,835 unique N-grams for bytecode. Then we filtered rare and frequent N-grams, since they contain less information about the object, and ended up with 15982 N-grams for syntax trees and 4560 for bytecode.

Further, we trained an autoencoder to reconstruct N-gram vectors and optimized hyperparameters of the autoencoder by a heuristic search, taking into account limitations on memory use. Resulting parameters for the autoencoder are: number of epochs — 5, mini-batch size — 1024, compression rates — 0.25, 0.5, and 0.75.

To measure the *anomaly score* of individual objects, we calculated the Euclidean distance between the input and the output vectors of the autoencoder. We considered anomalies all vectors of syntax trees with anomaly score exceeding 3 root mean square distances. For the compiler-induced anomalies, the threshold difference between the syntax tree and bytecode anomaly scores was set to 0.8. These values were also heuristically evaluated over multiple experiments to yield a reasonable number of anomalies. Using three autoencoder networks with different compression rates, we have extracted 191 unique syntax tree anomalies and 54 unique compiler-induced anomalies.

5.1.3 Final processing of the evaluation set. Using both explicit representations of source code and implicit representations of source code and bytecode, we obtained a total of 375 unique anomalies. After a quick manual filtering to remove non-interpretable items, we presented 145 remaining anomalies to Kotlin language experts for assessment of their usefulness. We describe the motivation behind expert assessment and its technique in the next section.

5.2 Assessment of anomalies

5.2.1 Choice of assessment technique. Code anomalies could be evaluated in several ways: by measuring the performance of the compiler on the anomalous code, by measuring runtime performance of a compiled program that contains anomalous code, or through expert assessment.

However, there are several important concerns that make performance measurements difficult to use and interpret. We can only measure influence of syntax tree anomalies on the performance of the compiler's parser. It is worth noting that a parser is usually one of the simplest parts of a compiler, and performance problems in a compiler usually arise during other compilation stages: type inference, resolving, or code generation. Thus, running a performance evaluation of only a parser would be a shallow evaluation technique. At other compilation stages, performance of the compiler is highly dependent on the environment (e.g. the speed of type inference depends on the type parameters signature of the function that is called from an anomalous fragment). Moreover, a performance problem might be observed in one environment and not observed in another, and going through the complete variety of environments is an almost impossible task.

As for compiler-induced anomalies, even though we know the initial source code and the produced bytecode, it is problematic to evaluate the performance of the compiler. Data and computation that are related to a specific anomaly may be ordered differently within compiler's internal structure, which would make it very difficult to pinpoint the specific contribution of the anomalous code to the overall performance. However, it would be possible to evaluate performance of the program runtime, because we already have a corresponding bytecode to run.

A performance measurement would be relevant for assessment of the anomaly types that cause performance problems, but other anomaly types exist as well: for example, anomalies that highlight problems in the programming language design. Such problems can only be detected by an expert who deeply understands the concepts of the programming language and the internal structure of the compiler.

5.2.2 Expert assessment. To cover all the anomaly types, we decided to settle on expert evaluation for assessment of the anomalies. We have invited the development team of the Kotlin compiler to serve as experts in our study. The developers of the compiler have the most comprehensive understanding of the compiler's internals, and thus can be capable of attributing individual code constructs to concrete problems. Such problems include compiler performance, program runtime performance, and language design. Two people agreed to volunteer their services as experts.

We asked the experts to evaluate each anomaly, using a five-point scale from 1 to 5. Evaluation criteria addressed three aspects: (1) whether the code fragment is typical structurally, (2) whether the code fragment can become a valuable compiler test, and (3) whether the code fragment can cause compiler performance issues. The experts were asked to select the highest rank that will hold a true statement from the following list:

- *Rank 1*: very typical code; could not be used as a compiler test of any kind; does not cause performance issues;
- *Rank 2*: code with typical architecture patterns; highly unlikely to be used as a compiler test; highly unlikely to cause performance issues;
- *Rank 3*: mostly typical code with rare use of distinctive features; could be used as a compiler test, but fragments similar to this are already used in tests; Unlikely to cause performance issues;
- *Rank 4*: the code comprises atypical combinations of language features; could become a valuable compiler test, fragments similar to this are already used in tests; likely to cause performance issues;
- *Rank 5*: highly atypical code structurally; will be a unique compiler test; most likely will cause a performance issue.

The experts evaluated anomalies together, discussing each anomaly to conclusion and providing a collective ranking decision.

We also asked the experts to rank each of the anomaly types that we obtained on the post-processing step (Section 4.2.3) to assess the perceived importance of individual types. This ranking ensures that in future experiments we are able to tune our pipeline to produce more anomalies that are of interest to the experts.

5.3 Assessment results

Figure 5 presents the results of expert importance assessment for syntax tree anomalies. 38 out of 91 source code anomalies selected for evaluation were considered to be important, i.e., helpful, by the experts.

Table 1 presents assessment results for the anomaly types. The *Rank* column contains the expert's evaluation rank, and the *Size* column shows the number of detected anomalies of this type.

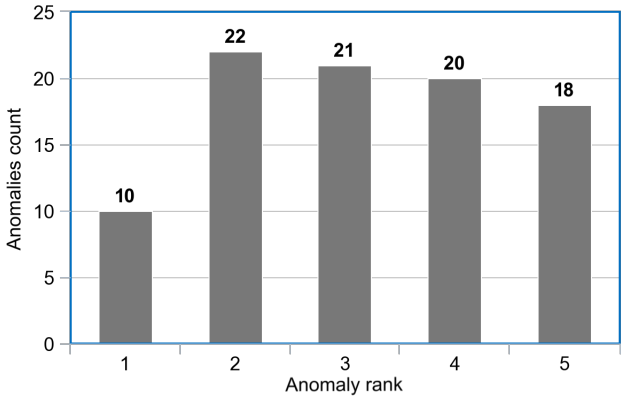

Figure 5: Expert evaluation of importance for the discovered syntax tree anomalies

Bytecode anomalies (54 out of 145) correspond to the properties of the compiler, rather than source code. It makes such anomalies difficult to classify, hence we did not assign class labels to them. Figure 6 presents the results of expert evaluations for compiler-induced anomalies. 31 out of 54 anomalies were considered helpful.

Based on the results of expert assessment, we have compared the assessment scores for syntax tree anomalies obtained from two experiments: the first one that used explicit vectors of metrics values and Local Outlier Factor/Isolation Forest (the explicit experiment) as well as the second one that used implicit representations based on N-grams obtained with an autoencoder (the implicit experiment). The comparison shows that anomalies from both experiments were rated high, and both approaches are suitable for code anomalies detection. On the one hand, the use of explicit code metrics simplifies the analysis of the resulting anomalies, and the set of these metrics can be easily extended to search for new anomaly types. On the other hand, implicit representations of code are able to capture properties that could be rather difficult, if possible, to describe with explicit code metrics.

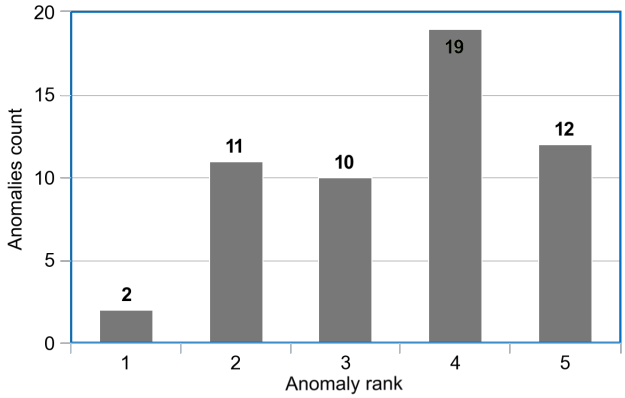

Figure 6: Expert evaluation of importance for the discovered compiler-induced anomalies

Table 1: Results of expert assessment of anomaly types, sorted by importance

	Anomaly type	Size	Rank
1	Delegates	1	5
2	Type arguments	8	5
3	"When" expression	35	5
4	Annotations	2	4
5	Call chains	5	4
6	Enumerations in "when"	2	4
7	"If" expressions	15	4
8	Nested calls	3	4
9	Similar call expressions	88	4
10	Strange code constructs	3	4
11	Assignments	57	3
12	Large methods	2	3
13	Code hierarchy	32	3
14	Function parameters	17	3
15	Multiline strings	27	3
16	"Try-catch" expressions	1	3
17	Arrays or maps	32	2
18	Class references	2	2
19	Concatenations	13	2
20	Lambdas	2	2
21	String literals	2	2
22	Logical expressions	6	2
23	Complex loops	8	2
24	Similar code fragments	2	2
25	"Throw" expressions	2	2
26	Assertions	1	1
27	Empty string literals	2	1
28	Local variables	2	1
29	Nested functions	2	1
30	Type casts	2	1

5.4 Data availability

The data that supports the findings of this study is openly available[1]. An anonymized distribution of the created tools for peer review is also available[2]. Upon publication of this paper, we will make the data and tools available in a GitHub repository and in persistent scientific archival services.

6 THREATS TO VALIDITY

Our study is subject to several threats to validity and generalizability.

External validity. Our dataset was comprised of only open source projects available on GitHub. This limits our ability to claim that our list of discovered anomalies is exhaustive, as code of proprietary projects might contain other anomalies.

Our anomaly detection pipeline was targeted at the Kotlin ecosystem from the very start of the project. We were able to obtain our results thanks to a combination of several unique factors, notably including quick growth of the Kotlin open source community, our ability to get in contact with the compiler development team, and slightly lower maturity of the compiler, compared to other more established languages and ecosystems. Considering these unique factors, we should acknowledge that obtaining similar results for other programming languages would require building a potentially sophisticated method from scratch.

Construct validity. Our resulting list of anomalies is ultimately influenced by our choices regarding the set of metrics, methods for anomaly detection, and parameters of the algorithms.

Bias of experts. Our ranking of importance for individual anomaly types is based on estimates provided by two experts. Such quantitative estimates based on expert opinion are intrinsically subjective [21]. Moreover, the role of individual biases in such a small group is particularly high. Finally, our experts are the developers of Kotlin, who are deeply involved with the compiler internals; thus, their subjective perception of importance might differ from the broader group of interested public — for example, from professional Kotlin developers.

We believe that these threats, while worth noting, do not invalidate our results or diminish their value. We do not claim that our methodology is universal or applicable to other languages. Instead, with this study we unravel a new scope for application of anomaly detection techniques in software engineering, and demonstrate that these techniques help to provide actionable insights to language developers.

7 DISCUSSION

7.1 Syntax tree anomalies

Syntax tree anomalies are code fragments that are not typical in the Kotlin community. They are either very rare combinations of code constructs, certain combinations of code constructs repeated a large number of times, or functions with atypical characteristics, like very large object hierarchy or extensive domain-specific constructs (Figure 7).

[1]The dataset used in this study: https://bit.ly/2txCL5q

[2] The tools supporting the proposed methods: https://bit.ly/2Sixau4

The detected syntax tree anomalies were divided into three categories:

(1) **Anomalies that correspond to language design issues**. Such anomalies were noted and archived by the Kotlin compiler team for further discussion and research. Later such discussions could lead, for example, to introduction of a new construct to the language, deprecation of constructs use in certain cases, extension of the language standard library, refinement of the reference documentation or the language specification. Thus, the detected anomalies will be able to play a role in the development of the programming language.
(2) **Anomalies that correspond to cases when some part of the compiler might work incorrectly or slowly**. The experts considered such anomalies as valuable compiler tests, and accepted to include them in the compiler's testing infrastructure. They can be used as performance tests for the compilation stages of type inference and resolving, as type inference tests for correctness, or as tests for generated bytecode. For example, the experts noted that some anomalies highlight various corner cases of the type inference as well as data flow analysis and can be very useful during refactorings in these compiler modules. An example of such an anomaly is presented in Figure 8. This function can act as a good test for the performance and correctness of the compiler type inference module, since processing it involves non-trivial analysis and type inference.
(3) **Anomalies that correspond to potential performance issues of the program's runtime environment**. In such cases, there may also be some issues with the compiler. For example, non-optimal code generation, failures, or lack of some optimizations. Such code fragments can also be used as compiler tests; more precisely, as performance tests for the compiled programs.

7.2 Compiler-induced anomalies

Compiler-induced anomalies correspond to cases when a complex or atypical bytecode was generated by a simple or "normal" syntax tree (or vice versa, a non-anomalous bytecode was produced from an anomalous syntax tree).

The detected compiler-induced anomalies were caused by the following issues:

(1) **Non-optimal code generation cases in the compiler that manifested in some rather exotic code examples**. In such cases, the anomaly can be used as a test for bytecode generation.
(2) **Rather complex functions were inlined in code fragments that were considered abnormal**. Such cases have nothing to do with compiler problems, but could be very useful for developers of libraries and frameworks, because their API is available to a potentially large number of programmers. Detection of function inlining problems could also help regular developers (i.e. users of the language) to fix perfomance issues in their projects. Figure 9 presents an example of such an anomaly containing problematic function inlining: the framework developers wrote a large and complex function to bind properties with configuration data, which

```
"INTERPRET" compose {
  label("loop") // empty stack
  "BL WORD"()
  "DUP COUNT NIP"()
  cbranch("end") // name
  "FIND"()
  "DUP"()
  cbranch("word_not_found") // wptr flags
  "STATE @ 0="()
  cbranch("compile") // wptr flags
  "DUP -1 <> SWAP 2 AND AND"()
  cbranch("compile_only") // wptr
  "$msgCompileOnly COUNT TYPE HERE COUNT TYPE"() // this "HERE" is cheating, this word needs to be refactored anyways
  "ABORT"()
```

Figure 7: An example of atypical, yet valid, code from a Forth implementation written in Kotlin.

was marked “inline”; therefore, its bytecode was copied to all places where it was called. In this example, the “bind” function is only called 9 times, but the bytecode is already very large and complex. This code fragment was not considered a syntax tree anomaly, but its bytecode was marked as anomalous, which helped to catch this code fragment.

Among the suspicious code fragments there were several cases when some of the compiler optimizations worked well, and complex syntax trees were minimized to form a rather simple bytecode. Surely, such cases are not issues of any kind, but they can nevertheless be useful to compiler developers: for example, they can learn from the successful optimizations and further improve them.

8 CONCLUSION

In this paper we present two kinds of experiments to detect different types of code anomalies (syntax tree anomalies and compiler-induced anomalies), using different approaches both at the code vectorization and at the anomaly detection stages. As a result, 91 syntax tree anomalies and 54 compiler-induced anomalies were presented for assessment of importance to the Kotlin compiler developers. According to the results of the assessment, 32 syntax tree anomalies and 31 compiler-induced anomalies were considered useful (they got rank values 4 and 5). Some of these anomalies were added into the compiler testing infrastructure as performance and correctness tests for the compiler front-end and back-end; several anomalies were postponed for further discussion on possible language design issues. Based on these results and further discussions with the Kotlin compiler developers, we believe that the detected anomalies are useful and valuable for the language development, and our proposed approach proved itself successful in detection of various code anomalies.

We outline several directions for future research:

(1) The post-processing step of the pipeline could be automated (at least to some extent) to provide tools for classification and labeling of the found anomalies. This would allow us to focus more on finding new and more interesting types of anomalies.
(2) The bytecode dataset could be increased (for example, by creating tools that automatically build projects from GitHub). At the moment, only a small part of the repositories publish builds of their projects in the “Releases” GitHub section. Automatic build tools would provide us with bytecode for all projects with syntactically and semantically correct code. With such dataset at hand, we could identify much more compiler-induced anomalies.
(3) Compiler-induced anomalies could be used to evaluate the impact of new language features on the compilation process. This could be implemented as tests that track anomaly scores of some code examples during the compiler development process. If the anomaly scores change too much after a specific change in the compiler code, then such changes should be analyzed in detail and it should be understood what led to such an effect.

```
@Suppress("UNCHECKED_CAST")
fun <V1, V2, V3, V4, V5, V6, V7, V8, V9, V10,
     V11, V12, V13, V14, V15, V16, V17, V18, V19, V20, E>
concreteCombine(
       p1: Promise<V1, E>, p2: Promise<V2, E>, p3: Promise<V3, E>, p4: Promise<V4, E>,
       p5: Promise<V5, E>, p6: Promise<V6, E>, p7: Promise<V7, E>, p8: Promise<V8, E>,
       p9: Promise<V9, E>, p10: Promise<V10, E>, p11: Promise<V11, E>, p12: Promise<V12, E>,
       p13: Promise<V13, E>, p14: Promise<V14, E>, p15: Promise<V15, E>, p16: Promise<V16, E>,
       p17: Promise<V17, E>, p18: Promise<V18, E>, p19: Promise<V19, E>, p20: Promise<V20, E>
)
: Promise<Tuple20<V1, V2, V3, V4, V5, V6, V7, V8, V9, V10, V11, V12, V13, V14, V15, V16, V17, V18, V19, V20>, E> {
    val deferred = deferred<Tuple20<V1, V2, V3, V4, V5, V6, V7, V8, V9, V10,
                                    V11, V12, V13, V14, V15, V16, V17, V18, V19, V20>, E>()

    val results = AtomicReferenceArray<Any?>(20)
    val successCount = AtomicInteger(20)

    fun createTuple(): Tuple20<V1, V2, V3, V4, V5, V6, V7, V8, V9, V10,
                               V11, V12, V13, V14, V15, V16, V17, V18, V19, V20> {
        return Tuple20(
                results.get(0) as V1,
                results.get(1) as V2,
                (and 19 similar expressions)
```

Figure 8: An example of a syntax tree anomaly used as a test for performance and correctness of type inference in the compiler

```
class UserConfigModel(val userConfig: UserConfig) : ViewModel() {
    val user = bind { configs.user }
    val email = bind { configs.userEmail }
    val userRegDate = bind { configs.userRegDate }
    val password = bind { configs.passwordProperty }
    val databaseConfig = bind { configs.databaseConfig }
    val userLogDate = bind { configs.userLogDate }
    val hasEditAccess = bind { configs.hasEditAccess }
    val hasReadAccess = bind { configs.hasReadAccess }
    val isBlocked = bind { configs.isBlocked }
}
```

```
{
  "getUser" : [ "aload_0", "getfield", "areturn" ],
  "getEmail" : [ "aload_0", "getfield", "areturn" ],
  "getUserRegDate" : [ "aload_0", "getfield", "areturn" ],
  "getPassword" : [ "aload_0", "getfield", "areturn" ],
  "getDatabaseConfig" : [ "aload_0", "getfield", "areturn" ],
  "getUserLogDate" : [ "aload_0", "getfield", "areturn" ],
  "getHasEditAccess" : [ "aload_0", "getfield", "areturn" ],
  "getHasReadAccess" : [ "aload_0", "getfield", "areturn" ],
  "getIsBlocked" : [ "aload_0", "getfield", "areturn" ],
  "getUserConfig" : [ "aload_0", "getfield", "areturn" ],
  "<init>" : [ "aload_1", "ldc", "invokestatic", "aload_0", ... (4428 instructions)
}
```

Figure 9: An example of a compiler-induced anomaly highlighing an issue with function inlining

REFERENCES

[1] 2019. Kotlin (KT) – Bug and Issue Tracker. https://youtrack.jetbrains.com/issues/KT. Accessed: 2019-08-18.

[2] 2019. Kotlin Programming Language. https://kotlinlang.org/. Accessed: 2019-08-18.

[3] 2019. The State of the Octoverse: top programming languages of 2018 – The GitHub Blog. https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/. Accessed: 2019-08-18.

[4] Mohiuddin Ahmed, Abdun Naser Mahmood, and Md. Rafiqul Islam. 2016. A Survey of Anomaly Detection Techniques in Financial Domain. *Future Gener. Comput. Syst.* 55, C (Feb. 2016), 278–288. https://doi.org/10.1016/j.future.2015.01.001

[5] Miltiadis Allamanis, Earl Barr, Premkumar Devanbu, and Charles Sutton. 2017. A Survey of Machine Learning for Big Code and Naturalness. *Comput. Surveys* 51 (09 2017). https://doi.org/10.1145/3212695

[6] Christoph Baur, Benedikt Wiestler, Shadi Albarqouni, and Nassir Navab. 2018. Deep autoencoding models for unsupervised anomaly segmentation in brain mr images. In *International MICCAI Brainlesion Workshop*. Springer, 161–169.

[7] Markus M. Breunig, Hans-Peter Kriegel, Raymond T. Ng, and Jörg Sander. 2000. LOF: Identifying Density-based Local Outliers. *SIGMOD Rec.* 29, 2 (May 2000), 93–104. https://doi.org/10.1145/335191.335388

[8] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*. 255–270.

[9] Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly Detection: A Survey. *ACM Comput. Surv.* 41, 3, Article 15 (July 2009), 58 pages. https://doi.org/10.1145/1541880.1541882

[10] Michel Chilowicz, Etienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 243–247.

[11] Hoang Anh Dau, Vic Ciesielski, and Andy Song. 2014. Anomaly detection using replicator neural networks trained on examples of one class. In *Asia-Pacific Conference on Simulated Evolution and Learning*. Springer, 311–322.

[12] Levent Ertoz, Michael Steinbach, and Vipin Kumar. 2002. A new shared nearest neighbor clustering algorithm and its applications. In *Workshop on clustering high dimensional data and its applications at 2nd SIAM international conference on data mining*. 105–115.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise.. In *Kdd*, Vol. 96. 226–231.

[14] Henry Hanping Feng, Oleg M Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. 2003. Anomaly detection using call stack information. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE, 62–75.

[15] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis. In *ICDM 2009, The Ninth IEEE International Conference on Data Mining, Miami, Florida, USA, 6-9 December 2009*. 149–158. https://doi.org/10.1109/ICDM.2009.60

[16] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. 1999. ROCK: A robust clustering algorithm for categorical attributes. In *Data Engineering, 1999. Proceedings., 15th International Conference on*. IEEE, 512–521.

[17] Sudheendra Hangal and Monica S. Lam. 2002. Tracking Down Software Bugs Using Automatic Anomaly Detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*. ACM, New York, NY, USA, 291–301. https://doi.org/10.1145/581339.581377

[18] Victoria Hodge and Jim Austin. 2004. A survey of outlier detection methodologies. *Artificial intelligence review* 22, 2 (2004), 85–126.

[19] Chun-Hung Hsiao, Michael Cafarella, and Satish Narayanasamy. 2014. Using Web Corpus Statistics for Program Analysis. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 49–65. https://doi.org/10.1145/2660193.2660226

[20] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 96–105.

[21] Barbara Ann Kitchenham. 1996. Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods. *ACM SIGSOFT Software Engineering Notes* 21, 1 (1996), 11–14.

[22] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2008. Isolation Forest. In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining (ICDM '08)*. IEEE Computer Society, Washington, DC, USA, 413–422. https://doi.org/10.1109/ICDM.2008.17

[23] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. 2012. Isolation-Based Anomaly Detection. *ACM Trans. Knowl. Discov. Data* 6, 1, Article 3 (March 2012), 39 pages. https://doi.org/10.1145/2133360.2133363

[24] I. Macia, R. Arcoverde, A. Garcia, C. Chavez, and A. von Staa. 2012. On the Relevance of Code Anomalies for Identifying Architecture Degradation Symptoms. In *2012 16th European Conference on Software Maintenance and Reengineering*. 277–286. https://doi.org/10.1109/CSMR.2012.35

[25] Oded Maimon and Lior Rokach. 2009. Introduction to knowledge discovery and data mining. In *Data Mining and Knowledge Discovery Handbook*. Springer, 1–15.

[26] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. ACM, New York, NY, USA, 383–392. https://doi.org/10.1145/1595696.1595767

[27] Alberto S. Nuez-Varela, Hctor G. Prez-Gonzalez, Francisco E. Martnez-Perez, and Carlos Soubervielle-Montalvo. 2017. Source Code Metrics. *J. Syst. Softw.* 128, C (June 2017), 164–197. https://doi.org/10.1016/j.jss.2017.03.044

[28] Willian N. Oizumi, Alessandro F. Garcia, Thelma E. Colanzi, Manuele Ferreira, and Arndt V. Staa. 2015. On the relationship of code-anomaly agglomerations and architectural problems. *Journal of Software Engineering Research and Development* 3, 1 (10 Jul 2015), 11. https://doi.org/10.1186/s40411-015-0025-y

[29] Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*. Springer, 547–553.

[30] YA Siva Prasad and G Rama Krishna. 2013. Statistical Anomaly Detection Technique for Real Time Datasets. *International Journal of Computer Trends and Technology (IJCTT)–volume* 6 (2013).

[31] Chong Zhou Randy C. Paffenroth. 2017. Anomaly Detection with Robust Deep Autoencoders. In *KDD '17 Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '17)*. ACM, New York, NY, USA, 665–674. https://doi.org/10.1145/3097983.3098052

[32] Bartley D Richardson, Benjamin J Radford, Shawn E Davis, Keegan Hines, and David Pekarek. 2018. Anomaly Detection in Cyber Network Data Using a Cyber Language Approach. *arXiv preprint arXiv:1808.10742* (2018).

[33] Bernhard Schölkopf, Robert Williamson, Alex Smola, John Shawe-Taylor, and John Platt. 1999. Support Vector Method for Novelty Detection. In *Proceedings of the 12th International Conference on Neural Information Processing Systems (NIPS'99)*. MIT Press, Cambridge, MA, USA, 582–588. http://dl.acm.org/citation.cfm?id=3009657.3009740

[34] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. 2001. A fast automaton-based method for detecting anomalous program behaviors. In *sp*. IEEE, 0144.

[35] Salvatore J Stolfo, Shlomo Hershkop, Linh H Bui, Ryan Ferster, and Ke Wang. 2005. Anomaly detection in computer security and an application to file system accesses. In *International Symposium on Methodologies for Intelligent Systems*. Springer, 14–28.

[36] R. N. Taylor and L. J. Osterweil. 1980. Anomaly Detection in Concurrent Software by Static Data Flow Analysis. *IEEE Transactions on Software Engineering* SE-6, 3 (May 1980), 265–278. https://doi.org/10.1109/TSE.1980.234488

[37] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 542–553. https://doi.org/10.1145/3196398.3196431

[38] Alberto Varela, Hector Perez-Gonzalez, Francisco Martinez, and Carlos Soubervielle-Montalvo. 2017. Source Code Metrics: A Systematic Mapping Study. *Journal of Systems and Software* 128 (04 2017). https://doi.org/10.1016/j.jss.2017.03.044

[39] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: Bug Detection with N-gram Language Models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 708–719. https://doi.org/10.1145/2970276.2970341

[40] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC-FSE '07)*. ACM, New York, NY, USA, 35–44. https://doi.org/10.1145/1287624.1287632

[41] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 283–294.