Detecting Argument Selection Defects

ANDREW RICE, University of Cambridge, England and Google Inc., USA
EDWARD AFTANDILIAN, CIERA JASPAN, and EMILY JOHNSTON, Google Inc., USA
MICHAEL PRADEL, TU Darmstadt, Germany
YULISSA ARROYO-PAREDES, Barnard College of Columbia University, USA

Identifier names are often used by developers to convey additional information about the meaning of a program over and above the semantics of the programming language itself. We present an algorithm that uses this information to detect argument selection defects, in which the programmer has chosen the wrong argument to a method call in Java programs. We evaluate our algorithm at Google on 200 million lines of internal code and 10 million lines of predominantly open-source external code and find defects even in large, mature projects such as OpenJDK, ASM, and the MySQL JDBC. The precision and recall of the algorithm vary depending on a sensitivity threshold. Higher thresholds increase precision, giving a true positive rate of 85%, reporting 459 true positives and 78 false positives. Lower thresholds increase recall but lower the true positive rate, reporting 2,060 true positives and 1,207 false positives. We show that this is an order of magnitude improvement on previous approaches. By analyzing the defects found, we are able to quantify best practice advice for API design and show that the probability of an argument selection defect increases markedly when methods have more than five arguments.

CCS Concepts: • **Software and its engineering → Software defect analysis**; *Automated static analysis*;

Additional Key Words and Phrases: empirical study, name-based program analysis, static analysis, method arguments

ACM Reference Format:
Andrew Rice, Edward Aftandilian, Ciera Jaspan, Emily Johnston, Michael Pradel, and Yulissa Arroyo-Paredes. 2017. Detecting Argument Selection Defects. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 104 (October 2017), 22 pages. https://doi.org/10.1145/3133928

1 INTRODUCTION

Identifier names are often used by developers to convey additional information about the meaning of a program over and above the semantics of the programming language itself. Recent research has shown that this information is useful for various program analyses that address common software engineering tasks, such as defect detection [Pradel and Gross 2011, 2013], code recommendation [Liu et al. 2016], recommending identifier names [Allamanis et al. 2015], and enforcing coding conventions [Allamanis et al. 2014]. Analyses that compare identifier names of arguments passed to a method and formal parameter names in method declarations have been used to detect programming errors, such as accidentally interchanging arguments of the same type [Pradel and Gross 2013] or using a type-compatible yet incorrect argument [Liu et al. 2016].

Corresponding authors' addresses: A. Rice, Computer Laboratory, University of Cambridge, 15 JJ Thomson Avenue, CB3 0FD, UK; E. Aftandilian, Google, Inc., 1600 Amphitheatre Parkway, Mountain View, CA 94043, USA.
Authors' addresses: Andrew Rice, Computer Laboratory, University of Cambridge, England , Google Inc. USA; Edward Aftandilian; Ciera Jaspan; Emily Johnston, Google Inc. Mountain View, California, USA; Michael Pradel, TU Darmstadt, Germany; Yulissa Arroyo-Paredes, Barnard College of Columbia University, New York, New York, USA.

2475-1421/2017/10-ART104
https://doi.org/10.1145/3133928

As a motivating example, consider the following two code snippets. Both are sanitized versions of real defects discovered in code developed at Google:

```
private User getUser(String companyId, String userId) { ... }

public void doSomethingWithUser(String companyId, String userId) {
  User user = getUser(userId, companyId);
  ...
}
```

The call to `getUser` has swapped the arguments, which are both Strings. A simple check of whether the argument names match the parameter names would have prevented this defect. The defect caused a surprising `UserDoesNotExist` error when calling the `doSomethingWithUser` method.

Consider a second example:

```
class JanitorFn {
  static Value configure(String fileUser, ...) { ... }
}

public class KeywordSink {
  private String fileUser;

  public Value write(String stageName, ...) {
    ...
    return JanitorFn.configure(stageName, ...);
  }
}
```

The call to `JanitorFn.configure` passes `stageName` as the first argument; however, the correct argument should have been the `fileUser` field from the enclosing class, which matches the parameter name in the method declaration.

These are examples of *argument selection defects*. When a programmer makes a method call, each argument they choose is drawn from an implicit set of candidate expressions. These might be local variables, fields from the enclosing class (and perhaps the superclass), method calls on local variables, and so on. We define an argument selection defect as occurring on arguments at call sites which satisfy the following conditions: a) the programmer has selected an incorrect argument; and b) the set of candidate expressions contains the correct argument.

We first present a study of 84 real argument selection defects that have been previously found and fixed in Google's codebase (Section 3). This kind of error is rare, with fixes occurring at a rate of approximately 1 in every 1.5 million changesets. However, 7 of the 11 bug reports associated with these fixes were associated with high severity issues, so analysis tools capable of detecting these defects would have a notable benefit. We then describe our defect detection algorithm (Section 5) and report on its effectiveness (Section 6). With a reasonable sensitivity setting we were able to achieve a true positive rate of 85% whilst detecting 459 true defects, including some in mature software libraries such as `java.util.concurrent`. We found 2,305 defects overall and analyze these to quantify existing best practice advice on API design (Section 7). We compare against two different approaches: Order [Pradel and Gross 2013] and Nomen [Liu et al. 2016], and show that our algorithm produces an order of magnitude improvement in precision (Section 9).

The contributions of this paper are:

- evidence that argument selection defects do occur in deployed production systems and can cause high severity issues
- a new algorithm for argument defect detection which is an order of magnitude better than existing approaches
- a summary of the 2,305 defects we discovered and their implications for API design
- an implementation of this algorithm as part of the open-source Google Error Prone project [Aftandilian et al. 2012]

We perform our evaluation on 200 million lines of internal code and 10 million lines of external code and show that we achieve comparable performance on both. Our work is the first to deploy, and report on, a name-based analysis at very large scale. We believe the evidence here makes a strong case for general deployment of this style of analysis.

2 RELATED WORK

2.1 Name-Based Analysis

This paper builds on earlier versions of name-based static analyses of arguments passed to methods, we refer to these techniques as Nomen [Liu et al. 2016] and Order [Pradel and Gross 2011, 2013]. We compare our work to these techniques in detail later in the paper (Section 9). The IntelliJ IDE[1] compares argument names and parameter names to flag call sites that pass unexpected arguments. This check is based on a small, built-in set of commonly-confused identifier names, such as `width` and `height`. In contrast, our analysis considers arbitrary identifier names and does not require any built-in knowledge about common mistakes.

The observation that source code has properties similar to natural language documents [Hindle et al. 2012] has motivated several NLP-based approaches, e.g., for learning coding conventions [Allamanis et al. 2014] and coding idioms [Allamanis and Sutton 2014]. Høst and Østvold [2009] and Allamanis et al. [2015] address the problem of finding unusual names of methods and classes and of suggesting more appropriate names. In contrast, our work focuses on errors caused by using incorrect method arguments.

The importance of identifier names has been studied empirically, showing the importance of function identifiers [Caprile and Tonella 1999], that poor names correlate with poor code quality [Butler et al. 2010], and that descriptive and long names improve code comprehension [Lawrie et al. 2007].

An important component of many name-based analyses is tokenization of identifier names. Enslen et al. [2009] propose to infer a tokenizer via source code mining. Work by Butler et al. [2011] considers tokenization beyond the commonly used camel case and underscore conventions. These techniques could be easily combined with our current approach, which splits identifiers based on the camel case and underscore conventions. Cohen et al. [2003] survey several similarity measures for entity names and compare them with each other empirically. Our work also considers several distance functions (Section 5.1), which we evaluate specifically for the purpose of identifier name matching using a labelled dataset of ground-truth matches.

An approach orthogonal to detecting incorrect arguments once the code has been written is automated recommendation of arguments during development, e.g., as part of IDE-based code recommendation. Existing approaches include mining of clients of a particular API [Zhang et al. 2012] and code completion via inferred statistical language models [Nguyen et al. 2013; Raychev et al. 2014].

[1] https://www.jetbrains.com/idea/

2.2 Rule Mining Approaches

Name-based analysis techniques use the convention (or rule) of name agreement in programs. There are various techniques for finding bugs by mining other rules or specifications from a large codebase and by searching for violations of these mined rules. PR-Miner [Li and Zhou 2005] mined rules of the form "when calling functions f and g, you should also call function h." AntMiner [Liang et al. 2016] improves this idea by slicing functions before the rule mining. Chang et al. [2007] use frequent subgraph mining on program dependence graphs to find missing condition checks. Other approaches consider the order of method call sequences, e.g., by mining graph-based rules for method call sequences [Nguyen et al. 2009], sequence association rules that encode how to handle exceptions [Thummalapenta and Xie 2009], object usage models expressed as finite state machines [Wasylkowski and Zeller 2009; Wasylkowski et al. 2007], or multi-object specifications [Pradel and Gross 2009; Pradel et al. 2012]. Wang et al. [2016] learn an n-gram-based model of token sequences and use it for detecting anomalies. Deployment is more of a challenge with these approaches since they learn specific features from the codebase they are trained upon and therefore need periodic retraining. In contrast, name-based analysis uses a simple, general rule and so can be applied locally to the call sites of a single method at a time without reference to the rest of the code.

2.3 Deploying Automated Bug Detection Techniques in Industry

Other automated bug detection techniques have been deployed in industry. Our work is the first to deploy a name-based analysis at a very large scale.

Bessey et al. [2010] report their experience from commercializing a static bug detection tool. They share the insight that a scalable detection algorithm and a low false positives rate are crucial for real-world applicability. Our work provides additional lessons learned both related to the nature of the analysis, e.g., the need to know identifier names, and to the considered codebase, e.g., the prevalence of generated code.

FindBugs has been deployed at large scale and found several thousands of issues [Ayewah et al. 2008]. Checks in FindBugs are written to target a specific issue and so achieve high precision. In contrast, the analysis considered here searches for a much more general form of error, so it is harder to reason about how well it will perform. We tackle this question in this paper through thorough evaluation at scale.

A recent article describes the infrastructure used to store and manage the codebase at Google [Potvin and Levenberg 2016]. Our focus on local analysis is particularly relevant to the ultra-large-scale infrastructure described in their paper. Our analysis is implemented using Google's Error Prone static analysis tool [Aftandilian et al. 2012] and is integrated into the code review process using Tricorder [Sadowski et al. 2015], a general program analysis platform that addresses challenges related to scaling static analyses to a huge codebase. See Section 6.1 for more information.

3 ARGUMENT SELECTION DEFECTS IN THE WILD

To understand whether and what kinds of argument-related bugs exist in a professionally developed codebase, we searched Google's revision history for changes related to incorrect arguments. To this end, we searched commit messages with combinations of terms [argument, parameter], [order, swap, incorrect, wrong] and manually inspected the results. This process yielded a total of 84 argument-related defects that were fixed in 23 revisions. Revisions fixing argument related bugs occurred approximately once in every 1.5 million revisions. Figure 1 shows a representative selection of (partially sanitized) bugs. We acknowledge that this methodology misses an indeterminate number of this kind of bug, but it does show that such bugs occur in production code. Most importantly, we

establish that these bugs are not only latent issues present in dead or unimportant code; they can represent actual defects that developers do (eventually) fix.

We further summarize these 84 defects to answer several questions about the kind of argument-related anomalies that occur in the wild.

- *How long do argument-related bugs remain in the codebase before being discovered?* An argument-related bug remains in the codebase for a **mean of 66 days and a median of 4 days**. 45% of such bugs were fixed within 1 day. One explanation for this is that these defects are being detected by post-submission testing. An automated detection technique would be valuable here since it would catch these defects earlier in the development process, which is a considerable cost saving [Shull et al. 2002].
- *Do argument-related bugs correspond to severe issues?* In our data set, 16 of the 84 fixed anomalies were tagged with 11 related bug reports, and **7 of those 11 were marked as "high priority"** by developers. One was at the highest priority; company policy is that the team should drop everything and immediately address the bug. It was fixed within an hour of the report.
- *Are most argument-related bugs the result of swapping multiple arguments or of using a single incorrect argument?* We find that 79 of the 84 bugs are the result of accidentally permuting two or more arguments. In contrast, only 5 bugs result from accidentally passing a single incorrect argument that should be replaced by another expression available in the current scope. Figure 1 summarizes information from a selection of the bugs that we found.

4 SOFTWARE CORPORA

Having established that argument selection defects are important to production systems, we now develop and evaluate a suitable technique for detecting them. We do this with reference to two software corpora obtained by dividing our source code into two sections: the internal corpus consists of approximately 200 million lines of Java code developed at Google, and the external corpus consists of approximately 10 million lines of code developed externally. Some code in the external corpus is proprietary, so we cannot go into full details, but this corpus is predominantly open-source and includes popular, well-studied programs such as Apache Commons[2], Apache Tomcat[3], Eclipse[4], Mockito[5], and OpenJDK[6]. Figure 2 shows the distribution of the lengths of argument names and parameter names from the external corpus (top row) and the internal corpus (bottom row).

We observe from Figure 2 that the two codebases differ significantly in their use of identifier names, with internal code having longer names, in particular for parameters. We attribute this difference to the style guide at Google, which recommends avoiding names such as `temp`, `var` or `data`, that do not aid the reader's understanding, and avoiding visually similar names (e.g. `limit` and `limits`) to prevent accidental confusion. Figure 2 shows evidence of this policy: we see far fewer single character names and many more long names in the internal corpus than in the external corpus.

[2] https://commons.apache.org/
[3] https://tomcat.apache.org/
[4] http://www.eclipse.org/
[5] http://site.mockito.org/
[6] http://openjdk.java.net/

ID	Type	Parameter	Original argument	Correct argument	Days in repo	Comment
1	Duration	responseTTLDuration	frequencyCapDuration	responseTTLDuration	11	Discovered because of request to test an existing feature.
	Duration	frequencyCapDuration	responseTTLDuration	frequencyCapDuration		
	List<A>	slotResponse	slotResponse	slotResponse		
2	long	communityId	a.toDataObject().getId()	a.toDataObject().getId()	14	Many arguments with general-purpose types.
	long	senderId	e.getSenderId()	e.getSenderId()		
	long	recipientId	e.getRecipientId()	e.getRecipientId()		
	long	subject	subject	subject		
	String	textContent	htmlContent	textContent		
	String	htmlContent	textContent	htmlContent		
3	User	owner	owner	owner	792	Highest priority bug, fixed within 1 hour of filing. Called method is overloaded in many different ways.
	long	objId	objId	objId		
	String	streamId	null	null		
	String	authKey	null	authKey		
	String	tag	authKey	null		
	int	offset	offset	offset		
	int	maxResults	maxResults	maxResults		
	Timer	timer	timer	timer		
	ComponentType...	components	components	components		
4	Builder	builder	builder	builder	163	Names are equal after removing the `get` prefix of method names
	boolean	isTransposed	isTransposed	isTransposed		
	int	startColumnIndex	a.getStartColumnIndex()	0		
	int	endColumnIndex	a.getEndColumnIndex()	rows.size()		
	int	startRow	0	a.getStartColumnIndex()		
	int	endRow	rows.size()	a.getEndColumnIndex()		
5	String	msgFormat	"Message"	e	139	Incorrect ordering resulted in wrong method being called because of overloading
	Object...	args	e	"Message"		
6	Object	actual	Util.createThing(fromStuff)	someVariable	1504	Calls to JUnit's `assertEquals` method that swap expected and actual.[7]
	Object	expected	someVariable	Util.createThing(fromStuff)		
7	String	fileUser	stageName	fileUser	20	Needed a field instead of a local variable
	Collection<String>	writtenFilenames	customKeywordFiles	customKeywordFiles		
8	Object...	parts	Long.parseLong(getAccountId())	Long.parseLong(getAccountId())	53	Argument accidentally duplicated.
			Long.parseLong(getAccountId())	Long.parseLong(getAudienceId())		

Fig. 1. Selected argument selection defects found in revision history

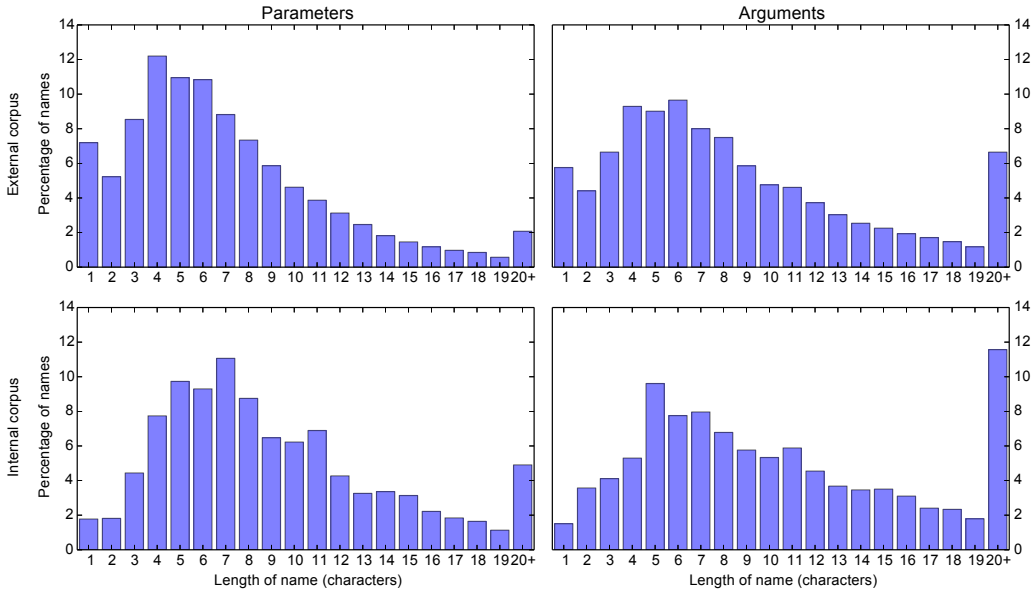

Fig. 2. Lengths of names in our External and Internal corpora

The distribution of name lengths in our external corpus is similar to the distribution reported for the open-source programs used to evaluate Nomen [Liu et al. 2016]. We therefore suggest that our external corpus can be used to understand the performance of these algorithms on a general open-source codebase whereas our internal corpus shows results that arise from within a more uniform software engineering process.

5 DETECTING ARGUMENT SELECTION DEFECTS

We now explain the various elements of our defect detection algorithm. The first priority is to select a suitable method for determining the similarity of parameters and arguments (Section 5.1). We then describe our strategy for deciding if an argument should be changed which we justify by simulating its behavior on our software corpus (Section 5.2). Finally, we combine these elements and describe the defect analysis algorithm as a whole (Section 5.3).

5.1 Determining the Similarity of Names

We first set out to find a suitable method for identifying whether an argument name matches its corresponding parameter. This is comprised of two steps: first, one must extract a name from an arbitrary expression used as an argument, and second, a suitable distance function must be used to determine if the extracted name and the parameter name are similar.

Java developers tend to follow either a camel case or underscore-separated naming convention. We therefore follow these conventions to split names into terms when required below.

5.1.1 Argument Name Extraction. We based our approach on that in Nomen, which we reformulate as follows:

[7]We would argue that this is a bug (albeit non-severe). This manifests as a problem when tests fail and the wrong error message is generated.

(1) **Identifiers** use the string representation except in the case of the `this` keyword, which uses the name of the class it represents.
(2) **Field accesses** use the name of the field (rather than the receiver name).
(3) **Method invocations** use the name of the method.
(4) **Constructor calls** use the name of the class being constructed.
(5) **All other cases** use a reserved name `*unknown*` which is defined to never match any name except that of the parameter in the same position. This means that arguments with `*unknown*` names are always left in their original position.

Bug 3 from Figure 1 demonstrates how errors can occur with `null` arguments. The word `null` itself should match equally well with all parameter names, so we add the concept of a wildcard name to represent this.

(6) **Null literals** use a reserved name `*wildcard*` which we define to match all parameter names.

We also notice (Bug 2) that naming conventions mean that some method prefixes (e.g. `get`) provide no additional information, so we remove these.

(7) If the first term of a method name is `get`, `set`, or `is`, we remove it. If this causes the method name to be empty, we recursively examine the receiver expression. For example, given a method `getAge()` which returns an `Optional<Integer>` we would extract the name `Age` from the argument `person.getAge().get()`.

5.1.2 Lexical Distance Functions. Our technique requires a distance function that takes two names and estimates the difference between them. A distance of 0 indicates the names are identical and a distance of 1 indicates the names are (maximally) different. For this work we limit ourselves to functions which can estimate distances using only local information available at the call site and so exclude measures which require knowledge of the whole corpus, such as the likelihood of a name.

We evaluated four candidate algorithms by building a test set of 4000 pairs of names from our corpus which we manually classified as matching or different. Cases for which we felt that the name was so general that it was not possible to determine whether it matched or not were classified as unknown. We first describe how we built this test set and then use it to evaluate our distance functions.

5.1.3 Constructing the Test Set of Example Names. The test set was built by visiting every call-site in the corpus and considering each method parameter in turn. For each parameter we emitted a pair for every argument that was assignable to the parameter. This ensured that we only generated pairs that have the potential to be an argument selection defect. This method extracted around 4.7 million unique pairs.

Figure 3 shows the distribution of Levenshtein distances[8] for these pairs on our two corpora. We can see from this distribution that drawing a direct sample of these names would be uninformative because the distribution is highly skewed towards dissimilar names. We therefore used weighted reservoir sampling [Efraimidis and Spirakis 2006] with each pair weighted in inverse proportion to the probability of its distance. This yielded a test set with a uniform distribution of Levenshtein distances. We note that the particular choice of distance function is not especially significant here: our objective was to build a balanced sample for labelling.

All 4000 pairs in the test data were independently labelled by two professional software engineers. After the labelling was completed, we extracted the pairs where they disagreed and discussed them further to reach a consensus.

[8] https://en.wikipedia.org/w/index.php?title=Levenshtein_distance&oldid=770797278

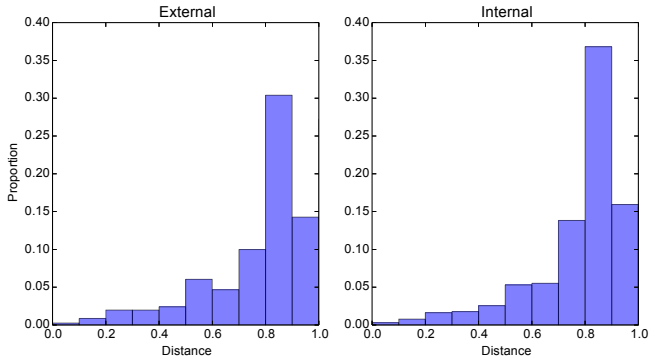

Fig. 3. Levenshtein distance between parameters and arguments

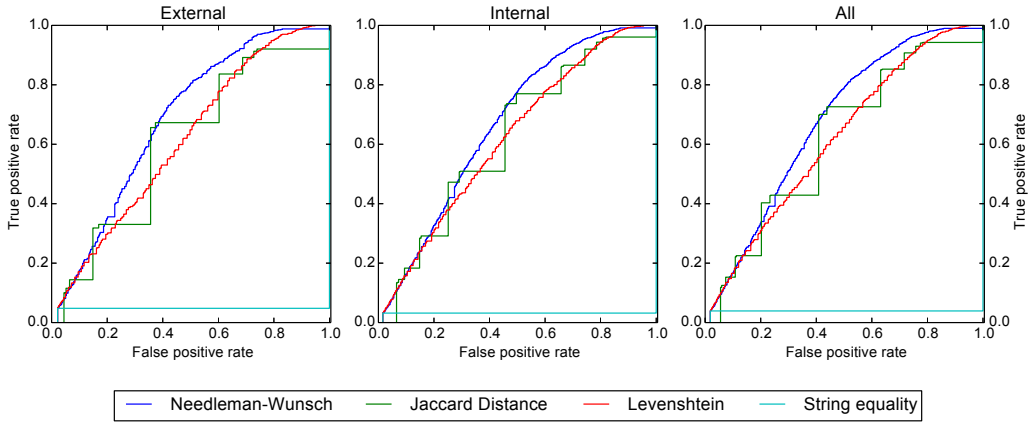

Fig. 4. ROC curves for the string distance functions

5.1.4 Evaluating Distance Functions. Approximately 800 pairs (20%) were still marked undecided at the end of this process. We deem these to be cases where the names were so general that it was not possible to determine if they matched or not. This observation was also made in Nomen in which a (large) set of general names was extracted by looking for parameters with large differences amongst the set of arguments used to call them. We excluded undecided pairs from the analysis below.

We considered 4 string distance functions:

(1) **String equality** assigns two identical strings a distance of 0, and all other pairs a distance of 1.

(2) **Levenshtein distance** is the minimum number of insertions, deletions, and updates required to convert one string to the other normalized by the maximum distance between strings of that length.

(3) **Needleman-Wunsch distance** [Needleman and Wunsch 1970] considers adjacent changes to have a different cost to isolated ones. We set the cost of a change to 8, the cost of beginning a sequence of changes to 8, and the cost of continuing a sequence of changes to 1, then again normalize by the maximum distance.

(4) **Jaccard distance** [9] reports the number of terms in common normalised by the total number of distinct terms. This is equivalent to the metric used in Nomen.

For the whole-string algorithms (1, 2, 3), we converted names to lowercase and separated terms with underscores since pairs such as `PERSON_ID` and `personId` should match.

We can consider each of these functions as a binary classifier. If the distance returned is less than some threshold, the strings are deemed to match; otherwise they are deemed not to match. Figure 4 shows the ROC curves for each of our functions. We can see that for both corpora the Needleman-Wunsch algorithm is best for almost all trade-offs of true- and false-positive rate.

We found this result surprising; we expected that the Jaccard distance would be more reliable since it is exploiting extra structure in the names. However, we note that one common difference between names involved the addition of an extra term, e.g. `html` and `htmlBody`. This applied to 30% of names we labelled as matching but only 1% of names labelled as different. We also saw substrings of terms used as abbreviations, e.g., `stringLength` and `strLen`. This applied to 7% of matching names and 1% of different names. In both these cases the Jaccard distance imposes a significant penalty because we have a relatively small total number of terms in the names. Both of these changes appear as adjacent edits to the Needleman-Wunsch algorithm.

We also experimented with a range of different edit penalties for the Needleman-Wunsch algorithm, but the values used above performed best.

5.2 Decision Strategy

Given a suitable distance function we next need to be able to determine if an alternative pairing (p, a) for a parameter is better than the original pairing (p, o). One way to do this is to use the distance function as a black box classifier which classifies a pair as matching if the distance is less than some threshold. In this case we would suggest replacing the original argument (o) with the alternative argument (a) if $\text{distance}(p, o) >= t$ and $\text{distance}(p, a) < t$. However, because we are considering two pairs (the original and the proposed alternative), we can instead classify them by applying a threshold to the difference in their distances and suggest a change if $\text{distance}(p, o) - \text{distance}(p, a) \geq t$. The latter approach is preferable since it allows us to factor the certainty of the replacement against the certainty of the original into the decision.

Because of the difficulty in classifying names, this technique generates too many false positives to be useful. We discovered this by looking at method calls in our corpus and calculating the distance between each original argument and its assignable alternatives. This then allows us to predict how many warnings we would generate for different thresholds. If we assume (based on our earlier study) that argument defect bugs are relatively rare, we should expect a small number of warnings. In reality we found that this approach generates unreasonably high warning rates. For example a threshold of 0.6 generates approximately 1000 warnings per million lines of code.

Considering all the arguments for a method call at the same time rather than independently offers a vast improvement. We consider only permutations of the method's arguments and report a defect only if $\frac{1}{n} \sum \text{distance}(p_i, o_i) - \text{distance}(p_i, a_i) \geq t$, where n is the number of proposed changes, and p_i, o_i, a_i are the parameter, original argument and proposed alternative for the ith change. This generates a much more reasonable warning rate: a threshold of 0.6 generates approximately 6 warnings per million lines of code.

[9] https://en.wikipedia.org/w/index.php?title=Jaccard_index&oldid=773856787

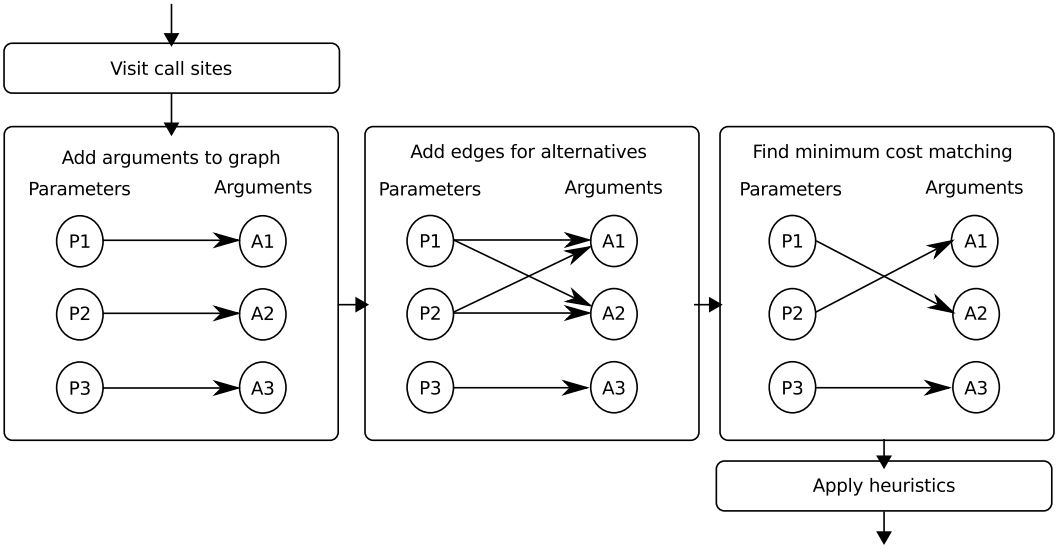

Fig. 5. Overall operation of the defect detection algorithm

5.3 Argument Defect Detection Algorithm

Figure 5 shows an overview of our defect detection algorithm. We formulate the problem as a bipartite graph problem matching the parameters of a method call to the minimum cost choice of arguments.

5.3.1 Visit Call Sites. The analysis operates by iterating over all method (and constructor) invocations, collecting the names and types of parameters and arguments.

Recovering the names and types of arguments (and other alternatives which are in scope) is a local analysis. This is highly beneficial from a performance viewpoint. However, recovering the parameter names requires information from the method declaration. There are three main options to recover this information:

- **Source availability**. If the source file containing the method declaration is in the list of source files to compile, then the compiler will resolve parameter names from this. This is unlikely to be the case in any significant project since most Java build systems pass dependencies as compiled class files.
- **Local symbol tables**. If the class file has been compiled with debugging information (`-g`) then names can be recovered from local symbol tables. However, these tables are not present in interfaces or abstract classes.
- **Parameter annotations**. Java 8 adds a compiler option (`-parameters`) which stores extra attributes in the class file containing the parameter names.

It is worth noting that changing compiler flags in a large production environment is a challenge. For example, parameter annotations create a change in the bytecode. This can cause feature-incomplete bytecode processing tools to fail, or brittle tests which rely on 'golden' bytecode to no longer pass. The annotations also increase the size of class files which can provoke out of memory failures on services. The results we report in this paper arise from running with local symbol table information (option 2).

5.3.2 Add Arguments to Graph. We next begin construction of the bipartite graph with an edge between each parameter and its associated argument. The cost of the edge is computed by the lexical distance function. For varargs methods with a variable number of arguments, we ignore the varargs parameter and arguments (if any).

5.3.3 Add Edges for Alternatives. We then add an edge between parameters and alternative arguments. Edges are only added if the alternative is assignable to the parameter and its distance score is lower than for the original argument. We consider type equality, subtyping, and auto-boxing. In the case of generic methods we assume that the currently instantiated type is correct rather than considering all of the generic type.

5.3.4 Find Minimum-Cost Matching. We use the Hungarian algorithm[10] to find the minimum cost assignment of parameter names to argument names. If the assignment differs from the original set of arguments, we have a candidate change.

5.3.5 Suggestion Heuristics. Even with a perfect name distance function we would still expect to report false positives. These arise from legitimate usage by the programmer. We therefore apply heuristics to identify this kind of usage.

Nested in reverse. In some cases, the purpose of the code is to call a method with arguments in a different order. For example, when rotating an image by 90 degrees, one swaps width and height when creating the destination canvas. We detect cases such as these by inspecting the names of any method or class definitions enclosing our method call. If these names contain a keyword indicating a deliberate swap then we reject suggestions within that method. We look for keywords matching any of the following regular expressions: `backward(s)?`, `complement`, `endian`, `flip`, `inver(t|se)`, `landscape`, `opposite`, `portrait`, `reciprocal`, `reverse(d)?`, `rotat(e|ed|ion)`, `swap(ped)?`, `transpose(d)?`, or `undo`.

Duplicate method calls. We often see multiple calls to the same method with different argument orderings. One example pattern is as follows:

```
switch(orientation) {
  case LANDSCAPE:
    bitmap = new Bitmap(width, height);
    break;
  case PORTRAIT:
    bitmap = new Bitmap(height, width);
  ...
}
```

In this case reordering method arguments is unlikely to be correct since the programmer has demonstrated the intention of using different orderings nearby. Another similar case is a recursive call with the arguments permuted.

```
public boolean test(int a, int b, boolean direction) {
  if (b < a) {
    return test(b, a, !direction);
  }
  ...
}
```

In this case the programmer is in the body of the method to be called and therefore will have the correct ordering in mind so that alternative ordering used in the recursive call is likely deliberate.

[10] https://en.wikipedia.org/w/index.php?title=Hungarian_algorithm&oldid=772831921

Style	Block after	Block before	Line after	Line before	Mixed

Example	`test(a /*a*/,`
	`b /*b*/);`
		`test(/*a*/ a,`
		`/*b*/ b);`
			`test(a, //a`
			`b); //b`
				`test(//a`
				`a,`
				`//b`
				`b);`	

Proportion of commented method calls	58%	4%	21%	13%	4%

Fig. 6. Argument commenting styles and their frequency in our corpus

To detect these instances we find other uses of the inspected method and discard changes to an argument which would be identical to another call. If the inspected call site is within a method body, we include all other calls in the same method body and also the method definition itself. If the inspected call site is in a field declaration, we include calls in all other field declarations in the same class.

Parameter name in comment. A common stylistic convention is to include the parameter name in a comment to indicate that the argument is correctly matched. We extracted instances of method invocations containing comments from our corpus and identified a variety of commenting styles, which we show in Figure 6. All of these are recognised by our implementation. We note that the 'line after' style is particularly problematic to parse because the final comment lies outside the bounds of the AST node for the method invocation, so a parser will naturally associate it with the subsequent node instead.

Low information names. When labelling names for evaluating the distance functions (Section 5.1.3), we found that some pairs did not contain sufficient information to determine a match or a difference. We therefore attempted to remove names such as these from consideration. From our labelling we decided to exclude:

- all names with one or two letters optionally followed by a digit
- `argX`, `paramX`, and `strX`, where X is any single digit
- the literal strings: `key`, `value`, and `label`

Any surviving candidate changes are then emitted to the developer as a warning.

6 RESULTS

We implemented a static checker based on the algorithm in Section 5.3 as part of Google's open-source Error Prone project [Aftandilian et al. 2012], a static analysis framework built on top of the javac compiler. We then executed this checker on our two corpora.

We explored the performance of our checker for a variety of different threshold values. The number of warnings generated varied from over 300,000 with the least restrictive (smallest) threshold to 165 with the most restrictive (largest) threshold. Overall, we classified 8,925 of these warnings as either true or false positives. This total comprises all warnings for thresholds of 0.4 or higher and a random sampling of warnings for lower thresholds. We conservatively assume that all remaining unclassified warnings are false positives. We found a total of 2,305 argument selection defects (true positives) over all thresholds. Figure 7 shows the precision and recall of our checker on our two corpora.

We are of course not able to report an accurate value for recall because we do not know the true number of argument defects in the corpora. Our reported recall is therefore an upper bound

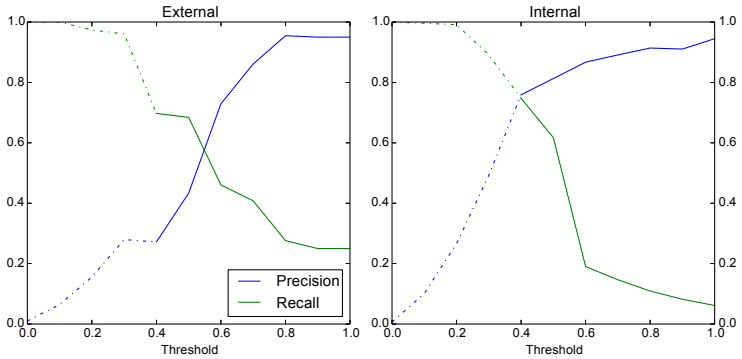

Fig. 7. Precision and recall of the checker

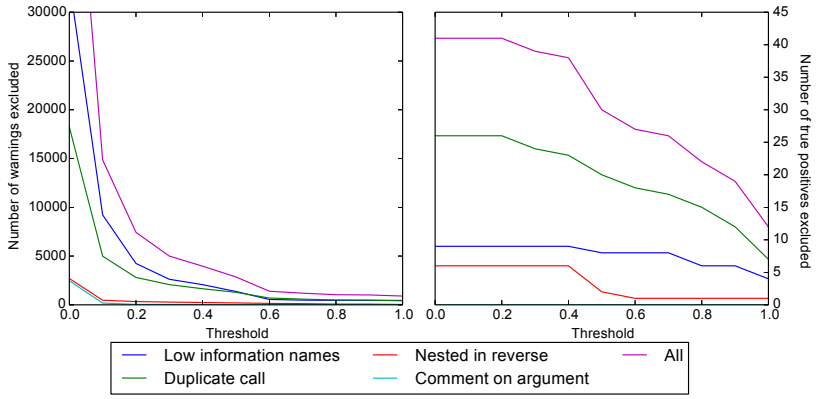

Fig. 8. Number of results eliminated by each heuristic

computed against the 2,305 defects that we discovered. Precision (true positive rate) only relies on counting true and false negatives, so we can report a true value for this. These values are very encouraging. A threshold of 0.6, for example, yields an overall precision of 73% on the external corpus and 87% on the internal corpus, reporting a total of 459 correctly identified defects.

Figure 8 shows the total number of results eliminated by each heuristic (left) and the total number of true positives (incorrectly) eliminated by each heuristic independently (right). All of these are highly effective and eliminate many results at the cost of excluding relatively few true positives.

The 'Comment on argument' heuristic only matches a small number of results but has 100% precision. This could be helpful as a suppression mechanism: engineers can override warnings from this check by providing explicit naming comments on arguments, with the added benefit that this also provides useful documentation for others. We note that during our evaluation we found two examples where the comment attached to an argument referenced the wrong parameter, as in a call to `test(int a, int b)` made as `test(b /*b*/, a)`.

In Section 5.1 we predicted that Needleman-Wunsch edit distance would provide the best performance for our check. Figure 9 shows the precision and recall for each of our string distance functions. We see that in our prediction was correct: Needleman-Wunsch has in general higher precision than all other methods and also higher recall except for very high thresholds.

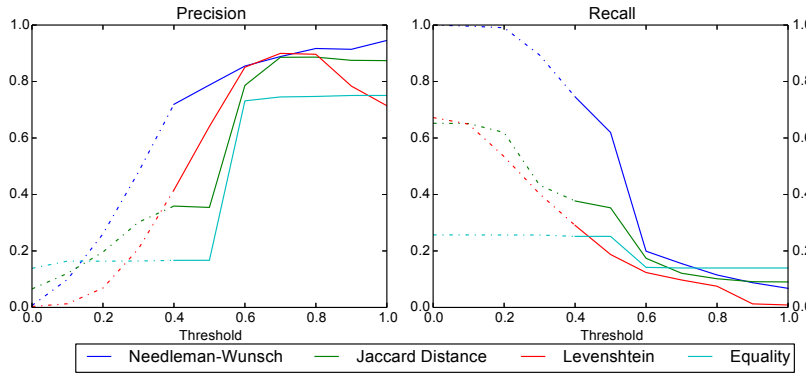

Fig. 9. Precision and recall for each of the string distance functions

Parameter names (in alphabetical order)	Number of defects
`actual`, `expected`	46
`height`, `width`	13
`familyName`, `givenName`	2
`autoSync`, `isSyncable`	2
`allowSharingIn`, `allowSharingOut`	2
`listingId`, `timestamp`	2
`columnName`, `rowName`	2
`handoffAttempted`, `isIncoming`	2
`payerShippingAddressDaysSinceFirstPurchase`, `payerShippingContactNameDaysSinceFirstPurchase`	2
`country`, `postalCode`	1

Fig. 10. Sample of reordered parameter name pairs

Figure 10 shows a selection of some of the more common parameter names (in any order) for which we found reordered arguments. The most common pair is associated with calls to methods in the family of JUnit's `assertEquals(Object expected, Object actual)`. In total 1,627 (71%) of the defects found had parameters containing the words "expected" or "actual". It is plausible that these would exist undetected in code because this kind of error is unlikely to be found at runtime since it is only visible when a test fails and the system prints out the wrong error message. Mistakes in calls to `assertEquals` are well known, and various alternative libraries have been developed to avoid this problem. One example is the Truth library[11] which uses a fluent interface[12] to increase clarity for the programmer.

We found no other library APIs in our results which accounted for defects across more than one source file. Notably, the next most common pair were `height` and `width`, but these occurred as pairs of `ints` across a range of API calls rather than in one place in particular.

Instead, the remainder of the defects were mostly one-off mistakes. Below we give four examples, all of which have been reported and accepted by the upstream developers.

[11] https://github.com/google/truth

[12] https://en.wikipedia.org/w/index.php?title=Fluent_interface&oldid=774034077

ConcurrentHashMap in OpenJDK. We found a defect [13] within an inheritance chain of three classes in which the middle class had reordered parameter names in its constructor. This manifested as two mistakes: in the call from the derived class to the middle class' constructor, and in the call from the middle class to the base class' constructor. These two mistakes negated each other, so the reordering had not been noticed.

MethodWriter in the ASM Library. We found the arguments `maxLocals` and `maxStack` were swapped in a call to `Attribute.put`. This was accepted as a defect[14] by the ASM developers, who noted that it was introduced 13 years ago. The defect is in code which is used if the user is writing their own custom attributes, so it would not affect the JVM but might affect any downstream user of the attributes.

SAXDocumentSerializer in OpenJDK. The arguments `systemId` and `publicId` were swapped in a call to `encodeDocumentTypeDeclaration`[15]. The XML standard requires that a `systemId` must be provided if a `publicId` is given so this error could result in emitting invalid documents in some cases which would be rejected by a validating parser.

ServerPreparedStatement in the MySQL JDBC driver. The arguments `resultSetConcurrency` and `resultSetType` were swapped in a call to `prepareStatement`[16]. This would result in the returned result having the wrong behavior and could have resulted in a subtle data concurrency bug for a system under load. The defect occurred in a method with protected visibility, so impact was limited to a particular use case of `PreparedStatement`. We reported this defect upstream, and it has been verified and accepted.

Finally, we observed that the majority of the defects were swaps of two (not necessarily adjacent) arguments, but a small number (21) involved permutations of 3 or 4 arguments.

6.1 Workflow Integration

We integrated our checker into the development workflow at Google to catch new bugs before they enter the codebase. In addition to detecting likely bugs, our checker suggests two potential fixes: swap the arguments to the best-performing arrangement, or add inline comments like `/* param= */` to make it explicit that the arguments were intentionally bound to those parameters.

We deployed this check into Google's code review tool through Tricorder [Sadowski et al. 2015]. The integration emits diagnostics generated by our checker into the code review tool and provides a way to apply the suggested fixes via a button click.

We find that for warnings on code changed in the proposed commit, 60% are addressed by applying the swap suggested fix, 25% are resolved in a different manner (e.g., swapping the parameters in the method declaration instead), and 15% are not acted upon. Only a very small number of users chose to apply the second suggestion of adding comments to names. There are a variety of possible reasons for this, such as user-interface issues, preference for a different comment style, or preference for no comments. We will be investigating these in our future work

Of note, we do not surface these warnings in IDEs. There are a variety of IDEs in use at Google, so, to save engineering effort, we choose to integrate analysis tools at the workflow points that all developers must pass – the command-line build tool and the code review tool. This is discussed further in the Tricorder paper.

[13] https://bugs.openjdk.java.net/browse/JDK-8176402
[14] Personal communication with Eric Bruneton
[15] https://bugs.openjdk.java.net/browse/JDK-8178411
[16] https://bugs.mysql.com/bug.php?id=85885

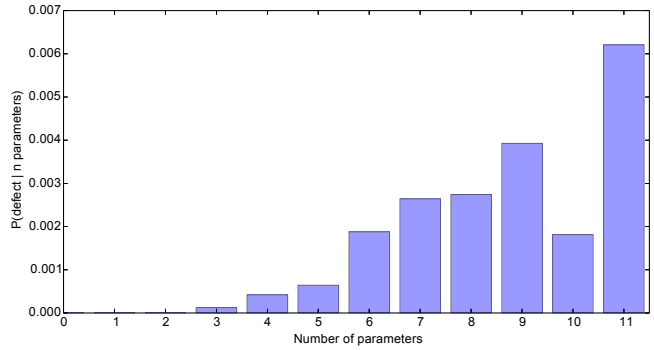

Fig. 11. Probability of an argument selection defect against number of method parameters

7 API DESIGN IMPLICATIONS

The defects found on calls to `assertEquals` provide a quantitative argument for improving its API design. The defects we found also provide evidence to support some more general advice in this area.

7.1 Methods Should Have 5 or Fewer Parameters

Various work discusses the benefits of APIs with fewer parameters [Bloch 2006] or using a builder pattern instead of a many-parameter constructor [Bloch 2008]. Here we show that the probability of a defect increases markedly with the number of method parameters.

We estimated the following probabilities:

- $P(n \text{ parameters}|\text{defect})$ is the probability of a defect occurring on a method with n parameters given that a defect occurred. We calculated this from the proportion of defects found on methods with n parameters.
- $P(n \text{ parameters})$ is the probability of a method with n parameters occurring. We calculated this from the proportion of methods in our corpus with n parameters.
- $P(\text{defect})$ is the probability of a defect occurring. We calculated this by dividing the number of defects found by the number of method invocations in our corpus

We excluded defects because of `assertEquals` from these figures in order to avoid bias, since they count for such a large proportion of the defects found. By application of Bayes rule, we then calculated $P(\text{defect}|n \text{ parameters})$, which is the probability of a defect given that a method has n parameters (Figure 11). A clear correlation is visible. It is also noticeable that the probability of a defect increases markedly after 5 parameters.

7.2 Parameters Should Be Sequenced Consistently

Previous authors have also proposed that parameters with similar meaning should be sequenced consistently to reduce API usage issues [Rama and Kak 2015]. We found evidence to support this: 23 of the defects found were in calls to an implementation of an `assertEquals`-style method which took parameters in the order `actual, expected` rather than `expected, actual`, as is the convention.

7.3 Ensure Names Are Consistent Through Inheritance Hierarchies

We found defects in very mature libraries because of mismatching of names in inheritance hierarchies. This was the case with ConcurrentHashMap. Another example defect was in the definition of

`namespaceAfterStartElement` in the interface `org.apache.xml.serializer.ExtendedContentHandler`. This library is included in a variety of software packages, including OpenJDK. The method is defined to take parameters `(uri, prefix)`. However, we found errors where callers of this method reorder these parameters. This resolves itself because all classes of the interface reorder the parameters again and thus negate the first error.

8 THREATS TO VALIDITY

This work addresses argument-related bugs only in Java code, which has particular naming conventions, style conventions, and program structure. As such, our choice of distance metrics and heuristics may not hold across other programming languages. However, our methodology for selecting among a set of candidate distance metrics and heuristics is applicable to other languages.

The majority of the code in the study comes from a single codebase at a single company, and thus our findings might be affected by company-specific style rules and tool usage. For example, Google's codebase makes heavy use of the Protocol Buffer[17] DSL and code generator, which affects our results (Section 9) via the generated names. Another potential issue is our wide use of the JUnit `assertEquals` methods (Section 6). Our study of the external corpus shows that our findings do generalize across other Java codebases with different style practices and tool usage.

We ignore low information names, but it is unclear how to match names that contain little semantic information. We only exclude names that lead a great number of undecided pairs, and we find that excluding matches involving these names leads to more useful results. Nomen [Liu et al. 2016] also excluded such names.

Other threats include our exclusion of undecided pairs and varargs from consideration. It is not clear how to handle name pairings that cannot be determined to match even by human inspection. Nor is it clear how to apply name matching to a pair which includes a single parameter name to a variable-length list of arguments.

There may exist better heuristics than the ones we list in Section 5.3.5, or tweaks to our heuristics that improve their performance (e.g., better regular expressions for *nested in reverse*). Better heuristics could further improve the false positive rate of our checker.

9 COMPARISON TO EXISTING APPROACHES

Order [Pradel and Gross 2011, 2013] looks for swaps between equally typed arguments in method calls by looking at the lexical similarity between arguments and parameters. Their analysis groups equally typed parameters together into sets and then considers a total lexical similarity score for each permutation of the set. If the permutation scores higher than the current combination, a swap is suggested.

Order was evaluated using a corpus of 12 Java programs (around 1 million lines of code) taken from the DaCapo benchmarks, and two anomalies affecting correctness were found from 31 warnings. However, the authors also considered warnings which arose from poor naming style (but which did not correspond to a defect) to be true positives and measure a naming true positive rate of 74%. These do not count as argument selection defects, so we discount them. This yields a defect true positive rate of 6.4%. The authors also investigated injecting synthetic defects. In this case, the injected true positive rate was around 80%.

There is a significant difference here between the real true positive rate and the synthetic rate. The authors do not comment on this, presumably because their naming true positive rate was similar to their injected true positive rate even though they were technically measuring different things. One explanation for this is that injecting defects assumes that programmers make these

[17] https://developers.google.com/protocol-buffers/

	Warning rate (per MLoC)	Number of warnings inspected	Number of true positives found	True positive rate
Nomen	157	169	9	5%
External	137	400	0	$< 0.25\%$
Internal	223	400	1	0.25%

Fig. 12. Performance of Nomen on our corpora

errors uniformly on all method calls. However, we see from our investigation that this is not the case.

The major difference between our approach and Order is the additional heuristics we developed for excluding false positives. With these heuristics disabled our true positive rate agrees with Order at around 10% on the external corpus. The algorithm used in Order is $O(n!)$ in the number of parameters since it considers all permutations, whereas our graph matching approach is $O(n^3)$.

Nomen [Liu et al. 2016] is a more aggressive approach which considers potential replacements from anywhere in scope rather than only looking for permutations of arguments. 5 of the 84 bugs we found in our original survey from commit messages fall into this class of error. Alternatives are drawn from scope based on the kind of the argument: local variables or fields of the enclosing class have all local variables or fields of the enclosing class as alternatives; field accesses or method invocations with no arguments have all fields and no-argument methods on the same receiver object as alternatives; other method invocations have all method invocations with the same parameter types on the receiver object as alternatives. The lexical distance is computed using Jaccard Distance.

Nomen was evaluated using a corpus of 11 open-source Java programs taken from SourceForge, comprising around 1 million lines of code. The authors found 9 defects from 169 reported warnings. The authors report their naming true positive rate of 80%, but the defect true positive rate is 5%.

The major question about this result is that of selection bias: the 11 programs were chosen manually from a larger corpus as those which were known to contain an argument selection defect. If the rate of these defects is low (as we have seen from our survey) then one would expect that the reported true positive rates are higher than would actually be achieved. We therefore decided to try to reproduce these results.

The authors of Nomen noticed that some parameter names are used with a wide variety of arguments (e.g. `key`, `value`) and so do not make good candidates for similarity comparison. They exclude arguments for these parameters from consideration with a blacklist of approximately 7,000 names which is built by identifying parameter names which have high average distance to all their corresponding arguments across all uses (globally).

The idea behind the blacklist is that there are certain names which are commonly overloaded and that this overloading is agnostic to the particular corpus under study. For example, we would expect `key` and `value` to be overloaded whatever the codebase. We therefore chose our blacklist parameters by experimentation so as to generate a similar number of names to Nomen.

Our results are shown in Figure 12. Our external corpus had a similar warning rate to Nomen, but our internal corpus rate was 42% higher. One explanation for this is the common use of the Protocol Buffer DSL and code generator in the internal codebase. When an enum is defined in this DSL, the code generator also generates additional, infrequently used constant fields that are needed for deserializing different versions of the same message type. These constant fields share the name of the enclosing type, whereas the enum values themselves do not (e.g., `MyEnum.VALUE1` vs. `MyEnum.UNKNOWN_MY_ENUM`). Because method parameters are often named after their type, these

tend to match more frequently than the existing argument. A manual inspection of the warnings on the internal corpus showed that 88 of the 400 were caused by this issue; removing that fraction from the set results in a warning rate of 173 per MLoC, which is in line with Nomen.

We found a true positive rate at least 20 times worse than reported in Nomen. We attribute this to the selection bias issue described earlier in Nomen's choice of corpus.

10 SUMMARY AND FUTURE WORK

In this paper, we focused on the issue of detecting argument selection defects. These are situations where an incorrect argument is chosen for a method parameter when the correct alternative was also available. We show that making judgements on single parameters at a time would produce high false positive rates and so focused on permutations of method arguments.

Our resulting algorithm has a true positive rate that is an order of magnitude more precise than comparable previous techniques. We found 2,305 defects, some of which were in mature libraries. We also analyzed the defects to quantitatively support existing best practice advice about API design.

Previous research has classified ambiguous or poor naming by developers as naming errors and included these as true positives. We excluded these and focused on incorrect arguments that were not the programmer's intent. This is important from an industrial context in particular: responding to error reports consumes developer time, so successful static analysis checks must maximise the value of their suggestions.

A significant proportion of the defects that we found did not actually result in incorrect behavior of the program because they were countered by an equal and opposite reordering elsewhere in the program. We believe that these are still worth fixing as they pose a risk to future maintenance of the code.

Many of the defects we found in existing code corresponded to subtle or uncommon use cases and are unlikely to occur in day-to-day use. This is to be expected since we would expect such defects in commonly used code to manifest themselves in testing. We deployed our checker as part of the code review process at Google, and 85% of issues reported are addressed by developers before committing their code.

We were able to show that the probability of an argument selection defect increases markedly beyond 5 parameters. One option for methods with many parameters is to redesign using a fluent interface or parameter objects. Another alternative is the use of named parameters, which could be emulated in Java with a static check in Error Prone which inspects the comments on arguments.

The choice of distance function is key to the performance of this algorithm and it is likely that there is room for improvement here. When labelling parameter and argument pairs, we noticed a variety of conventions that help in distinguishing names. For example, the parameters `start, end` might be legitimately passed `endOfPrevious, start` and the parameters `parent, child` might be legitimately passed `grandparent, parent`. It would be interesting to see whether a machine learning algorithm could learn these similarities.

ACKNOWLEDGMENTS

This work was supported by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within "CRISP", by the German Research Foundation within the Emmy Noether project "ConcSys".

REFERENCES

Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building Useful Program Analysis Tools Using an Extensible Java Compiler. In *International Working Conference on Source Code Analysis and Manipulation*

(SCAM). 14–23. https://doi.org/10.1109/SCAM.2012.28

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2014. Learning natural coding conventions. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 281–293.

Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles A. Sutton. 2015. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 38–49.

Miltiadis Allamanis and Charles A. Sutton. 2014. Mining idioms from source code. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 472–483.

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Experiences Using Static Analysis to Find Bugs. *IEEE Software* 25 (2008), 22–29. Special issue on software development tools, September/October (25:5).

Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM* 53, 2 (2010), 66–75.

Joshua Bloch. 2006. How to Design a Good API and Why It Matters. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '06)*. ACM, New York, NY, USA, 506–507. https://doi.org/10.1145/1176617.1176622

Joshua Bloch. 2008. *Effective Java (2nd Edition) (The Java Series)* (2 ed.). Prentice Hall PTR, Upper Saddle River, NJ, USA.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 156–165.

Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2011. Improving the Tokenisation of Identifier Names. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 130–154.

C. Caprile and P. Tonella. 1999. Nomen est omen: analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*. 112–122. https://doi.org/10.1109/WCRE.1999.806952

Ray-Yaung Chang, Andy Podgurski, and Jiong Yang. 2007. Finding what's not there: a new approach to revealing neglected conditions in software. In *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 163–173.

William W. Cohen, Pradeep D. Ravikumar, and Stephen E. Fienberg. 2003. A Comparison of String Distance Metrics for Name-Matching Tasks. In *Workshop on Information Integration on the Web (IIWeb)*. 73–78.

Pavlos S. Efraimidis and Paul G. Spirakis. 2006. Weighted random sampling with a reservoir. *Inform. Process. Lett.* 97, 5 (2006), 181 – 185. https://doi.org/10.1016/j.ipl.2005.11.003

E. Enslen, E. Hill, L. Pollock, and K. Vijay-Shanker. 2009. Mining source code to automatically split identifiers for software analysis. In *2009 6th IEEE International Working Conference on Mining Software Repositories*. 71–80. https://doi.org/10.1109/MSR.2009.5069482

Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 837–847.

Einar W. Høst and Bjarte M. Østvold. 2009. Debugging Method Names. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 294–317.

Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. 2007. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering* 3, 4 (2007), 303–318. https://doi.org/10.1007/s11334-007-0031-2

Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 306–315.

Bin Liang, Pan Bian, Yan Zhang, Wenchang Shi, Wei You, and Yan Cai. 2016. AntMiner: Mining More Bugs by Reducing Noise Interference. In *ICSE*.

Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. 2016. Nomen Est Omen: Exploring and Exploiting Similarities between Argument and Parameter Names. In *International Conference on Software Engineering (ICSE)*. 1063–1073.

Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology* 48, 3 (1970), 443 – 453. https://doi.org/10.1016/0022-2836(70)90057-4

Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2013. A statistical semantic language model for source code. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 532–542.

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 *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 383–392.

Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59 (2016), 78–87. http://dl.acm.org/citation.cfm?id=2854146

Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 371–382. https://doi.org/10.1109/ASE.2009.60

Michael Pradel and Thomas R. Gross. 2011. Detecting anomalies in the order of equally-typed method arguments. In *International Symposium on Software Testing and Analysis (ISSTA)*. 232–242.

Michael Pradel and Thomas R. Gross. 2013. Name-based Analysis of Equally Typed Method Arguments. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1127–1143.

M. Pradel, C. Jaspan, J. Aldrich, and T. R. Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. 925–935. https://doi.org/10.1109/ICSE.2012.6227127

Girish Maskeri Rama and Avinash Kak. 2015. Some Structural Measures of API Usability. *Software Prac. Experience* 45, 1 (Jan. 2015), 75–110. https://doi.org/10.1002/spe.2215

Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 44.

Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soederberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *International Conference on Software Engineering (ICSE)*.

Forrest Shull, Vic Basili, Barry Boehm, A. Winsor Brown, Patricia Costa, Mikael Lindvall, Dan Port, Ioana Rus, Roseanne Tesoriero, and Marvin Zelkowitz. 2002. What We Have Learned About Fighting Defects. In *Proceedings of the 8th International Symposium on Software Metrics (METRICS '02)*. IEEE Computer Society, Washington, DC, USA, 249–. http://dl.acm.org/citation.cfm?id=823457.824031

Suresh Thummalapenta and Tao Xie. 2009. Mining Exception-Handling Rules as Sequence Association Rules. In *International Conference on Software Engineering (ICSE)*. IEEE, 496–506.

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, Singapore, September 3-7, 2016*. 708–719.

Andrzej Wasylkowski and Andreas Zeller. 2009. Mining Temporal Specifications from Object Usage. In *International Conference on Automated Software Engineering (ASE)*. IEEE, 295–306.

Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *European Software Engineering Conference and Symposium on Foundations of Software Engineering (ESEC/FSE)*. ACM, 35–44.

Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. 2012. Automatic parameter recommendation for practical API usage. In *International Conference on Software Engineering (ICSE)*. IEEE, 826–836.