Assessing Reflection Usage with Mutation Testing Augmented Analysis

Iona Thomas, Stéphane Ducasse, Guillermo Polito, Pablo Tesone

Assessing Reflection Usage with Mutation Testing Augmented Analysis

Iona Thomas[1 [0000−0001−8490−3802]], Stéphane Ducasse[1 [0000−0001−6070−6599]], Guillermo Polito[1 [0000−0003−0813−8584]], and Pablo Tesone[1 [0000−0002−5615−6691]]

Univ Lille, Inria, CNRS, Centrale Lille, UMR 9189 - CRIStAL, Lille, 59650, France
{FirstName,LastName}@inria.fr

Abstract. Reflection is a powerful tool that allows a program to manipulate itself during its execution. However, developers may use it to circumvent data encapsulation and method visibility modifiers. Thus, it is important to assess how much an application relies on reflection.
Nonetheless, reflection is mostly incompatible with static analysis as it relies on runtime information (*e.g.,* to determine the attribute to be accessed or the method to evaluate). These problems worsen with dynamically-typed languages, where reflective operations are polymorphic with non-reflective operations, *e.g.,* in Pharo, array access is polymorphic with context variable modifications.
In this paper, we present RAPIM, an approach to study the uses of reflective APIs: it uses mutation analysis with a new mutation operator for dealing with core reflective methods. We analyze a serialization library from a developer perspective, showing the information it reveals. We evaluate our approach on a selection of five projects by comparing its performance against static analysis. We show that out of five projects, RAPIM disambiguates more potentially reflective call-sites than the static analysis. When the code coverage is good, the percentage of disambiguation is three times higher. Finally, we question the relevance of polymorphism between non-reflective and reflective APIs. Out of five projects, only one uses it, for only 1.4% of potentially reflective call-sites. We argue that reflective APIs could be renamed to avoid ambiguities.

1 Introduction

Reflective operations allow a program to manipulate itself (and its programming language) during its execution. This includes [17,22]:

- *Introspection*, the ability to examine its own structure and state,
- *Self-modification*, the ability to change itself,
- *Intercession*, the ability to alter the semantics of its programming language.

They are a powerful tool, allowing us to build debugging tools, refactorings, and new language features [10,24,28].

The need for reflection usage analysis. Although powerful, reflective features are usable as security exploits. For example, they allow malicious users to violate encapsulation and execute methods that were not intended to be executed [23,19,11]. This means that *developers must assess how much they rely on reflection.* It is important to understand if a reflective functionality is central to an application or if it is only confined to peripheral parts (See Section 2.1).

Challenges of reflection analysis. These problems become exacerbated in dynamically-typed languages. On the one hand, reflective features defeat static analysis [15,5] because the actual attributes/methods that are being used are decided at runtime. On the other hand, reflective APIs are often designed as normal methods and are often polymorphic with non-reflective operations. For example, in Javascript reflectively accessing an attribute (`object['attribute']`) is syntactically equivalent to array/dictionary accesses (`dictionary[index]`).

> *Reflection has always been a thorn in the side of Java static analysis tools. Without a full treatment of reflection, static analysis tools are both incomplete because some parts of the program may not be included in the application call graph, and unsound because the static analysis does not take into account reflective features of Java that allow writes to object fields and method invocations. However, accurately analyzing reflection has always been difficult, leading to most static analysis tools treating reflection in an unsound manner or just ignoring it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection [15].*

While the quote above is about Java, this tension is exacerbated in the case of deeply reflective languages such as Smalltalk descendants. Pharo, for example, as a descendant of Smalltalk is the essence of a reflective language with advanced reflective operations such as bulk pointer swapping [20], on-demand stack reification, and first-class resumable exceptions. In addition, in Smalltalk and many of its derivatives, reflective facilities are mixed with the non-reflective API of objects and classes [28,24,4]. They are a key part of the kernel of the language and libraries (See Section 2.2).

Mutation-based assessing reflection uses. In this paper, we propose to augment reflection security analysis with mutation analysis. We design reflection-specific mutations to obtain dynamic runtime information (See Section 3). We then use this information to assess the dependencies on reflective features and the degraded modes of an application (See Section 4).

Contributions.

- First, an approach based on mutation testing to assess the dependencies of an application on certain reflective APIs.
- Second, RAPIM an implementation strategy to contextualize the analysis *i.e.,* handling the fact that reflective features are often a core part of a language (Python, Pharo...) and cannot be simply pulled off.

- A way to report results in a structured manner.
- An evaluation of the approach on a selection of projects. We show that in four out of five projects RAPIM disambiguates more *potentially reflective call-sites* than static analysis. When the coverage is good, the disambiguation rate was up to three times higher or more.
- An evaluation of the relevance of polymorphism between reflective and non-reflective APIs. We show that the polymorphism is very rarely used, with only one project leveraging it for 1.4% of its *potentially reflective call-sites*.

2 Background: Reflection Analysis

2.1 Why Assessing Reflection Dependencies

Developers need to monitor how much they rely on reflection without having to analyze the code manually. We want to provide an automated way to assess how much an application or library relies on reflective operations, and which ones. This is important because reflective operations

- allow one to bypass encapsulation and break invariants,
- can make the system unstable, and
- might introduce a performance cost.

When performing an analysis developers should not get drowned in a sea of information. Information should be presented with different levels of granularity and potentially convey more semantical information. Here are more precise questions to characterize a reflective feature usage.

- *General use.* How much does a project rely on reflection in general? The first general objective is to understand if an application uses or not reflective features.
- *Faceted analysis.* How much does it rely on specific reflective features? There are different families of reflective features: simple introspection, encapsulation violation, memory scanning, and more [28]. Each of such families has different consequences on security issues. This is why this is important to propose a faceted analysis.
- *Spatial distribution.* Which application parts are impacted by a given reflective use? During the assessment developers need to understand the spread of a reflective use.
- *Degraded modes.* How much of the application still works without a given reflective use?

Note this article is not about validating if our solution addresses the previous questions since it would involve a large user study. The present paper is about how can we perform the analysis on top of which such questions could be answered. However, Section 4 is an example providing a glimpse of what such an analysis could look like.

2.2 The Challenges of Reflection Analysis

Dynamic decisions are a blind spot of static analysis. Reflective features defeat static analysis because the actual attributes/methods that are being used are decided and even crafted at runtime [15,23,12,13]. It is thus impossible to precisely know using static analyses what methods will be called by a reflective method invocation, or what fields will be read/written using a reflective field access.

Polymorphism between reflective and non-reflective methods. Moreover, such reflective dependency analysis is even more difficult in dynamically-typed languages where reflective APIs are often designed as normal methods that are polymorphic with non-reflective operations. As already mentioned, in Javascript and Python reflectively accessing an attribute (`object['attribute']`) is syntactically equivalent to array/dictionary accesses (`array[index]`/`dictionary[key]`). In Pharo, 44% of the reflective method's selectors have also non-reflective implementors.

This makes reflective operations look like any other message. Reflective features are handy but they should not be used by accident, and should not be mistaken for regular non-reflective messages. While solutions such as mirrors [6] offer a way to separate the base level from the meta-level, it requires the full redesign of some core functionalities of a language. This does not solve the problems of developers of existing languages.

3 RAPIM: Reflective API Mutation Analysis

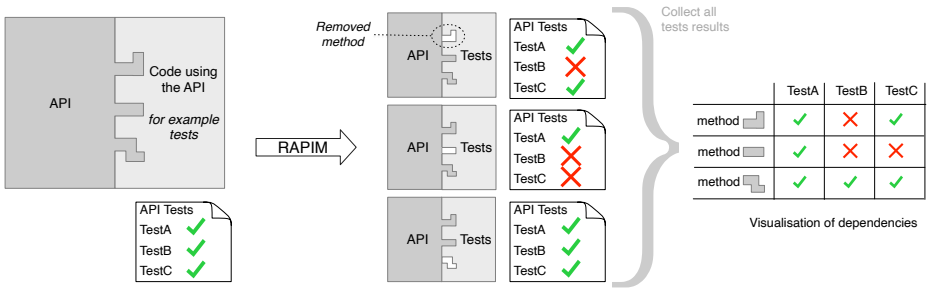

Fig. 1. RAPIM: Using mutation analysis to assess reflection usage. For each reflective method removed, all tests are run and results are collected.

Modern development methodologies such as Agile Programming and Test-Driven Design [3,2,1,18] advocate the systematic use of tests. Tests are an automated way to verify a sort of executable specification. While a suite of tests usually aims at testing various inputs, especially boundary values, and to cover as much code and paths as possible for the tested application, they are rarely

randomly written but centered around the validation of application features. We propose to leverage these tests to assess dependencies to reflective operations.

Our approach, RAPIM, extends mutation analysis to assess reflective features use as illustrated by Figure 1. Mutation testing is a technique that allows one to evaluate the coverage of a test suite by modifying the code and checking that at least one test break. Indeed if a test breaks, the test suite detects the modification. Here we remove reflective methods and we assess the impact by running the tests.

3.1 Terminology

This section introduces some terminology required to understand the rest of this article, supported by Figure 2.

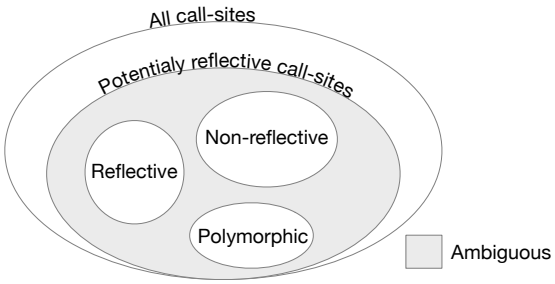

Fig. 2. Call-sites terminology

Potentially reflective call-site. A call-site is potentially reflective if it could directly call a reflective method (*i.e.,* not in a transitive manner). We consider that a call-site is potentially reflective if there is at least one reflective method implementing the call-site signature. For example, `at:put:` has at least one reflective implementor, therefore all `at:put:` call-sites are potentially reflective.

Reflective call-site. A *potentially reflective call-site* that only calls reflective methods.

Non-Reflective call-site. A *potentially reflective call-site* that only calls non-reflective methods.

Polymorphic call-site. A *potentially reflective call-site* that calls both reflective and non-reflective methods.

Ambiguous call-sites. A set of *potentially reflective call-sites* that we cannot identify as reflective, non-reflective or polymorphic.

3.2 Reflective Mutation Analysis Approach

Mutation testing works by applying a non-semantic-preserving mutation and checking if at least a test breaks after such a change. We extended Mutalk, a

mutation testing framework for the Pharo programming language, and designed a reflection-specific mutation operator to assess the use of reflection, explained in Section 3.3. This operator works on pairs of call-site and reflective methods to identify which one is called, if any. We use the list of reflective methods defined by Thomas et al. [28]. We use this list to consider that a call site is a *potentially reflective call-site*.

Since we want to provide a full assessment of the reflective API use, we configured the mutation framework to run all the tests covering a mutation instead of stopping at the first one that failed. This allows us to report the percentage of working tests after a mutation. Our operator will then fail tests using reflective features and this allows us to identify per test:

- **Surviving Mutant.** A reflective feature that is not used by the test.
- **Killed Mutant.** A reflective feature is used by the failed test.

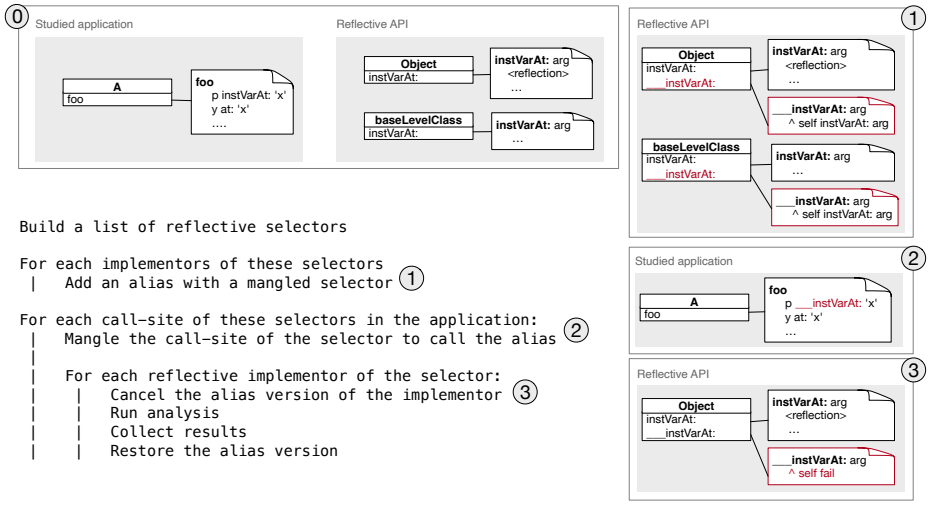

Fig. 3. Code transformation for *Reflective method canceling*.

3.3 Reflective Method Cancelling Operator

For the mutation analysis, we want tests to fail when a specific reflective method is called from a specific *potentially reflective call-site*. To cancel a reflective method (so that calling it makes the current test fail), it is not sufficient to rewrite its call-site to invoke a different (potentially failing) method. Canceling a call-site in such a way will indeed fail when calling reflective methods, but will also fail when calling non-reflective methods. Remember that in Pharo, for example, the message `at:put:` is both implemented by collections (non-reflective

setter) and the execution stack (reflective). Call-sites calling at:put: on collection should not fail.

To analyze the usage of reflection, we designed a reflection-specific operator that cancels all pairs (reflective method, call-site), one by one, taking into account for each *potentially reflective call-site* all reflective methods that could be invoked by that call site. We call this the *reflective method canceling operator*, illustrated in Figure 3. For each pair, the operator:

1. rewrites the *potentially reflective call-site* to an alias of the original method (see step 2, Figure 3)
2. introduces a canceled alias for the method (see step 3, Figure 3)

Then all tests covering this *potentially reflective call-site* are run. If the call-site calls the canceled reflective method the test fails, if it calls any other implementation, the code run normally, with only an additional layer of indirection.

Our implementation further optimizes this approach by pre-installing the aliases for all implementors of reflective selectors instead of doing it for each call-site (see step 1, Figure 3).

4 RAPIM by Example: a Developer Perspective

In this section we illustrate RAPIM with a use case: Pharo's STON serialization framework. Through this use case, we show how it helps the developer to answer questions listed in 2.1. Table 1 shows a first glance at the distribution of *potentially reflective call-site* 13% of the call-sites are not covered by the tests. 26% of the call-sites are only calling reflective methods. 61% are only calling non-reflective methods.

Percentage of:	Non covered	Refl.	Non-Refl.	Polymorphic
potentially reflective call-sites	13%	26%	61%	0

Table 1. STON's *potentially reflective call-sites* classification with RAPIM

Figure 4 shows that out of the 13 selectors used by these call sites:

- 5 of them only have callsites calling non-reflective methods (#value:value: #at: #value #size #lookupClass)
- 5 of them only have callsites calling reflective methods (#instVarNamed: #cull:cull: #instVarNamed #isMeta #allSubclasses). Those have only one call-site each, which means that these uses of reflection are not too spread around in the application.
- #class has two out of eleven callsites that are not covered by tests. However, a static analysis reveals that #class has only one implementor. Therefore, all of those callsites are reflective too.

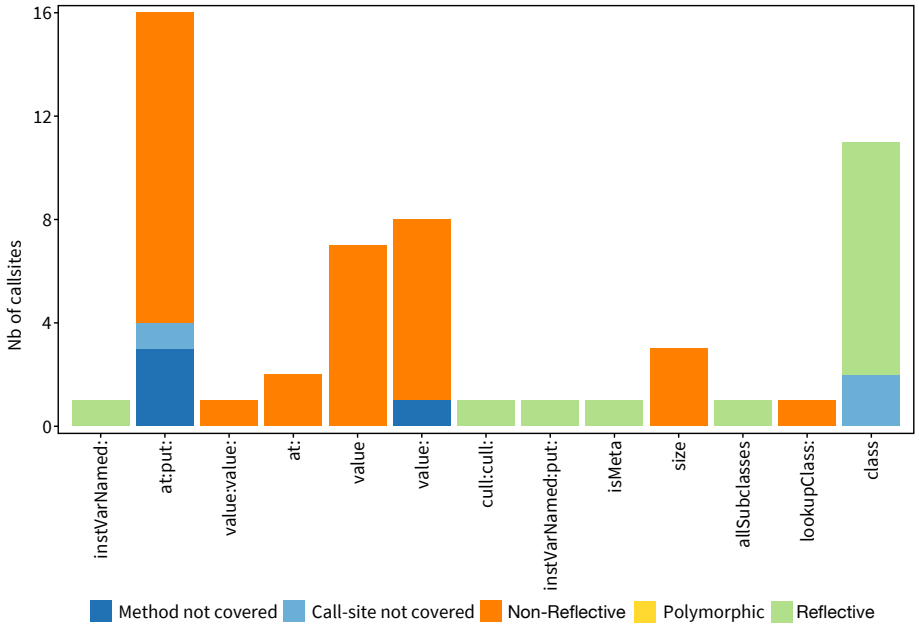

Fig. 4. Proportion of call sites types by selector for STON.

– All #at:put: and #value: covered call-sites are non-reflective, but some are not covered which means that we cannot conclude anything about them.

The matrix on Figure 5 gives fine-grained information: the percentage of broken tests in test classes if we actually remove each reflective method. STON relies on seven reflective methods. MySTONCStyleCommentsSkipStreamTest is the only test class whose tests do not use reflection. Assuming serialization is tested by MySTONWriterTest, it only relies on #instVarNamed: and #class which was expected. For deserialization, MySTONReaderTest requires more reflective methods, as it relies on six methods (See Figure 5 for the details).

Table 1, Figures 4 and 5 highlight different levels of detail that can lead to a better understanding of the dependency of a project on reflection. The matrix visualization provides also other levels, without grouping the tests by class for finer information, or by grouping reflective methods by categories of reflective methods for coarser ones (See Appendix B).

5 Evaluation of Reflective API Identification

In the previous section, we reported how our analysis supports the developers in assessing the dependencies of the system to reflective features. In this section, we answer the research questions that drove our experiments:

– RQ1. Mutation Comparison Does mutation analysis improve the detection of reflective API usages in comparison to static analysis?

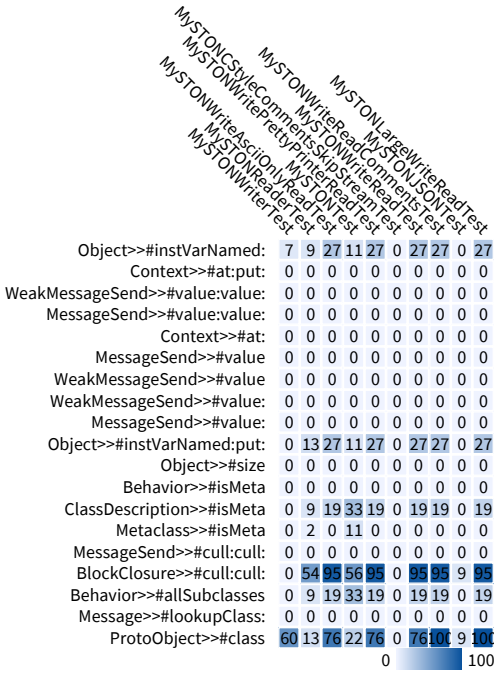

Fig. 5. Table of percentage of tests in a given class depending on a reflective method

– RQ2. Effective Polymorphism Is the polymorphism between non-reflective and reflective methods actually used by applications? Could we safely rename reflective methods to disambiguate *potentially reflective call-sites*?

5.1 Chosen Projects

Following is the list of the projects we studied, and the criteria for selection of each of them. Table 2 summarizes the results of applying RAPIM on these projects. (See Appendix A for links to repositories)

– *STON.* SmallTalk Object Notation. A textual object serializer/deserializer inspired by JSON. We expect reflection to be used for reading class information and instance variables for serialization and setting instance variables for deserialization.
– *Microdown.* A parser for a markup language derived from Markdown [8]. We do not expect many reflection uses in this application.
– *Refactoring.* The refactoring framework used in production by the Pharo development environment [26,25]. We studied both the *Refactoring-Core* and the *Refactoring-Transformations* packages. We expect a heavy use of reflection as it is manipulating methods and classes to perform code rewritings.
– *MuTalk.* A mutation testing framework, the same we use for this analysis. We worked on a copy of these packages to be able to execute our approach

on its framework. We expect the use of reflection to edit methods to install mutations.
– *Seaside.* A web application framework maintaining sessions using continuations [9]. We expect the use of reflection in the continuation management.

Project	version	#classes	#tests	#call-sites	Non-cov.	Refl.	Non-Refl.	Poly.
STON	bbe8f5f	14	310	54	13%	26%	61%	0
Microdown	72f4ac7	168	543	576	77%	3%	20%	0
Refactoring	Pharo12 build.1386	211	807	1022	39%	7,4%	52,2%	1,4%
MuTalk	e712ac5	150	303	238	48%	11%	41%	0
Seaside	56286ac	547	876	1314	74%	5%	21%	0

Table 2. Projects statistics and their *potentially reflective call-site* classification with RAPIM

5.2 Answering RQ1. Mutation Comparison

Does Mutation Analysis improve the detection of reflective API usages in comparison to static analysis?

Static analysis classifies *potentially reflective call-sites* according to the implementors of the method called. If all implementors are reflective, the call-site is identified as a reflective call-site. Otherwise, if there is at least one non-reflective implementor, the call-site is ambiguous.

Table 3 shows the number of *potentially reflective call-sites* that static analysis identifies as reflective or ambiguous, and whether they are covered by tests or not. As explained in Section 3.1, reflective call-sites have only reflective implementations, and ambiguous call-sites' implementors have a subset that is reflective and a subset that is non-reflective.

Project	Static analysis	Not Covered	Covered
STON	Statically Reflective	2 *(4%)*	11 *(20%)*
	Statically Ambiguous	5 *(9%)*	36 *(67%)*
Microdown	Statically Reflective	174 *(30%)*	16 *(3%)*
	Statically Ambiguous	272 *(47%)*	114 *(20%)*
Refactoring	Statically Reflective	108 *(10.5%)*	32 *(3%)*
	Statically Ambiguous	292 *(28.5%)*	590 *(58%)*
MuTalk	Statically Reflective	20 *(8%)*	15 *(6%)*
	Statically Ambiguous	95 *(40%)*	110 *(46%)*
Seaside	Statically Reflective	197 *(15%)*	52 *(4%)*
	Statically Ambiguous	779 *(59%)*	286 *(22%)*

24% disambiguated by static analysis

87% disambiguated by RAPIM

Table 3. Number of *potentially reflective call-sites* by projects, split by coverage and reflective ambiguity according to static analysis.

Static analysis can only identify reflective call-sites for which all implementors are reflective. On the other hand, RAPIM analysis scope only includes covered call-sites. Ambiguous and covered *potentially reflective call-sites* are analyzed by RAPIM to identify the reflective ones. Non-covered reflective call-sites are not detected by the dynamic analysis of RAPIM because this approach is based on code coverage. The ambiguous and not covered call-sites are the ones neither approach can disambiguate.

As shown in Table 3, all projects have ambiguous *potentially reflective call-site* that are covered by tests. Those call sites cannot be disambiguated by static analysis, but RAPIM allows one to classify them. For example in the STON project, 87% of the *potentially reflective call-sites* are disambiguated by RAPIM, and 24% of the *potentially reflective call-sites* are identified as reflective by the static analysis (See annotation on Table 3). Our analysis shows that 67% of the *potentially reflective call-sites* are ambiguous and covered. The 20% of statically reflective and covered call-sites are identified by both the static analysis and RAPIM. The 4% of not covered and statically reflective call sites are not taken into account by the dynamic analysis. Conversely, the 67% of ambiguous and covered call-sites are not disambiguated by the static analysis. The 9% remaining not covered and ambiguous call-sites are the ones neither approaches can disambiguate. The projects Refactoring and MuTalk have similar profiles.

The situation is slightly different in Microdown where the coverage of *potentially reflective call-sites* is lower (23%). There is only 3% of covered and reflective call-sites. This means that the overlap between static analysis and RAPIM is small. RAPIM disambiguates 20% of the *potentially reflective call-sites*. This is the lowest among the five studied projects. 47% of the call sites are never disambiguated. In addition, the static analysis disambiguates 33% of *potentially reflective call-sites* (which is the highest among our 5 projects) while RAPIM only 23%. This tilts the balance in favor of static analysis for this projects. The Seaside project exhibits the same profile, but the lower amount of statically reflective call-sites gives RAPIM the advantage.

Conclusion. On four out of five projects, RAPIM disambiguates more *potentially reflective call-sites* than the static analysis. When projects have good coverage (STON, MuTalk, Refactoring), our approach disambiguates **three times more** *potentially reflective call-sites* than the static analysis. For projects with low coverage, the disambiguation is on par with static analysis, but they present a high percentage of call sites that are still ambiguous.

5.3 Answering RQ2. Effective Polymorphism

Is the polymorphism between non-reflective and reflective methods used by applications? Could we safely rename reflective methods to disambiguate potentially reflective call-sites*?*

Table 2 shows that only one project uses polymorphism between reflective and non-reflective methods. This is expected because the refactoring engine has its meta-model of the code that mimics part of Pharo's reflective API [28]. Moreover, it mixes Pharo meta-objects with its own code model. Even in this case, Table 2 shows that polymorphic usage concerns only 1.4% of the *potentially reflective call-sites*.

Conclusion. Out of the five projects, only the refactoring engine leverages the polymorphism between reflective and non-reflective APIs. Even in this case, this is a very rare usage (1.4% of *potentially reflective call-sites*). This is **strong evidence** showing that reflective APIs could be re-designed to be mostly non-ambiguous in dynamically-typed languages.

6 Discussion

This section discusses the limitations of our approach and the threats to validity of our evaluation.

6.1 Limitations

Code Coverage. RAPIM cannot disambiguate *potentially reflective call-sites* that are not covered. We saw in Section 5 that we get better results with a higher coverage. This is a limitation inherited from mutation analysis in general. However, our evaluation shows that when tests are available, mutation analysis complements static analyses.

Other meta-object. RAPIM only identifies dependencies to reflective methods. It does not cover access to global variables or stack reifications with the pseudo-variable thisContext. However, many methods that could be called on thisContext are identified as reflective methods (*i.e.,* while accessing to the reified stack is not detected by RAPIM, the use of many methods on the stack reification is detected).

Indirect dependencies. If the studied application relies indirectly on reflection (*i.e.,* it uses a library that uses reflection), these dependencies will not be identified. RAPIM is designed to detect direct calls from the studied application to the reflective API scoped to a package.

Studying core packages. RAPIM cannot be used directly on the standard libraries of the Pharo programming language, as it could break the system. As in the case of Mutalk, this can be solved by creating copies of the studied packages and running RAPIM on this copy.

6.2 Threats to validity

Internal Validity.

Code Coverage. Studying real-life application is necessary to evaluate effective polymorphism, but our selection comes with a wide range of code coverage. Our analysis relies on the fact that tests cover both reflective and non-reflective cases. Having some projects with higher coverage mitigates the risk that reflective cases are overlooked in tests. The fact that we do not have a higher polymorphism percentage in those projects supports our conclusion.

Pharo exception handling. We left out of the analysis several tests (13) related to exception handling in the Seaside project. Running RAPIM on Seaside introduced a bug leading to infinite loops in these tests. The impact of removing thos is mitigated by the amount of other tests as removed tests only represent 1,5% of the total tests one this project.

Construct Validity.

Studying core packages. RAPIM cannot be used directly on the core libraries of the Pharo programming language, as it could break the system. To study such packages (*e.g.,* STON), we duplicate the package and run RAPIM on the copy. To ensure that results on the copy would be informative about the original package, we made sure that the duplicated version still runs well, and that all its tests are green.

External Validity.

Project selection for the evaluation. The evaluation of our approach relies on the results of RAPIM on five applications. To run this evaluation, we chose a set of various projects that we were expecting to use different amount and features of reflection. We specifically aimed for variety to mitigate the bias introduced by selecting only a few projects.

7 Related Work

This section presents related work in two different axes. First, the existing approaches to overcome the limitations of using only static analysis in dynamic languages. Then, the usage of mutation testing for program analysis and validation.

7.1 Overcoming the limits of static analyses

Ruf [27] proposes the use of partial evaluation to detect the meta-level operations, once these meta-level operations are detected they are replaced by equivalent code as they are not present at run-time. Braux and Noyé [7] improve the results of static analysis for Java programs by applying partial evaluation to reflection resolution to apply optimizations. Their paper describes extensions to a standard partial evaluator to offer reflection support. The idea is to *compile away* reflective calls in Java programs, turning them into regular operations on objects

and methods, given constraints on the concrete types of the object involved. The type constraints for performing specialization are manually provided.

Tip *et al.,* propose Jax [29]: an application extractor and compactor for Java. In their solution, Jax performs a static analysis of the program and builds a call graph of the application. It then removes unused methods and compacts the application. It is also affected by the limitations of static analysis. To handle the missing information the authors propose three alternatives: (1) it requires user intervention to handle the dynamic loading and execution of code, (2) it performs a conservative selection of possible methods for a given call site, and (3) assumes that all methods in external library interfaces are called.

Bodden *et al.,* [5] approaches the limitations of static analysis by integrating static analysis tools with runtime information. Their tool inserts runtime checks into the code. These checks warn the user in case the program is performing reflective calls that were not identified by the static analysis. More recently, Liu *et al.,* [14] improved the approach of Bodden *et al.,* by taking benefit from runtime information obtained from code coverage. They automatically generate test cases to improve the code coverage and the detection of reflective call targets. Similar to us, this work uses runtime information, and more specifically tests, to augment static analysis. However, their objective differs: while they intend to obtain information on reflective call targets, our main goal is to understand the usage of reflective operations and de-ambiguate reflective from non-reflective operations in dynamic languages.

7.2 Program analysis and validation via mutation testing

Mouelhi *et al.,* [21] propose using mutation testing to detect security issues. Loise *et al.,* [16] based on this idea propose a series of mutation operators to detect specific security issues. They propose 15 mutation operators that are applied to Java programs. Using this technique they detect security issues that are ignored by static analysis. Our solution is not designed specifically for detecting security issues, but it is possible to identify misuses of reflective calls that introduce them.

Wen *et al.,* [30] propose using mutation testing to detect misuses of library APIs. They represent API misuses as mutation operators applied in the code base. Then the mutant killing tests and their associated stack traces are collected to detect API misuses. We approach the identification of reflective usage in a similar way showing that a single mutation operator is enough for reflective usage analysis.

8 Conclusion

In this article we presented RAPIM, a mutation analysis approach based on a new mutation operator to understand dependencies to reflective APIs. This approach handles the fact that reflective features are often a core part of a language (Python, Pharo...) and cannot be simply pulled off. We then present

with several levels of detail the identified dependencies for STON, the object serialization library used in Pharo.

To evaluate our approach, we used RAPIM on five different projects relying on reflection. On four out of five projects, RAPIM disambiguates more *potentially reflective call-sites* than the static analysis. When projects have good coverage (STON, MuTalk, Refactoring), our approach disambiguates three times more *potentially reflective call-sites* than the static analysis. For projects with low coverage, the disambiguation is on par with static analysis, but they present a high percentage of call sites that are still ambiguous.

Out of the five projects, only the refactoring engine leverages the polymorphism between reflective and non-reflective APIs. Even in this case, this is a very rare usage (1.4% of *potentially reflective call-sites*). This supports the idea that reflective API could be renamed to avoid accidental polymorphism with non-reflective methods.

We believe this approach applies to domains other than reflection and could help to sort out critical dependencies and further security analysis, such as dependencies to file access APIs or user inputs.

Acknowledgements

We would like to thank the anonymous reviewers for their useful feedback. We are also grateful to the European Smalltalk User Group (http://www.esug.org) for their financial support.

References

1. Astels, D.: Test-Driven Development — A Practical Guide. Prentice Hall (2003)
2. Beck, K.: Manifesto for agile software development, http://agilemanifesto.org
3. Beck, K.: Extreme Programming Explained: Embrace Change. Addison Wesley (2000)
4. Black, A.P., Ducasse, S., Nierstrasz, O., Pollet, D., Cassou, D., Denker, M.: Pharo by Example. Square Bracket Associates, Kehrsatz, Switzerland (2009), http://books.pharo.org
5. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proceedings of the 33rd International Conference on Software Engineering. pp. 241–250. ICSE '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1985793.1985827
6. Bracha, G., Ungar, D.: Mirrors: design principles for meta-level facilities of object-oriented programming languages. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices. pp. 331–344. ACM Press, New York, NY, USA (2004), http://bracha.org/mirrors.pdf
7. Braux, M., Noyé, J.: Towards partially evaluating reflection in java. ACM SIGPLAN Notices **34**(11), 2–11 (1999)

8. Ducasse, S., Dargaud, L., Polito, G.: Microdown: a clean and extensible markup language to support pharo documentation. In: Proceedings of the 2020 International Workshop on Smalltalk Technologies (2020)
9. Ducasse, S., Renggli, L., Shaffer, C.D., Zaccone, R., Davies, M.: Dynamic Web Development with Seaside. Square Bracket Associates (2010), http://book.seaside.st/book
10. Forman, I.R., Forman, N.: Java Reflection in Action (In Action series). Manning Publications Co., USA (2004)
11. Hunt, G.C., Larus, J.R.: Singularity: rethinking the software stack. SIGOPS Oper. Syst. Rev. **41**(2), 37–49 (2007). https://doi.org/10.1145/1243418.1243424
12. Landman, D., Serebrenik, A., Vinju, J.J.: Challenges for static analysis of java reflection - literature review and empirical study. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). pp. 507–518 (2017). https://doi.org/10.1109/ICSE.2017.53
13. Li, S., Dietrich, J., Tahir, A., Fourtonis, G.: On the recall of static call graph construction in practice. In: ICSE (2020)
14. Liu, J., Li, Y., Tan, T., Xue, J.: Reflection analysis for java: Uncovering more reflective targets precisely. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE). pp. 12–23 (2017). https://doi.org/10.1109/ISSRE.2017.36
15. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for java. In: Proceedings of Asian Symposium on Programming Languages and Systems (2005)
16. Loise, T., Devroey, X., Perrouin, G., Papadakis, M., Heymans, P.: Towards security-aware mutation testing. In: 2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 97–102 (2017). https://doi.org/10.1109/ICSTW.2017.24
17. Maes, P.: Concepts and experiments in computational reflection. In: Proceedings OOPSLA '87, ACM SIGPLAN Notices. vol. 22, pp. 147–155 (Dec 1987). https://doi.org/10.1145/38807.38821
18. Martin, R.C.: Clean code: a handbook of agile software craftsmanship. Pearson Education (2009)
19. Miller, M.S., Samuel, M., Laurie, B., Awad, I., Stay, M.: Caja safe active content in sanitized javascript. Tech. rep., Google Inc. (2008), http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf
20. Miranda, E., Béra, C.: A partial read barrier for efficient support of live object-oriented programming. In: International Symposium on Memory Management (ISMM '15). pp. 93–104. Portland, United States (Jun 2015). https://doi.org/10.1145/2754169.2754186, https://hal.inria.fr/hal-01152610
21. Mouelhi, T., Le Traon, Y., Baudry, B.: Mutation analysis for security tests qualification. In: Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). pp. 233–242. IEEE (2007)
22. Paepcke, A.: User-level language crafting. In: Object-Oriented Programming: the CLOS perspective, pp. 66–99. MIT Press (1993)
23. Richards, G., Hammer, C., Burg, B., Vitek, J.: The eval that men do: A large-scale study of the use of eval in javascript applications. In: Proceedings of Ecoop 2011 (2011)
24. Rivard, F.: Smalltalk: a reflective language. In: Proceedings of REFLECTION'96. pp. 21–38 (Apr 1996)
25. Roberts, D., Brant, J., Johnson, R.E.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems (TAPOS) **3**(4), 253–263 (1997)

26. Roberts, D., Brant, J., Johnson, R.E., Opdyke, B.: An automated refactoring tool. In: Proceedings of ICAST '96 (Apr 1996)
27. Ruf, E.: Partial evaluation in reflective system implementations. In: Workshop on Reflection and Metalevel Architecture (1993)
28. Thomas, I., Ducasse, S., Tesone, P., Polito, G.: Pharo: a reflective language - A first systematic analysis of reflective APIs. In: IWST 23 - International Workshop on Smalltalk Technologies. Lyon, France (Aug 2023), https://inria.hal.science/hal-04217271
29. Tip, F., Laffra, C., Sweeney, P.F., Streeter, D.: Practical experience with an application extractor for java. In: Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 292–305. OOPSLA '99, Association for Computing Machinery, New York, NY, USA (1999). https://doi.org/10.1145/320384.320414, https://doi.org/10.1145/320384.320414
30. Wen, M., Liu, Y., Wu, R., Xie, X., Cheung, S.C., Su, Z.: Exposing library api misuses via mutation analysis. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 866–877 (2019). https://doi.org/10.1109/ICSE.2019.00093

A URLs to studied projects

Project	Url Link
STON	https://github.com/svenvc/ston
Microdown	https://github.com/pillar-markup/Microdown
Refactoring (a part of Pharo)	https://github.com/pharo-project/pharo
MuTalk	https://github.com/pharo-contributions/mutalk
Seaside	https://github.com/SeasideSt/Seaside

Table 4. Links to project repositories used for this analysis

B Additional STON matrices

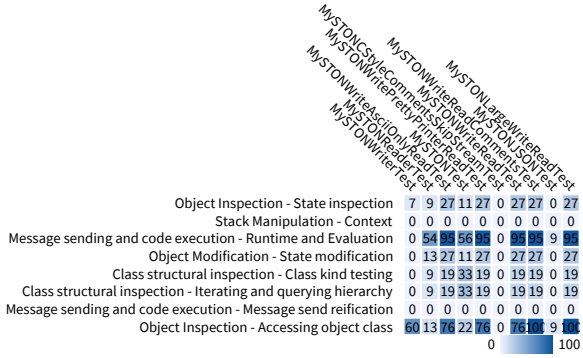

Fig. 6. Table of percentage of tests in a given class depending on a reflective category

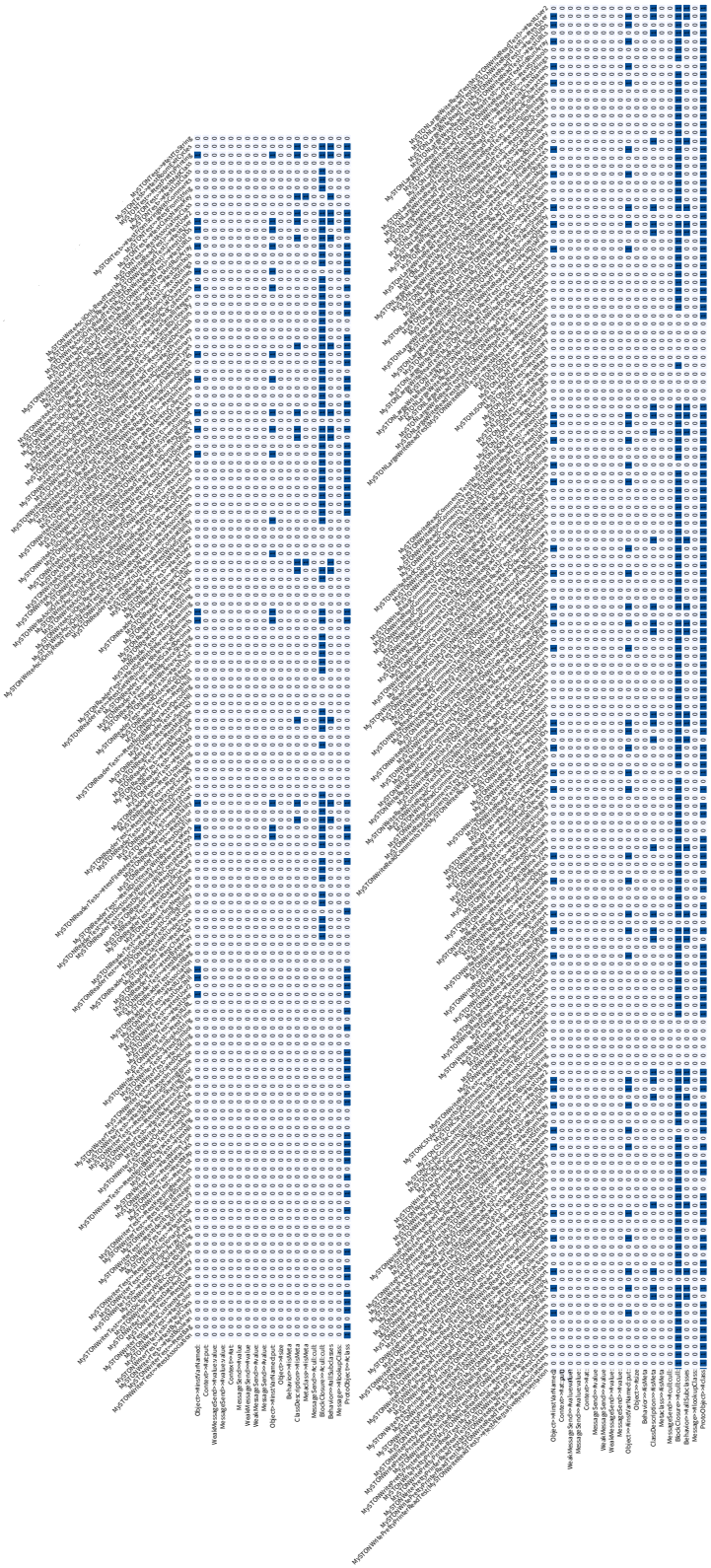

Fig. 7. Table of Tests depending on a reflective method. A 1 (blue cell) means that the test has failed when this reflective method is removed.