Mining API Expertise Profiles with Partial Program Analysis

Senthil Mani, Rohan Padhye and Vibha Singhal Sinha
IBM Research
{sentmani,ropadhye,vibha.sinha}@in.ibm.com

***Abstract*—A developer's API usage expertise can be estimated by analyzing source code that they have checked-in to a software repository. In prior work we proposed a system for creating a social network of developers centered around the APIs they use in order to recommend people and projects they might be interested in. The implementation of such a system requires analyzing code from repositories of large numbers of projects that use different build systems. Hence, one challenge is to determine the APIs referenced in code in these repositories without relying on the ability to resolve every project's external dependencies. In this paper, we consider a technique called Partial Program Analysis for resolving type bindings in Java source code in the absence of third-party library binaries. Another important design decision concerns the approach of associating such API references with the developers who authored them such as walking entire change history or use blame information. We evaluate these different design options on 4 open-source Java projects and found that both Partial Program Analysis and blame-based approach provide precision greater than 80%. However, use of blame as opposed to complete program history leads to significant recall loss, in most cases greater than 40%.**

I. Introduction

A developer's expertise can be characterized in two different dimensions, implementation expertise and usage expertise. *Implementation expertise* summarizes what a developer knows about a project. This information is used to answer questions such as "Who can tell me how this module works?" or "Who can fix this bug?". Various works have attempted to infer implementation expertise using different information sources. These include analyzing a project's change-history [1], bug reports [2] or a developer's actions within an IDE [3]. *Usage expertise* [4] addresses the question "What external libraries can a developer use?". Unlike implementation expertise, usage expertise is not limited in scope to the project from which it was mined. Developers often use the same libraries across projects and their familiarity with a library's application programming interface (API) acts as a transferable skill that is representative of their domain-specific knowledge.

In a previous ICSE NIER track paper [5], we proposed the construction of developer social networks using a graph of people, projects and the libraries they use. These networks were then used to generate recommendations such as a projects that a developer could join or people whom they could connect with. We also demonstrated a prototype of our application, APINet, which was seeded with data from over 500 open-source Java projects hosted on GitHub[1].

Any system that attempts to mine expertise profiles from hundreds or thousands of projects must make two design decisions: (1) how to extract API usages from source code and (2) how to associate these API references with their authors for creating expertise profiles.

In an ideal scenario, API usages could be extracted from source code by using standard program analysis tools which can generate typed intermediate representations (IR) of a program. However, in a real scenario, the number of projects is very large, encompassing a heterogeneous landscape of build processes and tools. It is impractical to attempt to resolve every project's dependencies as is required by most static analysis tools in order to generate typed IR. This is non-trivial to do even for the latest versions of each project, let alone for every snapshot in the project's change-history, as attempted by Williams and Hollingsworth [6], who note the extreme difficulties of such a approach. Therefore, we consider the use of *partial program analysis* (PPA) [7] for extracting API usages from source code in the absence of third-party libraries. PPA can analyze individual Java source files and perform best-effort type inferencing to resolve bindings for class and method references. These API references could then be matched to the developers who authored them for generating expertise profiles.

In order to associate API references with their authors, there are two options. The ideal approach would be to analyze each change committed to the repository of a project, identify the APIs referenced in the newly added or modified code, and associate these APIs with the author of the change-set. However, the computational cost associated with walking entire change histories of each project is non-trivial. Deploying this approach on a platform such as GitHub might imply that new commits get generated at a rate faster than they can be analyzed for API usage. As a scalable alternative, we can take snapshots of projects at periodic intervals and use the *blame* operation available in most version control systems to identify the author of each line of code, and assign to this developer all APIs referenced in that line. However, this approach is likely to miss instances of API usage by developers if the code they have contributed is subsequently overwritten by someone else.

We empirically evaluate these design choices by constructing a ground truth set derived from four open-source Java projects whose dependencies could be resolved using Maven[2].

[1]https://github.com

[2]http://maven.apache.org

```
1  package app.sample;
2
3  import java.sql.Connection;
4  import javax.servlet.http.HttpServlet;
5  import javax.servlet.http.HttpServletRequest;
6  import javax.servlet.http.HttpServletResponse;
7
8  import org.apache.log4j.Logger;                                    // Added in revision 2 by Bronn
9  import org.json.JSONObject;
10 import com.customdb.*;
11
12 public class ProfileServlet extends HttpServlet {
13         private Logger logger = Logger.getLogger(ProfileServlet.class);  // Added in revision 2 by Bronn
14
15         public void doGet(HttpServletRequest request, HttpServletResponse response) {
16                 Connection conn = null;
17                 try {
18                         conn = DatabaseManager.getConnection();
19                         JSONObject profileJson = new JSONObject(ProfileUtils.getProfileMap(conn,
20                                                 request.getRemoteUser(), getInitParameter("PROFILE_SOURCE")));
21                         response.getWriter().println(profileJson);
22                 } catch (Exception e) {
23                         logger.error("Something bad happened.", e);       // Added in revision 2 by Bronn
24                 } finally {
25                         try { conn.close(); } catch (Exception ignore) {} // Added in revision 3 by Bronn
26                 }
27         }
28 }
```

Fig. 1. A toy example of a Java source file with API usages underlined. The class is an HTTP Servlet that fetches a user's profile from a database using a utility method, and produces the resulting profile encoded in JSON. The source code shown here is the state of the file after revision number 3.

We use *precision* and *recall* as metrics to determine efficacy of each choice. Our main contributions in this paper are:

- An empirical evaluation of the effectiveness of partial program analysis for extracting API usages in order to generate developer expertise profiles.
- An empirical evaluation of the quality of blame-based expertise assignment to developers as opposed to a change-history walking approach.

The rest of the paper is structured as follows: We motivate this work with the help of some examples in Section II. We present the results of our empirical evaluation of these alternatives in Section III and enumerate the possible threats to the validity of these experiments in Section IV. Section V discusses related work and we conclude in Section VI along with a discussion of future work.

II. Theory and Motivation

In this section we first define some terminologies used in this paper and then formulate three research questions which are evaluated in the subsequent section. Figure 1 is a running example used throughout this section.

A. Terminology

1) API usage: A developer's familiarity with a third-party library depends on their ability to predict the effect of executing some operation provided by the library via an application programming interface (API). Hence, while analyzing Java code for the purposes of determining API expertise, we restrict ourselves to code elements that can possibly have side-effects, such as method invocations or class instantiations (via constructor methods).

> **Definition 1.** An instance of **API usage** in Java source code occurs at the invocation of an object's method, at an invocation of a static class method or at the point of class instantiation via an object's constructor.

In Figure 1, all API usages that match this definition have been underlined. The third column of Table I shows the fully qualified names of the API methods, as resolved by a complete program analysis (CPA) that would use third-party libraries. Although most classes have been explicitly imported, `ProfileUtils` and `DatabaseManager` are referenced without qualification anywhere in the file. While the former belongs to the same package as the enclosing `ProfileServlet`, the latter has been implicitly imported via a wildcard on line 10.

2) Expertise Profiles: Although the the notion of *expertise* is probably very subjective, in order to remain consistent with existing literature, we use the term *expertise profile* to refer to the set of artifacts that some automated technique deems to be familiar to a developer. For Java libraries, different granularity of library artifacts – such as packages, classes or methods – may be considered for building such a profile. In this paper, we use the granularity of Java classes to build these profiles since it is typically the API documentation of a class that

TABLE I
API USAGES EXTRACTED FROM `ProfileServlet`

Line	Invocation	API extracted using CPA at Revision 3	API extracted using PPA at Revision 3	API extracted using CPA at Revision 4
(1)	(2)	(3)	(4)	(5)
13	`getLogger`	`org.apache.log4j.Logger`	`org.apache.log4j.Logger`	`org.slf4j.LoggerFactory`
18	`getConnection`	`com.customdb.DatabaseManager`	`UNKNOWNP.DatabaseManager`	`com.customdb.DatabaseManager`
19	`new JSONObject`	`org.json.JSONObject`	`org.json.JSONObject`	`org.json.JSONObject`
19	`getProfileMap`	`app.sample.ProfileUtils`	`UNKNOWP.ProfileUtils`	`app.sample.ProfileUtils`
20	`getRemoteUser`	`javax.servlet.http.HttpServletRequest`	`javax.servlet.http.HttpServletRequest`	`javax.servlet.http.HttpServletRequest`
20	`getInitParameter`	`javax.servlet.GenericServlet`	`javax.servlet.http.HttpServlet`	`javax.servlet.GenericServlet`
21	`getWriter`	`javax.servlet.http.HttpServletResponse`	`javax.servlet.http.HttpServletResponse`	`javax.servlet.http.HttpServletResponse`
21	`println`	`java.io.PrintWriter`	`UNKNOWNP.UNKNOWN`	`java.io.PrintWriter`
23	`error`	`org.apache.log4j.Logger`	`org.apache.log4j.Logger`	`org.slf4j.Logger`
25	`close`	`java.sql.Connection`	`java.sql.Connection`	`java.sql.Connection`

TABLE II
EXPERTISE PROFILES GENERATED FROM `ProfileServlet`

	Profiles generated at Revision 3		Profiles generated at Revision 4	
Developer	With CPA and Commits/Blame	With PPA and Commits/Blame	With CPA and Blame	With CPA and Commits
(1)	(2)	(3)	(4)	(5)

Alayne	`javax.servlet.GenericServlet`
	`javax.servlet.http.HttpRequest`
	`javax.servlet.http.HttpResponse`
	`com.customdb.DatabaseManager`
	`app.sample.ProfileUtil`
	`java.io.PrintWriter`
	`org.json.JSONObject`
		`javax.servlet.http.HttpServlet`
		`javax.servlet.http.HttpRequest`
		`javax.servlet.http.HttpResponse`
		`org.json.JSONObject`
			`org.slf4j.LoggerFactory`
			`javax.servlet.GenericServlet`
			`javax.servlet.http.HttpRequest`
			`javax.servlet.http.HttpResponse`
			`com.customdb.DatabaseManager`
			`app.sample.ProfileUtil`
			`java.io.PrintWriter`
			`org.json.JSONObject`
				`org.slf4j.LoggerFactory`
				`javax.servlet.GenericServlet`
				`javax.servlet.http.HttpRequest`
				`javax.servlet.http.HttpResponse`
				`com.customdb.DatabaseManager`
				`app.sample.ProfileUtil`
				`java.io.PrintWriter`
				`org.json.JSONObject`

Bronn	`org.apache.log4j.Logger` `java.sql.Connection`	`org.apache.log4j.Logger` `java.sql.Connection`	`org.slf4j.Logger` `java.sql.Connection`	`org.apache.log4j.Logger` `java.sql.Connection`

TABLE III
HISTORICAL CHANGES TO `ProfileServlet`.

Ver.	Author	Message
1	Alayne	Creating the basic `ProfileServlet`.
2	Bronn	Adding Log4J logging in case an exception is thrown.
3	Bronn	Properly closing the DB connection in `finally`.

describes operations of all its methods; the intuition being that if a developer has read the API documentation of a class and is able to use one or more of its methods then they can probably use other methods in that class easily as well.

Definition 2. A developer's **expertise profile** is a set of Java classes that they may be familiar with.

Consider the commit log in Table III, which shows the change history of the `ProfileServlet` class up its current state as shown in Figure 1. The original author of the file, Alayne, created the file in revision 1 of the project and contributed most of its functionality including getting a database connection using `DatabaseManager`, determining the logged-in user from the `request`, fetching their profile using `ProfileUtils`, converting the resulting map into a `JSONObject` and printing out the result using `response.getWriter().println()`. Another developer, Bronn, added two important pieces of functionality in the next two versions. In revision 2, a `Logger` was created at what is now line number 13 in order to log error messages at what is now line number 23. In revision 3, a `finally` block was added with a call to the `close` method of `Connection` in what is now line number 25.

The second column of Table II shows the ideal expertise profiles for these developers when considering changes to the `ProfileServlet` class alone. Alayne's profile consists of API usages extracted from her initial contribution, which makes up most of the functionality of `ProfileServlet`. Bronn's profile consists of API usages contributed in revision 2 and revision 3, viz. the Apache Log4J `Logger` and the JDBC `Connection` object.

B. Research Questions

Recommender systems that can be developed using API expertise profiles usually require harnessing data from thousands of repositories in order to be effective. A system that attempts to mine API usage profiles from such large numbers of repositories would typically face the following roadblocks:

1) Every project has its own conventions for resolving external dependencies to compile its source code. A technique is needed to extract API usages (in the form of fully-qualified class names) from the code in these repositories without relying on complete program analysis.
2) Change-histories of large projects can stretch into thousands of commits. Also, the libraries used across versions do not change very frequently. In order to improve scalability, it would be desirable to have a technique that can assign expertise to developers without walking the entire change history.

```java
package app.sample;

import java.sql.Connection;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.slf4j.Logger;                                        // Added in revision 4 by Alayne
import org.slf4j.LoggerFactory;                                 // Added in revision 4 by Alayne
import org.json.JSONObject;
import com.customdb.*;

public class ProfileServlet extends HttpServlet {
        private Logger logger = LoggerFactory.getLogger(ProfileServlet.class);// Modified in revision 4 by Alayne

        public void doGet(HttpServletRequest request, HttpServletResponse response) {
                Connection conn = null;
                try {
                        conn = DatabaseManager.getConnection();
                        JSONObject profileJson = new JSONObject(ProfileUtils.getProfileMap(conn,
                                        request.getRemoteUser(), getInitParameter("PROFILE_SOURCE")));
                        response.getWriter().println(profileJson);
                } catch (Exception e) {
                        logger.error("Something bad happened.", e);       // Added in revision 2 by Bronn
                } finally {
                        try { conn.close(); } catch (Exception ignore) {} // Added in revision 3 by Bronn
                }
        }
}
```

Fig. 2. The `ProfileServlet` class after revision 4 in which Alayne migrated the logging framework from Apache Log4J to SLF4J.

We address these roadblocks by considering two approximate techniques for API usage extraction and expertise assignment respectively and formulate three research questions for evaluating the effectiveness of these techniques individually as well as in combination.

1) Partial Program Analysis (PPA): Most static analysis tools for Java, such as the Eclipse Java Development Toolkit [8] or the Soot [9] framework, require all of the project's dependencies, both internal and external, to be available in source or byte-code form, in order to generate intermediate representations (IR) with method and type bindings resolved. For applications such as program optimization which require hard guarantees on correctness of the IR, a complete program analysis tool is the ideal solution.

Partial program analysis [7] is a technique that was developed specifically for software engineering applications that can tolerate some level of imprecision in the IR. PPA uses type inference and heuristics to resolve as many bindings as possible in the source code of a Java class without requiring information about any other classes in either source or binary form. Hence, it is an ideal candidate for the purpose of API usage mining from software repositories.

PPA has in the past been evaluated for its quality of type resolution when analyzing only one class at a time independently of other artifacts. The original study evaluated four open-source projects for which 91.2% type facts were correctly inferred and only 2.7% of type facts were erroneously reported. However, three of these projects did not have any external dependencies (apart from the Java standard library) and the evaluation considered partially qualified type bindings as correct resolutions if the base identifiers matched. Since our application of API expertise profiles mainly deals with detecting fully qualified names of artifacts from third-party libraries, these results cannot be directly extended without formal evaluation.

Although the literature contains several uses of PPA for software engineering applications that analyze project change history [10]–[12], none of these studies have empirically evaluated the quality of the resulting applications or recommendations in comparison to those that could have been developed if it was possible to resolve all type bindings correctly. Our first research question is thus:

> **RQ 1.** Do the API usages extracted by *partial program analysis* enable the generation of expertise profiles of comparable quality to those that could have been generated using complete program analysis?

The fourth column of Table I shows the API usages that partial program analysis would extract from the code in Figure 1 by looking at the `ProfileServlet` class alone. PPA is able to correctly resolve references to types that have been explicitly imported, including method invocations on objects of those types. However, due to incomplete information about other artifacts in the project or about external dependencies, PPA is unable to resolve all types.

Firstly, due to the wild-card import on line 10, any unqualified usage of class names introduces an ambiguity. It is not possible to determine, by looking at one file alone, whether such a class belongs in the same package as this file, or whether it is implicitly imported from one of the wild-card imported packages. Hence, PPA returns a qualification of `UNKNOWN` for references to `DatabaseManager` and `ProfileUtils` on lines 18 and 19 respectively.

Secondly, PPA does not have a model of classes defined outside this file, such as `HttpServlet` or `HttpRequest`. Hence, it incorrectly attributes the call to `getInitParameter` on line 20 as belonging to `HttpServlet` when in fact it belongs to its super-class `GenericServlet`. Similarly, since PPA does not know the return type of `HttpRequest.getWriter()`, it cannot correctly identify the target of `println` on line 21.

In this example, out of the 10 API usages, PPA is able to correctly infer 6 instances while incorrectly reporting only 1 instance. The expertise profile of developers generated using these API usages is shown in the third column of Table II. Here, there is a loss in precision due to the assignment of an incorrect API (`HttpServlet`) to Alayne. Her profile is also missing some entries that were present in the previous column (profiles generated by CPA) which implies a loss of recall. However, since expertise profiles for developers are typically created by combining the result of analyzing all classes in a project, it is possible (and perhaps even likely) that another usage of these missing APIs by the same developer would be correctly inferred by PPA and hence the overall expertise profiles would have better recall overall. This is exactly what we intend to discover with our first research question.

2) Blame-based expertise assignment: The second roadblock for mining large numbers of repositories pertains to practical issues concerning scalability when having to walk through entire change histories of every candidate project.

An alternative approach is to analyze only one version of the project (e.g. the latest version), and use *blame* information to assign expertise to developers. The *blame* operation, available in most version control systems, annotates every line in a given file at some version with the name of the developer who last authored that line of code, along with the revision number in which it was last committed, much like the comments in Figure 1. Thus, an approximate expertise profile can be generated by extracting API usages at every line in every source file at the latest version and assigning these APIs to the developers who can be blamed for introducing that line of code.

Such a technique requires only one pass over all files at the latest version of every project, thereby greatly reducing the computational costs associated with expertise mining. However, the use of *blame* information at just the latest version implies that this technique is unable to mine expertise from code that was contributed in some earlier version but subsequently deleted or overwritten by another developer. Again, as in our earlier research question, we hypothesize this technique might still be useful because an instance of API usage by a developer that is missed in one context may be recovered by a successful assignment in another context where the code has not been overwritten.

> **RQ 2.** Is it sufficient to use *blame* information to assign API usage expertise to developers instead of walking the entire change history of a project?

Consider again the code in Figure 1 and the actual API usages (as extracted by a complete program analysis) in the third column of Table I. Also consider the log of changes in Table III. In the first three revisions, none of the lines of code that were originally introduced have been overwritten by another developer. Hence, the expertise profiles for developers generated using the *blame*-based technique at this point is the same as in the case of change-set based assignment and is shown in the second column of Table II.

Now say the `ProfileServlet` class undergoes another change (revision 4). In this case, Alayne decides to replace the use of Apache Log4J with the Simple Logging Facade for Java (SLF4J). The only modifications in the code required to do this involve changing the factory method at line 13 and adding corresponding imports for SLF4J. Figure 2 shows the source code of the `ProfileServlet` class at version 4.

Now, if API usage were to be extracted from this version of the code the results would be as shown in the right-most column of Table I. The two differences from the second column are: (1) Line 13 now contains a reference to `LoggerFactory` from `org.slf4j` and (2) Line 23 now invokes the `error` method of `org.slf4j.Logger`.

If expertise were to be assigned to developers using the *blame*-based assignment, two discrepancies would arise. Firstly, Bronn will incorrectly be assigned familiarity with `org.slf4j.Logger` due to the annotation at line 23, and secondly, the fact that he had once used the Apache Log4J API is now lost in history. The fourth column in Table II shows the profiles that would result with this technique. If instead API usages were extracted per-commit and assigned only to the author of that change-set, they would correctly retain the information that Alayne used `org.apache.log4j.Logger` as shown in the last column.

However, we expect such a change that modifies the declared type of an object while preserving its invocations (having identical method signatures) to be a rare occurrence in practice. Hence, while we expect the *blame*-based approach to generally suffer in recall, we do not expect much loss in precision overall. Since recommender systems are more sensitive to precision than recall, the second research question intends to explore whether this approach is acceptable.

3) PPA + Blame: The first two research questions explore the effect of changing two variables independently: (1) the technique used for extracting API references from code and (2) the approach for assigning expertise to developers using extracted APIs.

Since the aforementioned approximations (PPA and blame) have been proposed to improve the feasibility and scalability

TABLE IV
SUBJECTS USED IN OUR EXPERIMENT

Subject	Commits	Java Files	Dev.	> 5 Commits	JARs
Bukkit/CraftBukkit	2,410	507	162	40	17
nasa/mct	677	903	16	10	74
nysenate/OpenLegislation	702	146	12	7	63
square/picasso	330	47	43	7	53
Total	4,119	1,603	233	64	207

of expertise mining, we are also interested in evaluating the effect of applying these techniques in combination. Our third research question is thus:

RQ 3. Is the combination of *Partial Program Analysis* and *blame* information an effective heuristic for generating API expertise profiles?

III. EVALUATION

A. Data Set

We address each research question by performing an experiment on four subjects. These subjects, listed in Table IV, are open-source Java projects that are hosted on GitHub. `CraftBukkit` is an implementation of the MineCraft Server API. `mct` is a real time monitoring and visualization platform developed at NASA Ames Research Center for use in spaceflight mission operations, `OpenLegislation` is a web service that delivers legislative information from the New York State Senate and Assembly to the public in near real time and `picasso` is an image downloading and caching library for Android. The reason why we chose these projects is that they represent a variety of real-world applications with many domain-specific dependencies (the number of JARs required by the project in the latest version is shown in the last table of the column) that could fetched using Apache Maven. Also, these subjects have a good amount of change history (ranging from 330 to over 2,400 commits) and several developers (43 to 167) who have made at least one commit. However, since our application was the generation of expertise profiles which depends on analyzing code contributed by developers, we only generated profiles for developers who had a minimum of 5 commits (threshold identified using interquartile ranges) to the project. Also, we only analyzed a subset of files in each project which were under the path `src/main/java` inside various subcomponents in the projects. This was done so that we could carefully resolve dependencies using Apache Maven at each commit to generate the ground truth set as explained next. Finally, we consider API usage expertise only for third party libraries and hence did not assign to developers API usages of classes defined in the same project to developers.

B. Data Collection

1) Ground Truth: In order to address each of the three research questions, we need to evaluate each strategy and compare its results with an ideal truth set. For the purposes of building expertise profiles, the ideal strategy is to walk the entire change history of a project, and for each change-set, assign its author the APIs extracted from the committed delta using complete program analysis (CPA). Hence, we refer to this strategy as CPA-COMMITS.

We constructed this ground truth set by manually ensuring that each intermediate revision had its dependencies resolved (JARs were downloaded from the Apache Maven repository). For CPA we used the Eclipse JDT Core [8] which allowed us to construct an abstract syntax tree (AST) of a Java file and resolve type bindings as long as all the project's source files and binaries of external libraries were on the class-path. For each file modified in a commit, we constructed ASTs for the state of the file before and after the change, and extracted API references in all nodes present in lines that were modified or newly added. These API references were then added to the experience profile of the author of the commit.

2) Partial program analysis: For RQ1, we needed to evaluate partial program analysis (PPA) for its effectiveness in building expertise profiles. We leveraged the implementation of PPA for Eclipse [3] which allowed us to use the same AST-based approach as we did when constructing the ground truth, but instead used PPA for resolving type and method bindings in the AST nodes. The main difference between using CPA and PPA for type resolution is that PPA does not require information about other source files in the project or any library JARs on the class-path. In fact, PPA can analyze just a string of source code for a Java class on its own. Thus, in theory, PPA could be applied even for projects whose dependencies cannot be resolved. For RQ1, we wanted to only change the method of API usage extraction and not the approach for assigning expertise. Hence, we used the same change-history walking approach to assign extracted API references in lines modified or added in each change-set to the author of that change-set. We call this data set PPA-COMMITS.

3) Blame-based expertise assignment: For RQ2, we wanted to evaluate just the effect of using *blame* as a method of expertise assignment as opposed to walking entire change histories of every project. Hence, for this experiment, we compared different expertise assignment strategies using CPA only. The candidate data set collected in this experiment is thus called CPA-BLAME. This data was collected by performing the `git blame` operation on each Java file in the candidate data sets at the latest versions of the project. The *blame* operation annotates each line with the commit that it was last modified, which can be used to get the developer who authored that line of code. API references were extracted per Java file using the standard Eclipse JDT-based CPA technique for type resolution, and then the references on each line were associated with the author of that line. The resulting expertise were compared with the original ground truth of CPA-COMMITS.

4) PPA + Blame: For RQ3, we combined the previous two approaches by using PPA for extracting API references from Java files at the latest version of each project and associating API usages at each line with the author of that line of code

[3] http://www.sable.mcgill.ca/ppa/ppa_eclipse.html

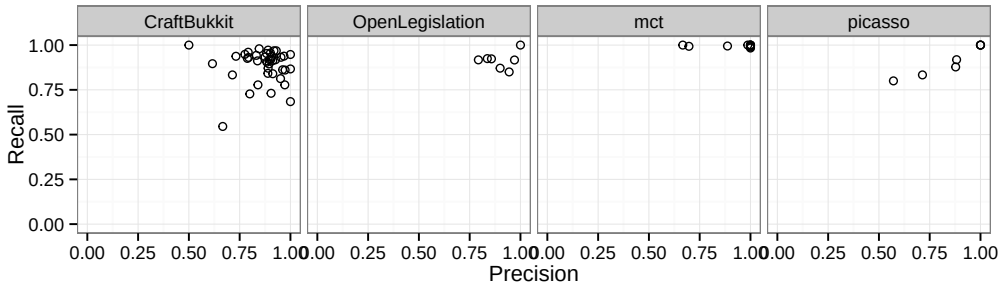

Fig. 3. Scatter plot of precision and recall per developer for PPA-COMMITS

using `git blame`. This data set is called PPA-BLAME and was evaluated against the ground truth of CPA-COMMITS. Thus, this experiment intended to evaluate the effect of changing both variables (API extraction and expertise assignment) at the same time.

C. Evaluation Metrics

We use *precision* and *recall* metrics to compare the goodness of PPA-COMMITS, CPA-BLAME and PPA-BLAME strategies with the ground truth of CPA-COMMITS. Even though our experiments are not evaluating a *search-and-retrieve* system, the expertise profiles we produce are a collection of APIs and we intended to evaluate the quality of these profiles with respect to those generated by the CPA-COMMITS approach. Our system can be thus be considered a *recommender system* for which precision and recall are standard evaluation metrics.

We explicitly define *precision* as the fraction of API usages present in the profile generated by the candidate strategy (S) in consideration (one of PPA-COMMITS or CPA-BLAME or PPA-BLAME) overlapping with API usages identified in the ground truth profile (CPA-COMMITS) over the total number of API usages in the expertise profile of the candidate strategy S.

$$Precision = \frac{|Profile_S \bigcap Profile_{\text{CPA-COMMITS}}|}{|Profile_S|} \quad (1)$$

Similarly, the *recall* of an expertise profile generation strategy S (one of PPA-COMMITS or CPA-BLAME or PPA-BLAME) is the number of API usages overlapping with API usages in the ground truth (CPA-COMMITS) over the total number of API usages in the profile of the ground truth (CPA-COMMITS).

$$Recall = \frac{|Profile_S \bigcap Profile_{\text{CPA-COMMITS}}|}{|Profile_{\text{CPA-COMMITS}}|} \quad (2)$$

In our experiments, precision and recall are calculated and presented both at a developer-level (which only considers profiles as a set of classes assigned to that developer in both the candidate and the ground truth) as well as at a project level (which considers the profile as a set of pairs of developers and the classes associated with them).

D. Experiments

1) PPA-COMMITS *vs.* CPA-COMMITS*:* This experiment was intended to answer the first research question:

> **RQ 1.** Do the API usages extracted by *partial program analysis* enable the generation of expertise profiles of comparable quality to those that could have been generated using complete program analysis?

Results: Figure 3 presents a scatter plot of precision vs. recall across all subjects. Each data point represents an individual developer's precision and recall values. For 52 of 64 developers, both precision and recall values are greater than 0.75. For 7 out of 64 developers, all API usages were perfectly identified (precision = 1 and recall = 1). The other precision and recall values are not too far away from 100% in all subjects. The project-level precision and recall values are listed in Table V (columns 2–4). For every developer whose who had an expertise profile using CPA, there was an expertise profile for PPA as well. Across the four subjects, the project-level precision is between 0.8 and 0.9, while recall is always greater than 0.9.

Discussion: In this experiment we evaluated the imperfection introduced in extraction of API usages by partial program analysis. PPA scores well in terms of precisely identifying the API usage expertise for developer in comparison with CPA. From a recall perspective, PPA suffers only when it cannot guess the right type due to import ambiguity (such as in the presence of wild-card imports). However, since expertise profiles are generated by collecting API references for each developer across multiple files, the final result has a high recall value. In general PPA seems to perform very well as compared to CPA for the purposes of expertise identification.

2) CPA-BLAME *vs.* CPA-COMMITS*:* This experiment was intended to answer the second research question:

> **RQ 2.** Is it sufficient to use *blame* information to assign API usage expertise to developers instead of walking the entire change history of a project?

Results: Figure 4 presents a scatter plot of precision vs. recall across all subjects. The average precision and recall values per subject is listed in Table V (columns 5–7). For 37 out of 56 developers, API usages have been precisely identified (precision = 1), though a perfect match occurred for only 4

TABLE V
PROJECT LEVEL PRECISION AND RECALL FOR RQ1, RQ2 AND RQ3

Subjects	RQ1: PPA-COMMITS			RQ2: CPA-BLAME			RQ3: PPA-BLAME		
	# of Active Developers	P	R	# of Active Developers	P	R	# of Active Developers	P	R
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
CraftBukkit	40	0.86	0.94	38	0.97	0.56	38	0.94	0.52
NASA mct	10	0.90	0.99	9	0.98	0.48	10	0.92	0.48
OpenLegislation	7	0.84	0.91	4	1	0.66	4	0.96	0.61
picasso	7	0.87	0.90	5	1	0.59	5	0.86	0.52

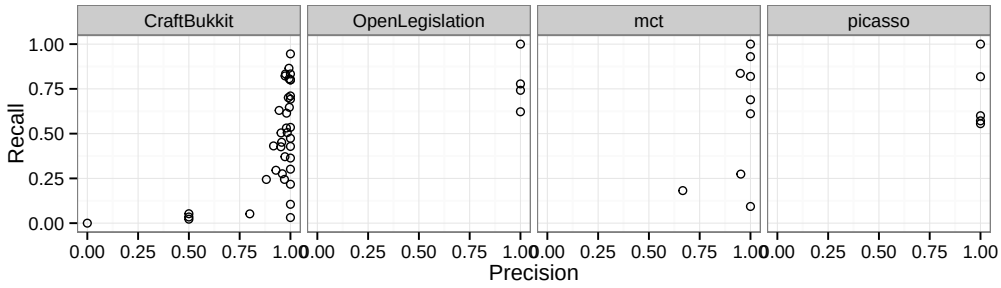

Fig. 4. Scatter plot of precision and recall per developer for CPA-BLAME

developers. Precision and recall values at project-level ranges from 0.97 to 1 and 0.48 to 0.66 respectively. This technique missed out API usage for 8 developers.

Discussion: In this experiment we evaluated the imperfection introduced in assigning of API usage expertise to developers. Since both the candidate and the ground truth were using CPA, the extraction of API usages would be almost identical. As observed, for 66% of developers the precision is 1.

The blame strategy suffers in recall because code contributed by one developer may have been overwritten by another developer. This means that developers might be assigned fewer API usages than what they actually used. In fact, this might even cause some developers to have completely empty expertise profiles. In our evaluation, we found empty profiles for a couple of developers in each subject.

3) PPA-BLAME *vs.* CPA-COMMITS*:* This experiment was intended to answer the third research question:

> **RQ 3.** Is the combination of *Partial Program Analysis* and *blame* information an effective heuristic for generating API expertise profiles?

Results: Figure 5 shows the scatter plots of precision and recall for the combined PPA-BLAME strategy at a developer level. Table V (columns 8–10) lists the project-level precision and recall. For 20 out of 57 developers, API usages have been precisely identified (precision = 1), however for only 2 developers all their API usages have been successfully identified (recall = 1). Precision values at the project level are above 86% across subjects, and recall is in the range of 48% to 61%.

Discussion: In this experiment we evaluated the imperfection introduced as a combination of both partial program analysis for extracting API usages and assignment of API usage to developers using the blame heuristic. We anticipated a loss of precision and recall when compared to CPA-BLAME (analysed in RQ2). Interestingly, the precision numbers do not decrease drastically and the precision, recall values are similar to combining CPA with blame, across all the projects. This emphasizes the observation that PPA is a viable alternative for CPA for API usage determination from source code, irrespective of the API usage assignment strategy.

IV. THREATS TO VALIDITY

In this section we enumerate some factors which might threaten the validity of our experiments.

Truth Set: The generation of the ground truth of CPA-COMMITS depends on the *compilable program assumption*. That is, we assume that at each version the source code is in a compilable state such that we can accurately resolve type bindings for extracting API usages. If some files have been checked-in with syntactic errors or with non-compliant code[4], our ground truth would have missed these API references. However, these API references could have been extracted at the latest version when the issues were fixed and hence they might show up in the *blame*-based approaches. In these cases, some of the false positives we reported might in fact be true positives, while some reported true negatives might actually be false negatives.

[4]Our analysis was restricted to Java 1.6 compliance since this was a constraint on the latest implementation of PPA

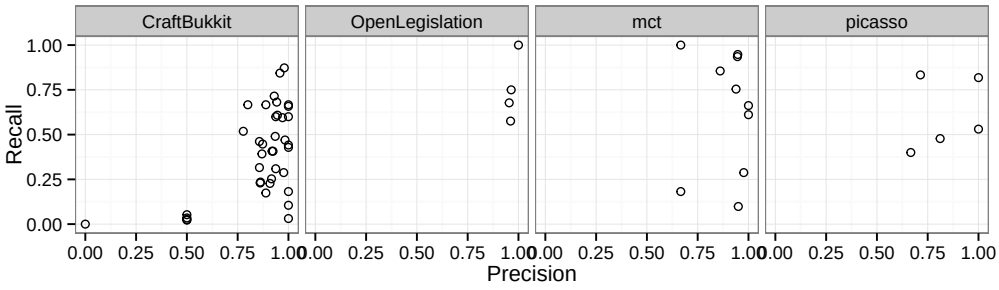

Fig. 5. Scatter plot of precision and recall per developer for PPA-BLAME

File renames and code refactoring: We do not track file renames in either our *blame*-based or change-set walking approaches. In our experiments, the renaming or moving of a file would appear as a deletion of the old file with the addition of a new file. Similarly, we do not explicitly identify commits that only perform refactoring. In the future, we plan to use automated tool support for pruning non-essential commits [12].

External validity: Our experiments were focused on evaluating PPA in the context of Java projects. Hence our results cannot generalize over non-Java projects. At this time we are not aware of partial program analysis techniques for dynamically typed languages such as Python or JavaScript. Further, within Java projects, we only used Java 6 compliant projects as the implementation of PPA we used is implemented for Java 6 and less. In order to replicate these results for Java 7 and beyond, the current implementation of PPA would need to be extended.

V. Related Work

In prior work [5], we proposed a system for generating social recommendations for developers by mining large numbers of project repositories. We also demonstrated a prototype, APINet, which generated recommendations for over 5000 developers whose profiles were mined from 568 Java projects hosted on GitHub. We observed that in cases where developers contributed to multiple projects, these projects often used similar libraries. This motivated our explorations of API usage expertise mining as we believe that the knowledge of a library API is a transferable skill that can be leveraged to seed various recommender systems in a multi-project scenario.

The differentiation between *implementation expertise* and *usage expertise* was first made by Schuler and Zimmerman [4], who extracted method invocations from CVS repositories of the Eclipse project. They, too, noted the potential difficulty in generating typed representations using build and link tools and thus only reported usage expertise in the form of unqualified method identifiers.

Several static analysis tools have been developed to handle incomplete programs. Example include *partial data flow analysis* [13] and *fragment class analysis* [14]. However, these techniques are mainly geared towards enabling sound analysis of parts of a program using summary information for the incomplete portions. Since we were mainly interested in resolving type bindings from code that may have syntactic ambiguities with a tolerance for slight imprecision, we chose partial program analysis [7] (PPA) as the tool that best suited our needs.

PPA itself has been used in several applications that analyze code from version histories [10]–[12]. However, we are not aware of any other study that specifically evaluated the effectiveness of PPA in terms of the results generated by the application as compared to the ideal output. Also, since the original evaluation of PPA [7] reported results based on qualified type inferencing, we could not extend those results directly for our application of expertise profile generation which depends on resolving fully qualified types.

The tool that we found came closest to PPA was Baker [15]. This tool is mainly aimed at extracting API references from snippets of code found on the Web (such as on Q&A forums) for the purpose of bi-directional linking of appropriate API documentation. Baker itself uses PPA, but resolves ambiguities by employing a constraint solving technique with the help of an oracle, which is basically a dictionary of millions of method signatures harvested from JARs of commonly used Java libraries. While this approach works very well for code snippets, its current design is not particularly suited for expertise mining within a project. The reason for this is that unlike PPA, which only qualifies class names using information from `import` statements, Baker assumes that the input is missing `imports` and tries to find a match from its large database. For example, when we asked Baker to analyze the code in Figure 1, it guessed that `DatabaseManager` possibly belongs to the package `org.nuiton.topia.migration`, since a class with the same name exists in that library and also has a method called `getConnection` which returns an object of type `Connection`. As our application of expertise identification is more sensitive toward precision rather than recall, we did not use Baker as-is. However, personal communication with the author revealed that it is possible to modify Baker to take advantage of the fact that the input is an entire file and not just a snippet of code. It remains to be seen how the oracle-based approach of such a customized version of Baker compares with the default PPA implementation for the purposes of API extraction within a project.

TABLE VI
SUMMARY OF DESIGN CHOICES

Libraries Available?	Resource Constraints?	Proposed Approach
Yes	No	CPA-COMMITS
Yes	Yes	CPA-BLAME
No	No	PPA-COMMITS
No	Yes	PPA-BLAME

VI. CONCLUSION AND FUTURE WORK

In this paper, we evaluated various design choices for generating external API expertise profiles of developers by analyzing their code contributions in version management systems. The techniques were: (1) complete program analysis on each change-set, (2) partial program analysis on each change-set, (3) complete program analysis on the latest snapshot with blame information, and (4) partial program analysis on the latest snapshot with blame information. Table VI summarizes our design choices and the constraints for which each choice was considered. We evaluated these design choices on four open source Java 6 projects available on GitHub.

We found that performing partial program analysis as opposed to complete program analysis on complete change history leads leads to a precision of 85–90% and a recall of 90–99% in the generated expertise profiles. This makes us reach the conclusion that for extracting API usage profiles, partial program based analysis provides a feasible alternative solution to complete program analysis. However, when we move from processing complete change history to processing code at a particular snapshot and use *blame* for expertise assignment, there is a significant recall loss. For the complete program analysis technique itself this recall loss varied from 34% to 52% and with partial program analysis between 39% to 52%. Thus, for the purposes of expertise assignment, the blame-based solution may not be acceptable. A trade-off would need to be made with respect to achieving higher levels of recall and taking into consideration practical resource constraints. The expertise profiles generated in our experiments are available at http://code.comprehend.in:8080/ppa-expertise.

This work sets the foundation for several interesting threads for future work. Firstly, the problem of extracting API usages is even harder in dynamically typed languages such as JavaScript and Ruby. Secondly, in this paper we restricted ourselves to identifying third-party APIs referenced by developers. However, does this really translate to a developer's expertise? Or do we need additional information about the manner in which developers use APIs? For example, is a developer more of an expert in an API if they have used the same API multiple times? Or is a developer's use of multiple methods in a class (or multiple classes within a package) a better indicator of their expertise? And does a developer's expertise decrease over time if they do not use an API? We plan to explore these factors via qualitative user studies in future work.

REFERENCES

[1] A. Mockus and J. D. Herbsleb, "Expertise browser: A quantitative approach to identifying expertise," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02. New York, NY, USA: ACM, 2002, pp. 503–512. [Online]. Available: http://doi.acm.org/10.1145/581339.581401

[2] J. Anvik and G. C. Murphy, "Determining implementation expertise from bug reports," in *Proceedings of the Fourth International Workshop on Mining Software Repositories*, ser. MSR '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 2–. [Online]. Available: http://dx.doi.org/10.1109/MSR.2007.7

[3] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A degree-of-knowledge model to capture source code familiarity," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 385–394. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806856

[4] D. Schuler and T. Zimmermann, "Mining usage expertise from version archives," in *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, ser. MSR '08. New York, NY, USA: ACM, 2008, pp. 121–124. [Online]. Available: http://doi.acm.org/10.1145/1370750.1370779

[5] R. Padhye, D. Mukherjee, and V. S. Sinha, "API as a social glue," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE Companion 2014. New York, NY, USA: ACM, 2014, pp. 516–519. [Online]. Available: http://doi.acm.org/10.1145/2591062.2591115

[6] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 466–480, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1109/TSE.2005.63

[7] B. Dagenais and L. Hendren, "Enabling static analysis for partial Java programs," in *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, ser. OOPSLA '08. New York, NY, USA: ACM, 2008, pp. 313–328. [Online]. Available: http://doi.acm.org/10.1145/1449764.1449790

[8] Eclipse Foundation, "Java Development Tools," http://www.eclipse.org/jdt/.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '99. IBM Press, 1999, pp. 13–. [Online]. Available: http://dl.acm.org/citation.cfm?id=781995.782008

[10] B. Dagenais and M. P. Robillard, "SemDiff: Analysis and recommendation support for API evolution," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 599–602. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070565

[11] K. S. Herzig, "Capturing the long-term impact of changes," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 393–396. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810401

[12] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: ACM, 2011, pp. 351–360. [Online]. Available: http://doi.acm.org/10.1145/1985793.1985842

[13] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical framework for demand-driven interprocedural data flow analysis," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 6, pp. 992–1030, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/267959.269970

[14] A. Rountev, A. Milanova, and B. G. Ryder, "Fragment class analysis for testing of polymorphism in Java software," *IEEE Trans. Softw. Eng.*, vol. 30, no. 6, pp. 372–387, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1109/TSE.2004.20

[15] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 643–652. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568313