Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study

Davy Landman*, Alexander Serebrenik*†, Jurgen J. Vinju*†
* Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{Davy.Landman, Jurgen.Vinju}@cwi.nl
† Eindhoven University of Technology, Eindhoven, The Netherlands
{a.serebrenik,j.j.vinju}@tue.nl

***Abstract*—The behavior of software that uses the Java Reflection API is fundamentally hard to predict by analyzing code. Only recent static analysis approaches can resolve reflection under unsound yet pragmatic assumptions. We survey what approaches exist and what their limitations are. We then analyze how real-world Java code uses the Reflection API, and how many Java projects contain code challenging state-of-the-art static analysis.**

Using a systematic literature review we collected and categorized all known methods of statically approximating reflective Java code. Next to this we constructed a representative corpus of Java systems and collected descriptive statistics of the usage of the Reflection API. We then applied an analysis on the abstract syntax trees of all source code to count code idioms which go beyond the limitation boundaries of static analysis approaches. The resulting data answers the research questions. The corpus, the tool and the results are openly available.

We conclude that the need for unsound assumptions to resolve reflection is widely supported. In our corpus, reflection can not be ignored for 78 % of the projects. Common challenges for analysis tools such as non-exceptional exceptions, programmatic filtering meta objects, semantics of collections, and dynamic proxies, widely occur in the corpus. For Java software engineers prioritizing on robustness, we list tactics to obtain more easy to analyze reflection code, and for static analysis tool builders we provide a list of opportunities to have significant impact on real Java code.

***Keywords*-Java; Reflection; Static Analysis; Systematic Literature Review; Empirical Study**

I. Introduction

Static analysis techniques are applied to support the efficiency and quality of software engineering tasks. Be it for understanding, validating, or refactoring source code, pragmatic static analysis tools exist to reduce error-prone manual labor and to increase the comprehension of complex software artefacts.

Static analysis of object-oriented code is an exciting, ongoing and challenging research area, made especially challenging by dynamic language features (a.k.a. reflection). The Java Reflection API allows programmers to dynamically inspect and interact with otherwise static language concepts such as classes, fields and methods, e.g., to dynamically instantiate objects, set fields and invoke methods. These dynamic language features are useful, but their usage also wreaks havoc on the accuracy of static analysis results. This is due to the undecidability of resolving dynamic names and dynamic types.

Until 2005, the analysis of code which uses the Reflection API was considered to be out of bounds for static analysis, and handled via user annotations or dynamic analysis; handling reflection would inherently be either unsound (due to unverified assumptions) or highly inaccurate (due to over-approximation) and render the contemporary static analysis tools impractical. Then, in 2005 Livshits *et al.* [1] published an analysis of how reflection was used in six large Java projects, proposing three unsound, yet well-motivated assumptions and using these to (partially) statically resolve the targets of dynamic method calls. Since then more tools were based on similar assumptions.

Very recently, in 2015, Livshits and several other authors of static analysis tools published the soundiness manifesto [2]. It argues for "soundy" static analysis approaches that are mostly sound, but pragmatically unsound around specific problematic language features. Java's Reflection API is one of the examples that can be handled more effectively after certain unsound assumptions are made. For future work they identified the need for empirical evidence on how these language features are used, such that tool builders can motivate the required unsound assumptions. We provide more unbiased empirical evidence on the use of reflection by focussing on the following **Main Research Question:** *What are limits of state-of-the-art static analysis tools when confronted with the Reflection API and how do these limits relate to real Java code?*

Hence, we investigate the following sub-questions:

SQ1. How do static analysis approaches handle reflection; which limitations exist and which assumptions are made? (Section III)

SQ2. How often are different parts (see Section II) of the Reflection API used in real Java code? (Section IV)

SQ3. How often does real Java code challenge the limitations and assumptions identified by SQ1? (Section V)

Together with answers to these questions, this paper contributes a representative corpus of open-source Java projects [3], and a comprehensive literature overview on the relation between static analysis and Java reflection. The main question is answered with a list of challenges and suggested tactics for static analysis researchers, ordered by expected impact.

II. The Java Reflection API

We first describe the Java Reflection API; how its features can be categorized. The resulting frame of reference is used for the interpretation of the findings in Sections III–V, because the different API features interact differently with static analysis.

```
<MetaObject> ::= <Class> | <Method> | <Constructor> | <Field>
<Member> ::= <Method> | <Constructor> | <Field>

<ClassLoader> ::=
TM      <Class>.getClassLoader()
LM    | ClassLoader.getSystemClassLoader()
LM    | new ClassLoader(<ClassLoader>)
LM    | <ClassLoader>.getParent()

<Class> ::=
LC      Class.forName(<String>)
LC    | Class.forName(<String>, <Boolean>, <ClassLoader>)
LC    | <ClassLoader>.loadClass(<String>)
LM    | <Type>.class
LM    | <Object>.getClass()
TM    | <Class>.get*Interfaces()
TM    | <Class>.asSubclass(<Class>)
TM    | <MetaObject>.get*Class{es}?()
TM    | <MetaObject>.get*Type*()
P     | Proxy.getProxyClass(<Class*>)

<Method> ::=
TM      <Class>.get{Declared}?Methods()
TM    | <Class>.get{Declared}?Method(<String>, <Class*>)
TM    | <Class>.getEnclosingMethod()

<Constructor> ::=
TM      <Class>.get{Declared}?Constructors()
TM    | <Class>.get{Declared}?Constructor(<Class*>)
TM    | <Class>.getEnclosingConstructor()

<Field> ::=
TM      <Class>.get{Declared}?Fields()
TM    | <Class>.get{Declared}?Field(<String>)

<Void> ::=
M       <Field>.set*(<Object>, <Object>)
AR    | Array.set*(<Object>, <int>, <Object>)
MM    | <Member>.setAccessible(<Boolean>)
AS    | <ClassLoader>.{set}?{clear}?*AssertionStatus(<Boolean*>)
AS    | <ClassLoader>.set*AssertionStatus(<String>, <Boolean>)

<Object> ::=
C       <Constructor>.newInstance(<Object*>)
C     | <Class>.newInstance()
AR    | Array.newInstance*(<Class>, <int*>)
P     | Proxy.newProxyInstance(<ClassLoader>, <Class*>, <Object>)
I     | <Method>.invoke(<Object>, <Object*>)
A     | <Field>.get*(<Object>)
AR    | Array.get*(<Object>, <int>)
DC    | <Class>.cast(<Object>)
AN    | <Method>.getDefaultValue()
TM    | <Class>.getEnumConstants()
P     | Proxy.getInvocationHandler(<Object>)
AN    | <MetaObject>.getAnnotation(<Class*>)
AN    | <MetaObject>.get*Annotations()
S     | <Class>.getSigners()

<ProtectionDomain> ::= S <Class>.getProtectionDomain()

<Boolean> ::=
SG      <Class>.isAssignableFrom(<Class>)
SG    | <Class>.isInstance(<Class>)
SG    | Proxy.isProxyClass(<Class>)
SG    | <MetaObject>.is*(<Class>) // other signature checks
SG    | <MetaObject>.equals(<Object>)
SG    | <MetaObject> == <MetaObject>
SG    | <MetaObject> != <MetaObject>
SG    | <Member>.isAccessible(<Class>)
AS    | <Class>.desiredAssertionStatus()
AN    | <MetaObject>.isAnnotationPresent(<Class>)

<String> ::=
ST      <MetaObject>.get*Name()
ST    | <MetaObject>.to*String()
ST    | <Class>.getPackage() // returns a wrapper for strings

<int> ::= SG <MetaObject>.getModifiers()

<Resource> ::= <URL> | <InputStream>
RS    | <Class>.getResource*(<String>)
RS    | <ClassLoader>.get*Resource*(<String>)
```

Figure 1. Grammar of the Java Reflection API. A '*' inside a terminal indicates zero or more other characters, and inside a nonterminal it indicates zero or more of this nonterminal. {X}? indicates an optional part of a terminal. MethodUtil.getMethod* was elided into the - non deprecated - replacement method.

The Java Reflection API consists of objects modeling the Java type system. These meta objects are split over 8 classes - java.lang.{Class,ClassLoader} and java.lang.reflection.{Array,Constructor}, {Field,Member,Method,Proxy} - totaling 181 public methods. The meta objects mostly provide an immutable view of the running system's types.

Figure 1 summarizes the API as a context-free grammar that defines construction of references to meta objects. We use a context-free grammar as a more concise alternative to class diagrams or interface definitions. Each production in Figure 1 defines a number of alternatives to produce an object of the defined non-terminal. The grammar naturally groups on return type to emphasize the construction of (immutable) meta objects and compresses methods of similar intent using regular expressions. With this, we completely mapped the 181 public methods of the entire API onto 58 productions, which are further grouped into 17 functional categories in Table I.

Next to the API listed in the java.lang.reflection package, there is: the Object.getClass() method and the literal Object.**class** language construct for class literals. There is also relevant Java expression syntax related to reflection, casts and **instanceof**. Class literals, such as MyClass.**class**, produce a meta object instance of the (static) type Class<MyClass>. They are a static alternative to Object.getClass. Cast and **instanceof** expressions also use literal types which interact with Java's execution semantics (e.g. throwing ClassCastException).

From the perspective of static analysis, the reflection API introduces dynamic language features for an otherwise statically resolved language. From this perspective, the API can be split in two parts. The first part (□ in Table I) are the *Dynamic Language Features* that simulate statically resolved counterparts: e.g., the <Method>.invoke API is the dynamic equivalent of the method invocation in Java. The second part (⬚) includes supporting methods for the dynamic language features (e.g., getting a Method meta object), and miscellaneous methods for accessing other elements of the Java runtime.

Even when infrequently used, a single occurrence of using a dynamic language feature does complicate static analysis of the entire program. For example, a single dynamic method invocation could in principle call any method in the currently loaded system, resulting in a highly inaccurate call graph for the entire system. For the rest of this paper, we are primarily interested in how static analyses approximate the effect of these dynamic language features.

Table I
CATEGORIES FOR REFLECTION PRODUCTIONS.

Category	Description
LC Load Class	Entry to the Reflection API, returns references to meta objects from a String. Considered harmful since it can execute static initializers.
LM Lookup Meta Object	Non harmful entries to the Reflection API, returns references to meta objects.
TM Traverse Meta Object	Get references to other meta objects related to the current meta object in the type system of Java.
C Construct Object	Create a new instance of an object, equivalent to the `new <ClassName>()` Java construct.
P Proxy	Proxies are fake implementations of interfaces, where every invoke is translated to a single callback method. Very harmful for static analysis, since there is no static equivalent for this feature.
A Access Object	Read the value of an Object's field. Equivalent to the `obj.field` Java construct.
M Manipulate Object	Change the value of a field. Equivalent Java construct: `obj.field = newValue`
MM Manipulate Meta Object	The only *mutable* part of the API: changing access modifiers.
I Invoke Method	Invoke an method. Equivalent Java construct: `recv.method(args)`.
AR Array	Create, access, and manipulate arrays.
SG Signature	Test the signature of a Meta Object, for example if it is a public field.
AS Assertions	Access and manipulate the assertion flag per class.
AN Annotations	Access and iterate annotations.
RS Resources	Read resources using the ClassLoader.
ST String representations	Get the name of the meta object's elements.
S Security	Security related calls
DC Casts	Cast to a dynamically `Class` meta object. Equivalent Java construct: `(Class)obj`

The ☐ categories represent core *Dynamic Language Features* which simulate statically resolved counterparts.
The ⬚ categories represent supporting APIs comparable to normal Java library code.

Modeling the supporting methods is often necessary to approximate the semantics of the dynamic language features. For example, invoking a method requires a `Method` meta object. Finding meta objects (with the exception of the LC productions) does not complicate static analysis on its own. It is merely an inspection of the type system. These methods can be either simulated by static analysis tools, or directly executed.

The pinnacle of dynamic behavior are `Proxy` classes P. The dynamic proxy feature allows one to instantiate objects - statically implementing a specific interface - that will dynamically forward all calls to a generic `invoke` method of another object (implementing the `InvocationHandler` interface). The proxy feature hides dynamic method invocation under a normal statically checked virtual method interface, rendering all virtual method invocations possibly dynamic.

III. STATIC ANALYSIS OF REFLECTION IN THE LITERATURE

To answer how reflection is handled by static analysis approaches (SQ1) we conduct a literature review. The result of the review is a list of techniques and associated properties of hard to analyse code which identify limitations and assumptions of static analysis tools. Note that the results of this review can not serve as a feature comparison between static analysis tools, because of different goals of those tools and because of our focus on the Reflection API, rather than the entire Java language.

A. Finding and selecting relevant work

Two commonly used literature review techniques are snowballing [25], [26] and Systematic Literature Review (SLR) [27]. Snowballing consists in iteratively following the citations of a small collection of serendipitously identified papers. However, several core papers have hundreds of citations, e.g., the work of Felt *et al*. [16] has been cited 940 times, rendering snowballing too labor intensive. Hence, we conduct an SLR.

1) Initial queries: As recommended by Kitchenham and Charters [27] we started by considering IEEE Xplore, ACM DL, and ScienceDirect. The search results, however, contained multiple inconsistencies. In IEEE Xplore, e.g., adding an `OR` to our query reduced the number of results. ACM DL and ScienceDirect search missed papers when limited to the abstract field, even though those abstracts contained the search terms. Hence, we decided that these sources were not well-suited for SLR. Instead, we opt for Google Scholar as it provides a wide coverage of different electronic sources as recommended [27] and its search engine did not exhibit these peculiarities.

Following the PICO criteria [28] we define our *population* as Java projects with reflection, *intervention* as static analysis and *outcomes* as approach, limitations and assumptions. We do not explicitly state the *comparison* element of PICO since our goal consists in comparing different ways reflection is handled by static analysis techniques *with each other* as opposed to comparing them with a predefined control treatment. Based on the population, intervention and outcome we formulate the following query: `java "static analysis" +reflection`. We do not explicitly include the outcome in the query since approaches, limitations and assumptions can be phrased in numerous ways. In October 2015 the query returned 4 K references.

2) Automatic selection criteria: Since manual analysis of 4 K documents is infeasible, we design six criteria to reduce the number of potentially relevant documents. To be included in the study the document should meet *at least* one of those criteria. Those criteria, presented in Table II, are based on frequency of keywords in the full text, the first 10% of the text (head), the last 10% (tail), and the last 10% without the references/bibliography (tail without references). We validated all thresholds of these criteria by sampling beyond the thresholds and manually scanning the additional papers for false negatives. We picked liberal thresholds to optimize on recall (e.g., $P \leq 80$ for deciding a document is a single paper rather than a collection).

3) Manually Improving Accuracy: 478 documents (11% of the original set) were matched by at least one of the six criteria in Table II. Including the 36 documents that `pdf2text` failed to analyse we had 514 documents to read. We reviewed all documents applying the practical screen [29] to exclude those meeting the following *exclusion* criteria: not about Java, not about static analysis, reflection is only recognized as a limitation, reflection is handled with an external tool, reflection is wrapped to guard against its effects, reflection

Table II
INCLUSION CRITERIA USED TO SELECT RELEVANT DOCUMENTS FOR MANUAL REVIEW.

1) Papers with reflection in introduction (head) and conclusion (tail). Moreover, at least one term related to accuracy should be used. To correct for Google's stemming of JavaScript to Java, we exclude papers that mention JavaScript too often: $P \leq 80 \wedge R_h > 0 \wedge (R_t > 0 \vee R_{t'} > 0) \wedge A > 0 \wedge S \leq 5$.

2) Thesis. A thesis discussing reflection, containing reflection code samples, and mentioning accuracy: $P > 50 \wedge T_h > 0 \wedge R > 1 \wedge A > 0 \wedge J > 0$.

3) Proceedings with frequent mentions of reflection: $P > 20 \wedge T_h = 0 \wedge C_h > 0 \wedge R > 5$.

4) Short papers frequently mentioning reflection. Smaller documents might have non standard layout, or be sensitive to the 10% cutoff points for the head and tail. These documents mentioning reflection at least 10 times are also included: $P \leq 40 \wedge R \geq 10 \wedge A > 0 \wedge S \leq 5$.

5) Proceedings with reflection code samples. Similarly to 3) but with reflection code samples: $P > 20 \wedge T_h = 0 \wedge C_h > 0 \wedge R > 0 \wedge J > 0$.

6) Large non-thesis, non-proceedings papers with frequent reflection: $P > 80 \wedge T_h = 0 \wedge C_h = 0 \wedge R > 5$.

The $\star_h$ denotes head section, $\star_t$ tail section , $\star_{t'}$ tail section without bibliography, and P amount of pages in a PDF. A represents terms related to "accuracy", "precision" and "soundness", C for "proceedings" and "conference", J for "lang.reflect", R for "reflection", S for "javascript", and T for "thesis" and "dissertation".

Table III
STATIC ANALYSIS APPROACHES FOR HANDLING REFLECTION. FOR OBJECT AND CONTEXT SENSITIVITY WE REPORT THE SENSITIVITY DEPTH. FOR THE STRINGS COLUMN: ○ NO ANALYSIS, ◔ ONLY LITERALS, ◓ LITERALS AND CONCATENATIONS, AND ● FULL FLEDGED (JSA) STRING OPERATIONS. FOR THE REMAINING PROPERTIES WE USE FILLED CIRCLES TO SUMMARIZE THE COVERAGE OF A PROPERTY: ○ FOR NONE, ◓ FOR PARTIAL, AND ● FOR FULL. THE TABLE IS SORTED ON THE "BUILD USING" AND "YEAR" COLUMNS.

Paper	Year	Tool	Related	Kind	Goal	Sensitivity[y]				Inter-procedural	Fixed-point	Strings	Casts	Meta-Objects	Dependency
						flow[z]	field	object	context						
[1]	2005	bddbddb		Static & Annotations	Call Graph[a]	○	○	0	0	○	●	◓	◓[k]	●	Datalog & bddbddb
[4]	2009	Doop	[1], [5]	Static	Points to	●[b]	●	0	1, 2	●	●	◔[c]	○	○	Datalog
[6]	2013	Datalaude	[1]	Static	Points to	○	○	0	0	●	●	◔	○	◓	Maude & Joeq
[7]	2014	Elf	[4]	Static	Points to	●[b]	●	0	1, 2	●	●	◓	●	○	Doop
[8]	2015	Solar	[7]	Static & Annotations	Points to	●[b]	●	0	1, 2	●	●	◓	●	●[d]	Doop & Elf
[9]	2015		[4]	Static	Points to	●[b]	○	1	1	●	●	◔	○	●	Datalog
[10]	2015	Doop	[4]	Static	Points to	●[b]	●	0	1, 2	●	●	◓[e]	●[e]	●[e]	Datalog
[11]	2003	JSA		Static	Call Graph	●[b]	●	0	0	●	○	●	○	○	Soot
[12]	2007		[11]	Static & Dynamic	Class Loading	●[b]	●[f]	0	0	●	○	●[g]	○	○	Soot & JSA
[13]	2009		[12]	Static & Dynamic	Class Loading	●[b]	●[f]	0	0	●	○	●[g]	○	◓	Soot & JSA
[14]	2013	Averroes		Static & Dynamic	Modeling API	○	○	0	0	○	○	◔	○	○	Soot & TamiFlex
[15]	2007	ACE		Static & Dynamic	Call Graph	○	○	1	1	●	○	○	◓[k]	○	
[16]	2011	Stowaway		Static	Name	●	○	0	0	◓	○	◔	○	●	
[17]	2012	ScanDal		Static	Taint	●	○	0	1	●	○	◔	○	○	
[18]	2013		[16]	Static	Name	●[h]	○	0	∞[h]	●[i]	○	◔	○	◓	
[19]	2014			Static	CFG	●	○	0	0	●	○	◔	○	○	
[20]	2014	FUSE		Static	Points to	●[b]	0	0	0	●	○	○	◓[k]	○	
[21]	2015	WALA		Static	Multiple	●[b]	●	0/∞	0/∞	●	●	◓	●	◓	
[22]	2015	part of SPARTA	[23]	Static & Annotations	Implicit CFG	●	○	0	0	○	○	◓	○	●	Checker Framework
[24]	2015	EdgeMiner		Static	Implicit CFG	○	○	0	0	●	○	◔[j]	○	○	dx

a) Including points-to analysis.
b) After SSA transform.
c) Only for `Class.forName`.
d) Lazy
e) Only if it points to a small set of candidates (subclasses / fields / methods).
f) Only string fields.
g) JSA extended with environment information, modeling field, and tracking of objects of type Object.
h) Backwards slicing.
i) With heuristics.
j) Only for base (JRE/Android) framework.
k) Only for `newInstance`.
y) None of the papers are path sensitive.
z) The reported flow sensitivity was always intra-procedural.

Table IV
REPORTED OPEN AND RESOLVED LIMITATIONS OF STATIC ANALYSIS TOOLS, USING LITERATURE FROM TABLE III.

Name	Description
CorrectCasts [1]	Assumption that casts never throw `ClassCastException`
WellBehavedClassloaders [1]	Assumption that all `ClassLoaders` implementations follow a specific contract, i.e., if a class with the (fully qualified) name X is requested from the LC API then a reference to a class named X is produced
ClosedWorld [1]	Assumption that the classpath configured for static analysis equals that of the analysed program
IgnoringExceptions [4]	Not modeling the control effect of exceptions, which is relevant around common exceptions of the Reflection API (e.g., `ClassCastException`)
InaccurateIndexedCollections [4]	Not modeling index positions in arrays and lists, which is relevant when meta objects end up in such collections
InaccurateSetsAndMaps [13]	Not modeling `hashCode` and `equals` semantics in concert with hash collections, which is relevant when meta objects end up in such collections
NoMultipleMetaObjects [7]	Ignoring usage of TM API methods which return multiple meta objects in an array
IgnoringEnvironment [12]	Not modeling the content of configuration strings which come from `System.getEnv` for tracing LC, LM or TM methods
UndecidableFiltering [16]	Conditional control flow and arbitrary predicates are hard in general, while for code which filters meta objects even an approximate answer would greatly help
NoProxy [7]	Assumption that Proxy objects are never used. Proxy objects may invoke dynamically linked code opaquely behind any (dynamic) interface, undermining otherwise trivial assumptions of static analysis of method calls

is used to solve a problem, or a homonym of "reflection" was the cause of the match. We have logged the exclusion decisions in a shared online spreadsheet and reviewed each others decisions. This process produced 50 documents. Next we removed non-peer-reviewed publications: locating and substituting conference papers for equivalent technical reports, masters theses or PhD theses; locating and substituting extended journal versions for conference papers; removing non-peer-reviewed publications such as technical reports and masters thesis' without corresponding publications at the time; and finally as recommended by Kitchenham and Charters [27] merging duplicate documents produced by noise in Google Scholar. This results in 39 documents.

All 39 documents were then read by one author and scanned by another, producing 4 new relevant documents from the citations (all missing from the original Google Scholar results). The 4 new papers introduced Soot [30], Spark [31] (a plugin for Soot), WALA [21], and JSA [11]. Only JSA and WALA handle reflection specifically, while Soot requires plugins (such as TamiFlex [32] or Spark), and Spark requires user annotations.

While reading the documents we applied the methodological quality screen [29] and identified another 10 documents to be excluded, due to the following reasons: taint analysis pushing taints through the reflection API [33], [34], using existing techniques for handling reflection [14], [35]–[40], and handling reflection in generated bytecode rather than in source code [41].

B. Documenting Properties of Static Analysis Tools

To answer SQ1, we read the 33 ($39 + 4 - 10$) documents to list approaches or techniques which are involved in resolving dynamic language features of Java reflection. The end result is summarized in Table III. When we could not find enough information to extract information about the properties of a tool from the respective paper, we analysed the latest version of the tool's source code and documentation (if available). As recommended by Brereton *et al*. one author extracted the data, and another one checked it [42].

We classified the techniques in three kinds of analysis, different in the kind of information which is used to resolve reflection: *static* uses code analysis to resolve reflection (listed in Table III), *dynamic* uses information acquired at run-time for resolving reflection rather than code ([30], [32], [43]–[47]) and *annotations* groups techniques based on are human-provided meta data rather than code or dynamic analysis ([31], [48]–[51]). Note that papers solely about dynamic analysis were excluded in an earlier stage.

Next we record the **goal** of the static analysis as mentioned in the paper (e.g., call graph construction), the name of the tool, and possible **dependency** on other related tools. We also distinguish between intra- and **inter-procedural** algorithms.

Diving further into the explanations of techniques of each static analysis tool revealed a diverse collection of mostly incomparable algorithms and heuristics in terms of functionality and quality attributes. Based on this reading we documented the authors' descriptions of properties of the analysis tools in terms of **sensitivity**. Sensitivity defines the smallest level of distinction made by the abstract (symbolic) representations of run-time values and run-time effects that static analysis tools use. Finer-grade distinctions mean more different abstract values and result in more accurate but slower analyses, while coarser-grade distinctions lead to less different abstract values and less accurate but faster analyses.

Flow sensitivity entails distinctions between subsequent assignments

Field sensitivity entails distinction between different fields in the same object

Object sensitivity entails the distinctions between individual objects, via groups of objects, to general class types, at increasing levels of indirection

Context sensitivity entails the distinction of method executions between different calling contexts of a given depth

We also record whether the analysis requires a **fixed-point** computation. Finally we identified and documented the use of three specialized measures taken by static analysis tools:

String analysis approximates run-time values of strings as accurately as possible. These results can then be used to approximate class and method names which flow into the LC, TM reflection API, after which the semantics of `invoke` and `newInstance` may be resolvable.

Casts provide information about run-time types under the assumption that no `ClassCastException` occurs. Some analyses also reason *back* from the correct-casts assumption.

MetaObjects signifies the full simulation (or execution) of the LC, LM, and TM reflection API to find out which meta objects may flow into the dynamic language features.

By inspecting Table III we observe that flow sensitivity is very common (often as a side-effect of the SSA transform), field sensitivity is used for half of the approaches (more common in Doop and Soot), and, most analyses are inter-procedural and track at least string literals. Tracing Doop through the years, we see more modeling of Strings, Casts and Meta Objects.

C. Self-reported limitations and assumptions

The self-reported assumptions about actual code and limitations of the tools are summarized in Table IV. All tools discussed in the 33 studies assume well-behavedness of `ClassLoader` implementations and absence of `Proxy` classes. The other reported limitations are either resolved and fixed by a given paper, or mentioned as a known limitation of the currently described approach. We do not provide a feature comparison per tool, but rather report "common" assumptions made by static analysis tools. We choose not to extend Table IV with how many tools use each assumption, to avoid it being interpreted as a (crude) comparison between incomparable tools.

SQ1: State-of-the-art static analysis tools use inter-procedural, flow and field sensitive analysis. Some explicitly model Strings, Casts and Meta Objects. All tools assume well-behavedness of `ClassLoader` implementations and absence of `Proxy` classes. The techniques and their limitations are summarized in Tables III and IV.

IV. How often is the reflection API used?

Regardless of the conceptual relation between reflection and static analysis, we need support for the relevance of this relation in real Java code to answer SQ2 and motivate further investigation.

Table V summarizes the related work found during the review (Section III) reporting empirical observations of reflection usage. From these reports we *hypothesize that also in arbitrary Java code the usage of reflection is widespread*. This is likely true, but it may not be deduced from the reported numbers in Table V, since these studies have been done on corpora selected and filtered for answering different questions.

In particular focusing only on large corpora of Android apps would not be acceptable for our current study since they are an identifiable subgroup of all Java applications. Also the much smaller SPECjvm[1] or DaCapo [52] benchmarks have been compiled to reflect typical performance characteristics of (concurrent) Java programs rather than be representative of the usage of reflection.

[1]https://www.spec.org/benchmarks.html#java

Table V
Empirical observations of reflection in the literature of Table III.

Year	Ref.	Corpus	Report
2005	[1]	6 applications (643 KLOC)	The accompanying technical report discusses reflection use cases, which are used to formulate the three now very popular assumptions.
2011	[16]	900 Android apps	61% use `invoke`. Reflection is also used for serialization, hidden APIs, and backwards compatibility.
2013	[18]	1.3 K Android apps	73% use `invoke`. Primarily for API calls, however, this reflects only 0.07% of all API calls.
2014	[19]	1.7 K Android apps	73% use reflection. `invoke` is most common
2014	[19]	150 Android apps	Analyzing the string argument for `forName` and `getMethod`, 17.30% use only constant strings, 25.30% use a single variable, and 38.70% use more than one variable.
2014	[7]	14 Java programs (DaCaPo benchmark and 3 other applications)	Identified 609 invocations of reflection with Soot, reports popularity of the harmful API, the kind of string operations performed on arguments, and how often the APIs return meta object arrays were used.
2015	[47]	29 K Android apps	81.10% used either `invoke` or `newInstance`
2015	[22]	35 Android apps	142 calls to `invoke`, classifying 81% for backwards compatibility, 6% accessing hidden APIs, and 13% as unknown.

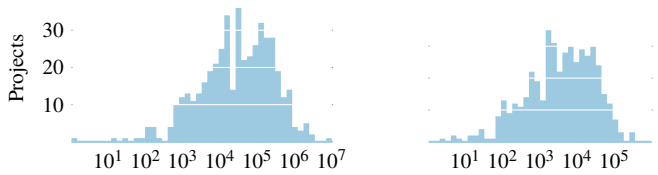
A. Corpus Construction

To test the above hypothesis and answer SQ2 we construct a corpus of the source code of 461 open-source software projects. Hunston has observed that in corpus linguistics the main issues related to corpus design pertain to its size, contents, representativeness and permanence [53]. Tempero *et al*. have argued that the same concerns pertain to software corpora [54].

Contents of the corpus is determined by the research questions we answer using it, i.e., SQ2 and SQ3. Hence, our corpus contains Java programs. Permanence, i.e., regular corpus updates, are considered as future work. Next we discuss how size and representativeness are balanced in our corpus.

1) Selecting projects: To balance the corpus size with representativeness, we construct a corpus small enough to analyze while still covering a wide range of open source Java projects. We use the Software Projects Sampling (SPS) tool [55] by Nagappan *et al*. Given a universe of projects on Ohloh/OpenHub[2], SPS measures representativeness of a smaller *corpus* with respect to the universe in terms of diversity dimensions and constructs a maximally representative corpus by iteratively adding projects that would increase the representativeness most. Diversity dimensions considered include total lines of code, project age (Young, Normal, Old, Very Old), activity (Decreasing, Stable, Increasing), and of the last 12 months, number of contributors, total code churn, and number of commits.

[2]Since the access to the live OpenHub project collection is rate-limited, we used the May 2012 database snapshot when it was still called Ohloh [55].



(a) SLOC (b) Methods

Figure 2. Histograms of projects size (bin width 0.15 on the log X-axis)

The entire collection contains 20 K projects, of which around 3 K have Java recorded as the main language. From this universe the SPS tool identified a sample of 468 projects, maximizing the spread of all diversity dimensions.

We tried to download the source code of the 468 projects. For 33 projects the source code was no longer available. We reran SPS to extend 435 = 468 − 33 projects and maximize the diversity. SPS suggested 27 additional projects. The source code of two of these was not available. Repeating the procedure, SPS suggested one additional project. The resulting 461 = 468 − 33 + 27 − 2 + 1 projects cover 99.47% of the universe.

After downloading the projects we cleaned the corpus by removing arbitrary copies of the code of projects that originate from folder-based version management. Using MD5 hashes to identify full file clones, we manually reviewed and cleaned all projects. We made the cleaned and annotated corpus openly available [3], totaling 79.4 MSLOC of Java code, to be used to reproduce the analysis results, or to benchmark static analysis research tools on systems of documented representativeness. Figure 2 summarizes the corpus in terms of size.

2) Annotated Abstract Syntax Trees: We need a precise count of actual calls into the reflection API, rendering fast *grepping* or other efficient partial parsing methods out of scope due to their inherent inaccuracy [56], [57]. To unambiguously identify the calls to the Reflection API methods we first parsed the source code, resolved names and types, then serialized the Abstract Syntax Trees (ASTs), using the Eclipse Java Development Tools (JDT) and Rascal [58]. We deleted the 4 projects the JDT crashed on (labeled #294, #399, #420, #455)[3].

B. Descriptive Statistics

To describe how the Reflection API is used by the corpus projects we make use of the context-free grammar in Figure 1 and categories of Table I. Per category we count the percentage of projects that make use of at least one production belonging to the category. We aggregate to project level since one instance is enough to complicate static analysis and projects are a common unit for static analysis applications.

Inspecting Figure 3 we observe that reflection is used in almost all the projects (only 4% did not use any reflection). However, there are more use cases for reflection than just dynamic language features. The `<Type>.class` and `<Object>.getClass()` are, for example, often used as a log message prefix. The reported distributions of API method invocations over projects, should be interpreted by tool builders with the API

[3]We opt not to replace these projects as we consider the corpus as a separate contribution independent from the subsequent research.

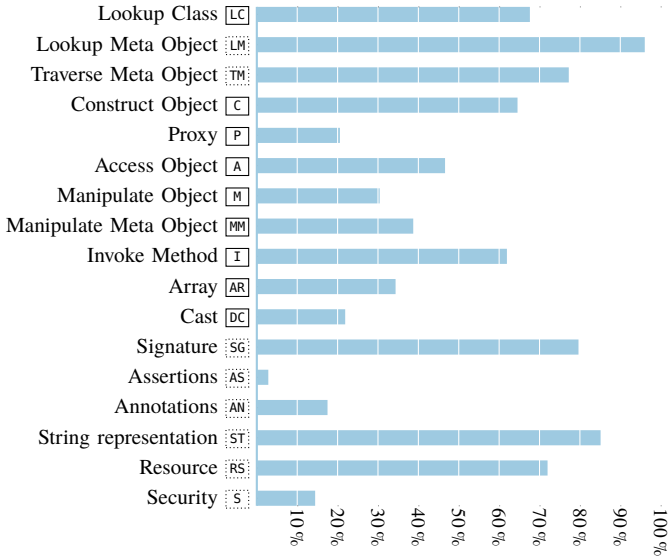


Figure 3. Reflection API usages, grouped per category (Table I), aggregated on project level, 17 projects (3.72%) contained no reflection. 356 projects (77.90%) contained at least one of the dynamic language features (□ categories).

definition itself as a frame of reference, because the API enforces certain data dependencies between different invocations into the API, e.g. `<Method>.invoke` can not be called without first retrieving an instance of an `Method` meta object, which in turn can only come from a `Class` meta object (see Figure 1).

We aggregated all dynamic language features API calls. Of all projects, 78% contain at least one form of these harder to analyze methods of the API. For these projects, a static analysis needs some form of reflection support. Note we only count in the Java source code of a project, reflection usage in its libraries it depends on can only increase the amount of projects impacted by the dynamic language features of reflection.

SQ2: Hard to analyse parts of Reflection API are very common: 78% of all projects contain at least one usage of those.

V. The Impact of Assumptions and Limitations

In this section we answer SQ3: how often the assumptions and limitations in Table IV (Section III) of state-of-the-art static analysis tools are challenged by real Java code. For each identified assumption or limitation of Table IV we devise one or more AST patterns and manually validate their precision in detecting occurrences of challenging code. Then we automatically identify all matches of each pattern in the corpus described in Section IV-A. We reuse the corpus since we look for similar representativeness and need similarly accurate unambiguously resolved classes and methods.

A. Detecting Patterns

To implement pattern detectors we used the builtin AST pattern matching and traversal facilities of Rascal [59], which have been used in many other projects [60]–[62]. The pattern code is around 150 SLOC and it is openly available [63].

Table VI
DESCRIPTIONS OF AST PATTERNS USED TO DETECT THE LIMITATIONS IDENTIFIED IN TABLE IV WITH THEIR RATIONALE.

Name	Pattern description	Pattern rationale
CorrectCasts	`try` blocks with the body calling either `invoke`, `get`, or `newInstance`, and with a `catch(ClassCastException e)` case that neither contains `throw` nor calls a method with either "log" or "error" in the identifier.	Finds code which does not obviously deal with a `ClassCastException` as an unexpected error.
WellBehaved-ClassLoaders	No pattern	This is a deep semantical constraint for which we have no accurate AST pattern
ClosedWorld	No pattern	This is not a code property, as it assumes something about the classpath configuration for the static analysis tool
Ignoring-Exceptions-1	`try` blocks with the body calling any *dynamic language features*, with at least one `catch` block that also calls a *dynamic language feature* method.	Finds code that intents to use reflection in the "normal" path, and continues to use reflection in the "exceptional" path.
Ignoring-Exceptions-2	`for`,`while`,`do` statements with in the body a `try` block with its body using *dynamic language features*, and at least one `catch` block that does not `throw` or call a method with either "log" or "error" in the identifier	Find codes where the exceptional path is the way to continue to the next alternative way of using a *dynamic language feature* which is generated by a loop
IndexedCollections	Any call to a method which retrieves information from the Java Collection API (e.g., `get` & `iterator`) of the containers that allow random indexing or an array access expression where the stored type is a meta object type.	This exactly identifies (using Java's type system) where meta objects may be stored in and retrieved from collections
MetaObjects-InTables	Any call to a method which retrieves values from the Java Collection API's hash-based containers (e.g., HashMap), where the stored type is a meta object type.	This exactly identifies (using Java's type system) where meta objects may be stored in and retrieved from hash collections
Multiple-MetaObjects	A call to the methods in the TM category (Table I) that return arrays of meta objects	This exactly identifies the construct of interest using Java's name resolution
Environment-Strings	Any call to the methods retrieving strings from the environment [13] which are inlined as actual parameters in calls to reflection	Finds trivial flow of information from the environment into the reflection API, and nothing more
Undecidable-Filtering	`for`,`while`,`do` statements with in the body a call to any of the methods in the TM or SG categories (Table I) and a call to any *dynamic language feature*	Probably finds code where filtering is implemented using a loop code idiom, given that it also uses predicates and lookups from the Reflection API.
NoProxy	Any usage of the Reflection API for dynamic proxies	This is an exact query (also used in Section IV).

The patterns we devised are described and motivated in Table VI. We strive for high precision for each pattern (a low number of false positives). Each AST pattern will capture "typical" code instances for which a clear rationale exists to relate it to the assumptions and limitations of Table IV.

Note that assuming each pattern is 100% exact, counting their matches will generate a lower-bound on the number of code instances which challenge static analysis tools. As a tight lower-bound more accurately answers SQ3 than a loose upper-bound would, we will not sacrifice precision for recall by generalizing patterns. Some patterns have non-empty intersections, i.e., two patterns may match on the same piece of code. This effect must be considered when interpreting the results below, next to that they are not all 100% exact.

Because the main threat to validity of this research method is the precision of the patterns, we manually estimated their precision by reading random samples of matched code in the corpus. For each pattern which is not exact by definition, we report the precision after sampling 10 instances and record the intent of the code examples as we interpreted it to confirm or deny the rationales of Table VI.

The patterns performed well; at least 8 out of the 10 sampled methods did challenge the limitation or assumption. In the sampled methods we observed that most of the challenging cases involve highly dynamic reflection, where the code uses complex data-dependent predicates to decide which methods to invoke or fields to modify. These predicates operated both on strings and meta objects. We also observed that exceptions were often ignored to continue with a next possible candidate.

B. Results for Corpus Impact Analysis

The Impact column of Table VII answers SQ3, detailing for each pattern its impact in the corpus in terms of projects covered by at least a single match. Note that between the patterns the percentages are not comparable due to possible overlap. Each percentage implies a minimal amount of problematic code instances for the related assumption or limitation, so we find a lower-bound on the impact of a static analysis tool which would be able to resolve these hard cases.

Here we interpret the reported impact percentage for each limitation qualitatively: (a) the impact of CorrectCasts seems low, so we do not find evidence in this corpus that this is a bad assumption; (b) we can conclude that detailed modeling of exceptions can not be avoided; (c) we see that the combination of collections and reflection (arrays, lists, and tables) is relevant for about half of the corpus, so this is an important area of attention; (d) we find complex computations around the filtering of meta objects in almost half of the projects, which signals new opportunities for soundy assumptions for computing with meta objects; finally, (e) a significant part of the corpus is tainted directly by the use of dynamic proxies, for which no clear solution seems to be on the horizon.

SQ3: Real Java code frequently challenges limitations of the existing static analysis tools, in particular, in relation to modeling of exceptions, collections, filtering of meta objects and dynamic proxies. The impact of CorrectCasts seems low.

Table VII
IMPACT OF LIMITATION PATTERNS (TABLE VI) IN THE CORPUS.

Pattern	Impact	Precision	Code intent
CorrectCasts	4%	8/10	Supplying a fallback or looping through candidates and swallowing the exception
Ignoring-Exceptions1	23%	10/10	Falling back to a less specific Meta Object, or switching to a different `ClassLoader`
Ignoring-Exceptions2	38%	9/10	Iterating through candidates and either breaking when one does not throw an exception, or continuing to the next candidates
Inaccurate-Indexed-Collections	55%	exact	Iterating through a signature of an meta object
InaccurateSets-AndMaps	38%	exact	Meta objects as function pointers in a table, mapping to objects, caching around Reflection API
NoMultiple-MetaObjects	54%	exact	Looking through candidates, performing mass updates of fields, checking signatures
Ignoring-Environment	2%	10/10	Only 9 instances found, they were all dependency injection
Undecidable-Filtering	48%	8/10	Trying different names of meta objects, filtering method and fields based on signature
NoProxy	21%	exact	Wrapping objects for caching or transactions, automatically converting between comparable interfaces

The summary **answer to the Main Research Question** is that apart from CorrectCasts, the limitations and assumptions of static analysis tools for which we have an AST pattern are challenged in significant numbers in this corpus.

VI. DISCUSSION

A. Threats to validity

A different categorization of "dynamic language features" in Section II might influence the answers to our research questions. To mitigate issues with the categorization we explicitly included a grammar fully covering the reflection API.

The SLR in Section III was conducted in 2015. To the best of our knowledge all material appeared since has been included in Section III. The reading and annotating of the literature itself was a human task for which we implemented mitigating cross checks and validation steps.

Although the corpus in Section IV has been constructed using state-of-the-art methods for maximum variation of meta data, the choice of meta data variables and the universe the projects are sampled from can be discussed. To the best of our knowledge there exists no better means for sampling an unbiased and representative corpus of open-source projects.

In Section V, we used AST patterns to assess the occurrence of challenging code. To mitigate the arbitrariness of the patterns, we maintained a direct trace between the patterns and literature study in Section III in Table IV. However, any undocumented assumptions or implicit limitations have naturally not been mapped. The patterns themselves could be inaccurate, which was discussed and mitigated in Section V.

The answer to the main question, claiming a high impact of known limitations of static analysis tools, must be interpreted in context of the aforementioned threats to validity.

B. The Dual Question of SQ3

The question of how well static analysis tools actually do on code which uses reflection, rather than their limitations is relevant. The review in Section III and the corpus in Section IV provide a starting point for answering it. However, a set of full comparative studies would be necessary, grouped by the **goal** of comparable analyses, by running the actual tools (where available) on the corpus. The respective coverage of the corpus for selecting the first 50, 100 or 200 projects is 56%, 72% and 88%. The first projects in the corpus are the most representative, so initial studies could be performed on one of these prefixes of the corpus. The configuration and execution of each tool for each project in the corpus, and the interpretation of detailed results per analysis group in this proposed study is at the scale of a community effort.

C. Related work

Next to the focused literature review of Section III we position this paper in a wider field of empirical analysis of source code. Reflection and related forms of dynamic behavior are supported by many programming languages. Not surprisingly, reflection usage has been studied, e.g. for such languages as Smalltalk [64], JavaScript [65], [66], PHP [62], [67] and Python [68], [69]. Despite the differences between programming languages studied as well as the methodologies used by the authors, all those papers agree with each other and with our observations made in Section IV: reflection mechanisms are used frequently, and they often cannot be completely resolved statically.

Even if the current observations are in line with previous work, they are unexpected. The current study is on the statically typed language Java rather than the aforementioned dynamically typed languages; for Java the use of reflection is expected to be the exception rather than commonplace. The Java language is designed to provide both clear feedback to the programmer and a built-in notion of code security, based on its static semantics. We find it surprising that reflection - the back door to dynamic language features - is used so often and in such a way that it does undermine these design goals. Selecting Java as a platform for robust and safe software engineering provides fewer guarantees than perhaps thought.

A related topic is language feature adoption. Parnin *et al*. have studied adoption of Java generics [70], Pinto *et al*. studied concurrent programming constructs [71], and Dyer *et al*. studied features prior to their official release [72]. Similar studies have also been conducted, e.g., for C# [73] and PHP [60].

Since we have conducted our SLR in October 2015 additional papers have been published on static analysis of Java programs using reflection, witnessing the continuing attention to this topic from the research community. Harvester [74] combines static and dynamic analyses to combat malware obfuscation. Resolution of reflective calls is done by the dynamic analysis.

HornDroid [75] implements a simple string analysis and, similarly to DroidSafe [35], replaces reflective calls with the direct ones whenever the string analysis renders it possible. DroidRA [76] models the use of reflection with COAL [77] and reduces the resolution of reflective calls to a composite constant propagation problem.

Beyond related work for Java, without going into details, all research in and applications of static analysis techniques to dynamically typed programming languages is relevant, e.g., [78], [79]. Our empirical observations (Section V) suggest that application of the existing soundy techniques for analyzing dynamic languages to Java could have an impact.

D. Implications for Java Software Engineers

The data shows that reflection is not only used often, but it is also used in a way challenging to static analysis. If robustness is of high priority, then the following tactics are expected to have a positive effect: (a) do not factor out reusable reflective code in type-polymorphic methods, since the CorrectCasts assumption is highly useful, keeping casts to concrete types close to the use of dynamic language features will keep code analyzable; (b) avoid the use of dynamic proxies at any cost (c) use local variables or fields to store references to meta objects rather than collections; (d) avoid loops over bounded collections of meta objects; and (e) test for preconditions rather than to wait for exceptions such as `ClassCastException`.

Given the observations in Section V, applying these tactics should lower the impact of the assumptions and limitations of static analysis tools and hence will make Java code more robust. All tactics trade more lines of code for better analyzability.

E. Implications for Static Analysis Researchers

For all reported challenges for static analysis tools for which we have an AST pattern, save the CorrectCasts assumption, the evidence suggests investigating opportunities for more soundy assumptions in static analysis tools. It can also motivate Java language or API extensions which cover the current uses of the reflection API with safer counterparts. The literature survey suggests looking into combinations with dynamic analysis and user annotations. Note that the highly advanced analysis tools *already solve* a number of these challenges (such as exception handling), but further improvement to get similar accuracy for higher efficiency is warranted since these tools would run faster on a part of the corpus [10].

The negative impact of the CorrectCasts assumption seems low, so even more aggressive use of said assumption to reason back from a cast and infer more concrete details about possible semantics is warranted.

A novel soundy assumption on the semantics of dynamic proxies would have a significant impact, since currently all static analysis techniques ignore their existence completely (which is definitely unsound). For example, we observed that a useful soundy assumption might be that client code can remain "oblivious" to any proxy handlers that wrap arbitrary objects (that implement the same interface) to introduce ignorable aspects such as caching, offline serialization or transactional behavior.

We observed that exceptions are used as gotos, especially in the context of reflection. Hence, a special treatment of the code which catches these exceptions is warranted. Treating common idioms of such "error handling" should have a significant effect in the corpus, without having to use or introduce a general solution for exception handling per sé.

We see how relevant collections of meta objects (arrays, lists, and tables) are for analyzing the corpus. Since most collections of meta objects are bounded - they are acquired via bounded Reflection API methods - it should be possible to make more aggressive soundy assumptions around their usage. For instance, one can aggressively unroll iterators over meta object collections, or to soundily assume order independence.

Finally, considering the impact of UndecidableFiltering in the corpus in combination with MultiMetaObjects and the collection usage we see opportunities for the application of analysis techniques designed for dynamic languages (e.g., Javascript). Such dynamic Java code is akin to Javascript or PHP code. For example a form of determinancy analysis [79], [80] might be ported for the Java reflection case.

VII. Conclusions

Contemporary Java static analysis tools use pragmatic soundy techniques for dealing with the fundamental challenges around analyzing the Reflection API. Earlier work identified the need for empirical studies, relating these techniques to the way programmers actually use the Reflection API in real code.

With this paper we contributed (a) a comprehensive survey of the literature on the features and limitations of static analysis tools targeting reflective Java projects, (b) a representative corpus of 461 open-source Java projects, (c) an overview of the usage of the Reflection API by real Java code and (d) an AST-based analysis of how often the assumptions and limitations of the surveyed static analyses are challenged by real Java code.

The highlights among the empirical observations are that of all projects, in 78% dynamic language features are used. Moreover, 21% use dynamic proxies, 38% use exceptions for non-exceptional flow around reflection, 48% filter meta objects dynamically, and 55% store meta objects in generic collections. All those features are known to be problematic for static analysis tools. We could identify violations of the correct casts assumption in only 4% of the projects.

We conclude that (a) Java software engineers could make their code more analyzable by avoiding challenging code idioms around reflection, (b) introducing new soundy assumptions for novel static analysis techniques around the Reflection API is bound to have a significant impact in real Java code.

Acknowledgments

We thank Jeroen van den Bos, Mark Hills, and Paul Klint for helpful feedback on drafts of this paper and Anders Møller for his feedback on the topic of our research.

References

[1] B. Livshits, J. Whaley, and M. S. Lam, "Reflection analysis for Java," in *APLAS*, ser. LNCS, K. Yi, Ed., vol. 3780. Springer, 2005, pp. 139–160.

[2] B. Livshits *et al*., "In defense of soundiness: a manifesto." *Communications of ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[3] D. Landman, "A corpus of java projects representing the 2012 ohloh universe," https://doi.org/10.5281/zenodo.162926, Mar 2016.

[4] M. Bravenboer and Y. Smaragdakis, "Strictly Declarative Specification of Sophisticated Points-to Analyses," in *OOPSLA*. ACM, 2009, pp. 243–262.

[5] O. Lhoták and L. J. Hendren, "Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, no. 1, 2008.

[6] M. A. F. Gabaldón, "Logic-based techniques for program analysis and specification synthesis," Ph.D. dissertation, Universitat Politècnica de València, September 2013.

[7] Y. Li, T. Tan, Y. Sui, and J. Xue, "Self-inferencing reflection resolution for java," in *ECOOP*, ser. LNCS, R. Jones, Ed., vol. 8586. Springer, 2014, pp. 27–53.

[8] Y. Li, T. Tan, and J. Xue, "Effective soundness-guided reflection analysis," in *SAS*, ser. LNCS, S. Blazy and T. Jensen, Eds., vol. 9291. Springer, 2015, pp. 162–180.

[9] Y. Smaragdakis and G. Balatsouras, "Pointer analysis," *Foundations and Trends in Programming Languages*, vol. 2, no. 1, pp. 1–69, 2015.

[10] Y. Smaragdakis, G. Kastrinis, G. Balatsouras, and M. Bravenboer, "More Sound Static Handling of Java Reflection," in *APLAS*, ser. LNCS. Springer, 2015, pp. 485–503.

[11] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *SAS*, ser. LNCS, R. Cousot, Ed., vol. 2694. Springer, 2003, pp. 1–18.

[12] J. Sawin and A. Rountev, "Improved static resolution of dynamic class loading in java," in *SCAM*. IEEE, 2007, pp. 143–154.

[13] ——, "Improving static resolution of dynamic class loading in java using dynamically gathered environment information," *Autom. Softw. Eng.*, vol. 16, no. 2, pp. 357–381, 2009.

[14] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *ECOOP*, ser. LNCS, G. Castagna, Ed., vol. 7920. Springer, 2013, pp. 378–400.

[15] P. Centonze, R. J. Flynn, and M. Pistoia, "Combining static and dynamic analysis for automatic identification of precise access-control policies," in *ACSAC*. IEEE, 2007, pp. 292–303.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *CCS*, Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 627–638.

[17] J. Kim, Y. Yoon, K. Yi, and J. Shin, "ScanDal: Static analyzer for detecting privacy leaks in android applications," in *MoST*, H. Chen, L. Koved, and D. S. Wallach, Eds. IEEE, May 2012.

[18] J. Han, Q. Yan, D. Gao, J. Zhou, and R. H. Deng, "Comparing mobile privacy protection through cross-platform applications," in *NDSS*, P. Ning, Ed. The Internet Society, 2013.

[19] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen, "Formalisation and analysis of dalvik bytecode," *Sci. Comput. Program.*, vol. 92, pp. 25–55, 2014.

[20] T. Ravitch, E. R. Creswick, A. Tomb, A. Foltzer, T. Elliott, and L. Casburn, "Multi-app security analysis with FUSE: statically detecting android app collusion," in *PPREW@ACSAC*, M. D. Preda and J. T. McDonald, Eds. ACM, 2014, pp. 4:1–4:10.

[21] IBM, "T.J. Watson Libraries for Analysis (WALA)," http://wala.sourceforge.net/wiki/index.php/Main_Page, 2015, retrieved on 2015/12/10.

[22] P. Barros *et al*., "Static analysis of implicit control flow: Resolving java reflection and android intents," in *ASE*. ACM, 2015, pp. 669–679, to appear.

[23] M. D. Ernst *et al*., "Collaborative verification of information flow for a high-assurance app store," in *SIGSAC*, G. Ahn, M. Yung, and N. Li, Eds. ACM, 2014, pp. 1092–1104.

[24] Y. Cao *et al*., "EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework," in *NDSS*, E. Kirda, Ed. The Internet Society, 2015.

[25] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *EASE*. ACM, 2014, pp. 38:1–38:10.

[26] J. Webster and R. T. Watson, "Analyzing the past to prepare for the future: Writing a literature review," *MIS Quarterly*, vol. 26, no. 2, pp. xiii–xxiii, Jun. 2002.

[27] B. Kitchenham and S. Charters, "Guidelines for performing Systematic Literature Reviews in Software Engineering," Keele University and Durham University Joint Report, Tech. Rep. EBSE 2007-001, 2007.

[28] B. Kitchenham, E. Mendes, and G. H. Travassos, "A systematic review of cross- vs. within- company cost estimation studies," in *EASE*. British Computer Society, 2006, pp. 81–90.

[29] A. Fink, *Conducting Research Literature Reviews: From the Internet to Paper*. SAGE Publications, 2010.

[30] R. Vallée-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Conference of the Centre for Advanced Studies on Collaborative Research*, S. A. MacKay and J. H. Johnson, Eds. IBM, 1999, p. 13.

[31] O. Lhoták and L. Hendren, "Scaling java points-to analysis using SPARK," in *CC*. Springer, 2003, pp. 153–169.

[32] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders," in *ICSE*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 241–250.

[33] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in *FASE*, ser. LNCS, vol. 8411. Springer, 2014, pp. 140–154.

[34] W. Yang, X. Xiao, B. Andow, S. Li, T. Xie, and W. Enck, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *ICSE*. IEEE, 2015, pp. 303–313.

[35] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard, "Information Flow Analysis of Android Applications in DroidSafe," in *NDSS*, E. Kirda, Ed. The Internet Society, 2015.

[36] K. Ali and O. Lhoták, "Application-only call graph construction," in *ECOOP*, ser. LNCS, J. Noble, Ed., vol. 7313. Springer, 2012, pp. 688–712.

[37] M. Alpuente, M. A. Feliú, C. Joubert, and A. Villanueva, "Defining datalog in rewriting logic," in *LOPSTR*, ser. LNCS, D. D. Schreye, Ed., vol. 6037. Springer, 2009, pp. 188–204.

[38] ——, "Datalog-based program analysis with BES and RWL," in *Datalog Reloaded - First International Workshop*, ser. LNCS, O. de Moor, G. Gottlob, T. Furche, and A. J. Sellers, Eds., vol. 6702. Springer, 2010, pp. 1–20.

[39] J. Sawin and A. Rountev, "Assumption hierarchy for a CHA call graph construction algorithm," in *SCAM*. IEEE, 2011, pp. 35–44.

[40] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," in *PLDI*, M. Hind and A. Diwan, Eds. ACM, 2009, pp. 87–97.

[41] K. Ali, M. Rapoport, O. Lhoták, J. Dolby, and F. Tip, "Constructing call graphs of scala programs," in *ECOOP*, ser. LNCS, R. Jones, Ed., vol. 8586. Springer, 2014, pp. 54–79.

[42] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, "Lessons from applying the systematic literature review process within the software engineering domain," *Journal of Systems and Software*, vol. 80, no. 4, pp. 571–583, 2007.

[43] M. Hirzel, D. von Dincklage, A. Diwan, and M. Hind, "Fast online pointer analysis," *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 2, 2007.

[44] B. Dufour, B. G. Ryder, and G. Sevitsky, "Blended analysis for performance understanding of framework-based applications," in *ISSTA*, D. S. Rosenblum and S. G. Elbaum, Eds. ACM, 2007, pp. 118–128.

[45] A. Thies and E. Bodden, "Refaflex: safer refactorings for reflective java programs," in *ISSTA*, M. P. E. Heimdahl and Z. Su, Eds. ACM, 2012, pp. 1–11.

[46] M. Islam and C. Csallner, "Generating test cases for programs that are coded against interfaces and annotations," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 3, p. 21, 2014.

[47] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci, "Stadyna: Addressing the problem of dynamic code updates in the security analysis of android applications," in *CODASPY*, J. Park and A. C. Squicciarini, Eds. ACM, 2015, pp. 37–48.

[48] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for java," in *OOPSLA*. ACM, 1999, pp. 292–305.

[49] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical extraction techniques for java," *ACM Trans. Program. Lang. Syst.*, vol. 24, no. 6, pp. 625–666, 2002.

[50] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, "F4F: taint analysis of framework-based web applications," in *OOPSLA*, C. V. Lopes and K. Fisher, Eds. ACM, 2011, pp. 1053–1068.

[51] S. Blackshear, A. Gendreau, and B. E. Chang, "Droidel: a general approach to android framework modeling," in *SOAP@PLDI*, A. Møller and M. Naik, Eds. ACM, 2015, pp. 19–25.

[52] S. M. Blackburn *et al*., "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA*. ACM Press, Oct. 2006, pp. 169–190.

[53] S. Hunston, *Corpora in Applied Linguistics*, ser. Cambridge applied linguistics series. Cambridge University Press, 2002.

[54] E. Tempero *et al*., "The Qualitas corpus: A curated collection of Java code for empirical studies," in *APSEC*, Nov 2010, pp. 336–345.

[55] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *ESEC/FSE*. ACM, 2013, pp. 466–476.

[56] J. Kurš, M. Lungu, and O. Nierstrasz, "Bounded seas: Island parsing without shipwrecks," in *SLE*, ser. LNCS, B. Combemale, D. Pearce, O. Barais, and J. Vinju, Eds., vol. 8706. Springer, 2014, pp. 62–81.

[57] L. Moonen, "Generating robust parsers using island grammars," in *WCRE*. IEEE, 2001.

[58] B. Basten *et al*., "M^3: a General Model for Code Analytics in Rascal," in *SWAN*, O. Baysal and L. Guerrouj, Eds. IEEE, 2015, pp. 25–28.

[59] P. Klint, T. van der Storm, and J. J. Vinju, "Rascal: A domain specific language for source code analysis and manipulation," in *SCAM*. IEEE, 2009, pp. 168–177.

[60] M. Hills, "Evolution of dynamic feature usage in PHP," in *SANER*, Y. Guéhéneuc, B. Adams, and A. Serebrenik, Eds. IEEE, 2015, pp. 525–529.

[61] D. Landman, A. Serebrenik, E. Bouwers, and J. J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions," *Journal of Software: Evolution and Process*, 2015.

[62] M. Hills, P. Klint, and J. J. Vinju, "An empirical study of PHP feature usage: a static analysis perspective," in *ISSTA*, M. Pezzè and M. Harman, Eds. ACM, 2013, pp. 325–335.

[63] D. Landman, "cwi-swat/static-analysis-reflection," https://doi.org/10.5281/zenodo.163326, Oct. 2016.

[64] O. Callaú, R. Robbes, É. Tanter, and D. Röthlisberger, "How (and why) developers use the dynamic features of programming languages: the case of smalltalk," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1156–1194, 2013.

[65] G. Richards, S. Lebresne, B. Burg, and J. Vitek, "An analysis of the dynamic behavior of javascript programs," in *PLDI*. ACM, 2010, pp. 1–12.

[66] G. Richards, C. Hammer, B. Burg, and J. Vitek, "The eval that men do: A large-scale study of the use of eval in javascript applications," in *ECOOP*. Springer, 2011, pp. 52–78.

[67] M. Hills, "Variable feature usage patterns in php," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, Nov 2015, pp. 563–573.

[68] A. Holkner and J. Harland, "Evaluating the dynamic behaviour of python applications," in *ACSC*. Australian Computer Society, Inc., 2009, pp. 19–28.

[69] B. Åkerblom, J. Stendahl, M. Tumlin, and T. Wrigstad, "Tracing dynamic features in python programs," in *MSR*. ACM, 2014, pp. 292–295.

[70] C. Parnin, C. Bird, and E. Murphy-Hill, "Adoption and use of java generics," *Empirical Software Engineering*, vol. 18, no. 6, pp. 1047–1089, 2013.

[71] G. Pinto, W. Torres, B. Fernandes, F. C. Filho, and R. S. M. de Barros, "A large-scale study on the usage of java's concurrent programming constructs," *Journal of Systems and Software*, vol. 106, pp. 59–81, 2015.

[72] R. Dyer, H. Rajan, H. A. Nguyen, and T. N. Nguyen, "Mining billions of AST nodes to study actual and potential usage of java language features," in *ICSE*, P. Jalote, L. C. Briand, and A. van der Hoek, Eds. ACM, 2014, pp. 779–790.

[73] P. Capek, E. Kral, and R. Senkerik, "Towards an empirical analysis of .net framework and c# language features' adoption," in *CSCI*, Dec 2015, pp. 865–866.

[74] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting runtime values in android applications that feature anti-analysis techniques," in *NDSS*. The Internet Society, 2016.

[75] S. Calzavara, I. Grishchenko, and M. Maffei, "Horndroid: Practical and sound static analysis of android applications by SMT solving," in *EuroS&P*. IEEE, 2016, pp. 47–62.

[76] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Droidra: taming reflection to support whole-program analysis of android apps," in *ISSTA*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 318–329.

[77] D. Octeau, D. Luchaup, M. Dering, S. Jha, and P. McDaniel, "Composite constant propagation: Application to android inter-component communication analysis," in *ICSE*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE, 2015, pp. 77–88.

[78] M. Sridharan, J. Dolby, S. Chandra, M. Schäfer, and F. Tip, "Correlation tracking for points-to analysis of javascript," in *ECOOP*, ser. LNCS, J. Noble, Ed., vol. 7313. Springer, 2012, pp. 435–458.

[79] E. Andreasen and A. Møller, "Determinacy in static analysis for jquery," in *ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14. ACM, 2014, pp. 17–31.

[80] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip, "Dynamic determinacy analysis," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13. New York, NY, USA: ACM, 2013, pp. 165–174.