On the Use of Mined Stack Traces to Improve the Soundness of Statically Constructed Call Graphs

Li Sui, Jens Dietrich, Amjad Tahir
School of Engineering and Advanced Technology, Massey University, Palmerston North, New Zealand
Email: {L.Sui,J.B.Dietrich,A.Tahir}@massey.ac.nz

Abstract—Static program analysis is a cornerstone of modern software engineering – it is used to detect bugs and security vulnerabilities early before software is deployed. While there is a large body of research into the scalability and the precision of static analysis, the (un)soundness of static analysis is a critical issue that has not attracted the same level of attention by the research community.

In this paper we investigate the question whether information harvested from stack traces obtained from the GitHub issue tracker and Stack Overflow Q&A forums can be used in order to complement statically built call graphs. For this purpose, we extract reflective call graph edges from parsed stack traces, and check whether these edges are correctly computed by Doop, a widely used tool for static analysis with built-in support for reflection analysis. We do find edges that Doop misses when analysing real-world programs, even when reflection analysis is enabled. This suggests that mining techniques are a useful tool to test and improve the soundness of static analysis.

Index Terms—stack trace analysis, static analysis, empirical studies, soundness, call graph construction, mining software repositories

I. INTRODUCTION

Static program analysis is an approach to optimise code and to detect vulnerabilities and bugs early in the lifecycle of a program. An example of such an analysis is taint analysis [22]. The objective here is to establish whether the execution of code starting at dedicated program entry points (such as Java’s main methods) can execute code that can invoke methods such as Runtime.exec() with unsafe (tainted) parameter values. In other terms, static analysis is used to examine programs’ proneness to code injection attacks.

Static analysis requires to build models of suitable program abstractions, in this case, call graphs. Call graphs model the invocation of methods from call sites within other methods. Building a call graph by means of static analysis seems to be simple at first as method bodies contain references to the methods they invoke, but there are several complications. The first problem is precision, caused by the fact that virtual method calls must be resolved (“devirtualised”) to model dynamic dispatch. There are several established methods to deal with this such as Class Hierarchy Analysis (CHA) [10], taking into account method names, signatures and class hierarchy information. But this only yields a rather coarse over-approximation as it produces a large number of false positives: call graph edges that do not correspond to possible method invocations. A more precise method is to use the actual types of the objects referenced at the call site, computed by a simultaneously performed points-to analysis [21]. [26]. While this so-called call-graph-construction on-the-fly generally yields better precision, it comes at a price: the algorithms used are complex, usually in $O(n^3)$ (dubbed the “cubic bottle neck” of program analysis [17]), as graph reachability problems (plain transitive closure or CFL reachability) must be solved, and even the best known algorithms are still super-quadratic [9], [7].

While progress has been made to improve the precision and scalability of static analysis, this has brought another issue to the fore. Static analysis is sometimes portrait as sound (as the entire program is analysed) but not precise. On the other hand, dynamic analysis that “exercises” the program under analysis is unsound (as it misses the parts of a program not exercised by a harness or driver) but precise (as it only reports actual behaviour) [14]. However, modern programming languages are full of dynamic features that are difficult to capture by static analysis, such as reflection, proxies, dynamic class loading etc. Not capturing those features in a static analysis leads to under-approximation. This implies that the models analysed do not represent the entire program and the analysis is therefore also unsound as it produces false negatives. In applications like taint analysis, this can have serious consequences as the analysis would miss some vulnerabilities. The need for more research into the soundness of static analysis has been recently highlighted by Livshits et al. [20].

To illustrate this, consider the program in listing 1. This is a simple Java program that uses reflection. The reflective call site is in b(), the invoked method is c(). A sound static analysis should build the following call graph: main(String[]) → a() → b() → c() → d(). In this work, the issue we are interested in is that static analysis tools will have problems computing the b() → c() edge.

Many static analysis tools will try to infer this by trying to interpret type references in cast statements or string literals that occur in the method, class, program or even configuration files (such as web.xml in J2EE applications) as class and method name. For instance, in listing 1 the string literals in lines 8 and 9 can be directly interpreted as class and method names, respectively. But in general this is not that easy, as the references to the respective names and reflective objects must be tracked [21]. But even this approach does not guarantee

\[^1\text{There is an additional edge from main(String[]) to the constructor Foo() which we ignore as it is not relevant to our discussion.}\]
soundness if more complex programming patterns are used, e.g. if method names are passed as parameters, computed from
conventions, if non-standard file formats are used to configure
service class names or if methods are overloaded.

```java
import java.lang.reflect.Method;

public class Foo {
    public static void main(String[] p) throws Exception{
        new Foo().a();
    }
    void a() throws Exception {b();}
    void b() throws Exception {
        Class c = Class.forName("Foo");
        Method m = c.getDeclaredMethod("c", new Class[]{});
        m.invoke(this, new Object[]{});
    }
    void c() { d();}
    void d() { throw new RuntimeException();}
}
```

Listing 1. A simple Java program with a reflective call

In this paper, we evaluate whether data from public software
repositories such as GitHub\footnote{https://github.com} and Q&A sites such as Stack Overflow\footnote{https://stackoverflow.com} can be used in order to augment unsound statically constructed call graphs. The idea is that there are many users that will have exercised (executed) a program, and the stack traces represent partial actual program behaviour. This behaviour might even be particularly interesting and valuable in the sense that it has led to exceptions, and therefore is likely to be a behaviour that has not been encountered during a standard dynamic analysis procedure such as testing.

Consider again the program in listing\footnote{https://api.github.com/} in line 13, a runtime exception is thrown. When a user encounters this exception, the stack trace shown in listing\footnote{https://github.com} is generated.

```java
Exception in thread "main" $\$.InvocationTargetException
  at $\$.Method.invoke(Method.java:498)
  at Foo.b(Foo.java:7)
  at Foo.a(Foo.java:6)
  at Foo.main(Foo.java:3)
  ...
Caused by: java.lang.RuntimeException
  at Foo.c(Foo.java:12)
  ...
```

Listing 2. Stacktrace for the Java program in listing\footnote{https://github.com} $\$$. Exception is thrown in a method invoked through reflection, an InvocationTargetException exception is generated that wraps the original exception using Java’s standard exception chaining mechanism. The respective stack traces show this as two different blocks, with the method that has the reflective call site in the first block just below the reflective call site (Method.invoke), and the target at the bottom of the second block. This information can be used to infer the edge $b() \rightarrow c()$ and then add it to a statically constructed call graph. We discuss our data collection and analysis methodology in the following Section.

II. METHODOLOGY

We conducted experiments on the DaCapo 2009 data
set\footnote{https://github.com}, a Java benchmark that is widely used in program-
ing language research. We augmented this dataset

with several additional programs that are known to use re-
flexion, including log4j-2.1.4, antlr-4.0.1, hbase-hbase-client-
0.98.0-hadoop1, guava-11.0, spring-boot-loader-1.2.5 and
weld-core-impl-2.2.12 (by jboss).

We investigated the use of APIs and replicated datasets to
mine repository data, in particular the GitHub API\footnote{https://api.github.com/} and the GHTorrent project\footnote{https://github.com} that collects and publishes GitHub snapshots. Both had limitations that have rendered them as unsuitable for our study. The GitHub API does not support fine-grained text search, a limiting factor as we were interested to analyse pages from the issue tracking system containing certain strings. GHTorrent does not store the bodies of the actual issues, but only the respective metadata.

We therefore decided to write a custom HTML client to search within GitHub issue tracker and Stack Overflow Q&A sites for issues and discussions that include the text java.lang.reflect.InvocationTargetException.

The client mimics a web browser session using the appro-
riate headers, and returns the URLs of the respective static issue web pages. We then downloaded the respective web pages, stripped HTML markup and extracted stack traces, instantiating the meta-model depicted in Figure\footnote{https://github.com}. We also applied a filter to remove stack traces not containing any of the classes from our dataset. We extracted the respective class list using bytecode analysis on the respective programs. Then we inferred edges representing reflective method invocations from stack traces containing “caused by” clauses, as discussed above. The web scraps were performed on 19 January 2017.

We also built static call graphs using Doop\footnote{https://github.com} with and without reflection analysis enabled\footnote{https://github.com} and checked whether any of the edges extracted were missing from these call graphs. The complete process is depicted in Figure\footnote{https://github.com}. At the end of this process, the first and second author manually validated the results by investigating the respective source code. To validate the correctness of our approach, we also conducted a manual cross-validation of the first 100 recorded stack traces which found 17 false positives results. Those stack traces were then eliminated.

We have used the following Doop options to enable reflection analysis: --reflection --reflection-classic
--reflection-high-soundness-mode
--reflection-substring-analysis
--reflection-invent-unknown-objects
--reflection-refined-objects
--reflection-speculative-use-based-analysis

Fig. 1. Stacktrace metamodel

---

**Fig. 1. Stacktrace metamodel**

```
StackTrace

- thrown (1..1)
- StackTraceElement (1..many)
- className: string
- methodName: string
- lineNumber: int
- cause: (0..1)
```

---

https://github.com

https://api.github.com/

https://stackoverflow.com
III. RESULTS

A. Overview

We found 18,431 pages with references to java...InvocationTargetException, 11,932 from GitHub and 6,499 from Stack Overflow, from which we extracted 12,329 stack traces. From these stack traces we computed 11,920 reflective call graph edges. There are many cases of duplication, caused by stack traces reported on different pages describing the same call graph edge. After removing those duplicates, we ended up with 4,747 edges. Amongst those, 495 have a matching reflective call sites in our data set. To enable the comparative analysis with Doop, we further removed edges with targets outside the data set. We found 15 unique edges that we could cross-reference with the statically built call graphs. Not surprisingly, the number of relevant edges we found was low as the respective analysis tools tend to precisely model this. However, the class must extended with information read from project-specific configuration files. General-purpose static analysis tools are unlikely to precisely model this. However, the class must extend org.apache.hbase.filter.Filter, and a possible approach is to ensure soundness by over-approximating the analysis. This can be achieved by adding edges to all "parseFrom(byte[])

B. Fop in DaCapo-9.12

Fop is part of DaCapo. The Main class contains some reflective code that contains literals for the class and the method name of the invocation target, and the parameter types. Advanced static analysis tools should be able to correctly extract this edge, and this is indeed the case for Doop: if reflection analysis is enabled, Doop will find this edge.

C. Antlr4-4.0

Antlr is a popular parser generator. The reflective call we detected is similar to the invocation described above in Fop, although slightly more sophisticated: in GrammarTreeVisitor, the visitGrammar(GrammarAST, String) method invokes visit(GrammarAST, String) using the string literal "grammarSpec" as a second rule name parameter. The rule name is then interpreted as method name in visit(GrammarAST, String). Advanced static analysis tools that track string literals interpreted as method names across procedures would still be able to find this edge. Instead, Doop with reflection analysis enabled will find this edge.

D. Hbase-client-0.98.0-hadoop1

Hadoop is a popular framework for storing and processing big data. The reflective call detected in hadoop invokes a method with a fixed name ("parseFrom") and signature (byte[].class), both are defined within this method. But the class that provides the method is computed using a dynamic class loader that is configured with information read from project-specific configuration files. General-purpose static analysis tools are unlikely to precisely model this. However, the class must extend org.apache.hadoop.hbase.protobuf.ProtobufFilterList#parseFrom, and a possible approach is to ensure soundness by over-approximating the analysis. This can be achieved by adding edges to all "parseFrom(byte[])

E. Log4j-2.1

Log4j is a widely used logging framework. The use of reflection that creates the reflective call site is the most sophisticated we have encountered, however, this is common for frameworks that supports plugins. Reflection is

Listing 3. Reflective method invocation in fop, from https://goo.gl/JoXoam lines 140–143

```java
class clazz = Class.forName("org.apache.fop.cli.Main", true, loader);
Method mainMethod = clazz.getMethod("startFOP", new Class[] {
    String[].class});
mainMethod.invoke(null, new Object[] {args});
```

```
Method mainMethod = clazz.getMethod("startFOP",new Class[]{String[].class});
mainMethod.invoke(null,new Object[]{args});
```
IV. Threads to Validity

There are a few potential issues with the extraction process. We might have missed some stack traces that are formatted in unusual ways - for instance, stack traces produced by log frameworks that allow custom stack trace formatting. This would have given us some more results, but based on our own experience we think that this did not have a major impact.

There are several issues that could have caused false positives. Firstly, stack traces only contain methods names, but neither signatures nor descriptors. This can introduce false positives when methods are overloaded. Secondly, imprecise parsing could have produced false positives. We sampled 100 results to validate the correctness of parsed stack traces, 17 false positives were found. Thirdly, stack traces lack version information, although in some cases version information can be found on the enclosing web sites. This means that we might have extracted edges not present in the version of the program we statically analysed. We have mitigated this issue by running a script that matched the line numbers found in stack traces against program versions, and then selected the best matched version for future analysis.

We addressed all of the issues related to precision by manually checking those 15 edges obtained against the version of the source code of the program we analysed with Doop.

The scripts used to extract and process stack traces are available from the repository.

V. Related Work

A. Reflection and Static Analysis

Several authors have tried to improve the handling of reflection by static analysis. We only discuss the most influential work, for a recent in-depth discussion of works in this area readers are referred to the recent literature review by Landman et al [19]. Livshits et al [21] have investigated how to use points-to analysis to handle reflection. They associate objects with reflective call sites, and track strings that are interpreted as class names. Furthermore, they use information in casts often associated with reflective object creation sites to improve the precision of their analysis. They demonstrated that with this approach, a large percentage of Class.forName calls can be resolved. This approach relies on the precision of the underlying points-to analysis, and therefore users must make trade-offs between precision and scalability when applying this approach. Smaragdakis et al [25] have further refined this approach by adding substring and string flow analysis, further improving soundness.

Wala [1] is another static analysis tool developed by IBM. It employs a context-sensitivity policy to deal with reflections. Constructs like Class.forName(), Class.newInstance(), Method.invoke() are supported by this library.

In our recent work, we discussed some specific Java features that may cause unsoundness of static analysis [13].

B. Hybrid Analysis

One possible approach to tackle the unsoundness of static analysis is to combine it with a dynamic (pre-) analysis that informs the static analysis by recording actual program behaviour.

Bodden et al [4] proposed a tool called tamiflex to improve the soundness of static analysis. Tamiflex is a hybrid analysis tool that uses additional information inferred from recorded program runs to improve call graphs. The quality of these information relies on the fact that the driver (harness) used in order to exercise the program has a high coverage. The overall idea of tamiflex is similar to our approach in that the mined stack traces can be considered as partial execution logs. tamiflex is now widely used with the popular soot analysis platform. Grech et al [16] have extended this approach by also harvesting information from heap dumps that is then fed back into the static analysis performed by Doop in the form of additional relations.

C. Stack Trace Analysis

Several works have looked into mining and analysing large stack traces for the purpose of program comprehension and analysis. Kim et al [18] proposed a technique called "crash graph" that uses stack traces from crash reports to predict fixable crashes and detecting duplicate crash reports. Sinha et al [24] provided a stack trace - based approach (based on static and dynamic analysis) to locate and fix bugs that are caused by runtime exceptions. Schröter et al. [23] studied how stack traces can be helpful for developers to locating and fix bugs. The study found that developers tend to fix bugs faster when the stack traces are reported with an issue. Cabral and Marques [6] analysed stack traces of 32 Java and .NET open-source projects. They found that exceptions are not being correctly used by developers as error recovery mechanism. They observed that the actions inside handlers were often very simple (e.g. logging and presenting a message to the user). Coelho et al [8] studied Java stack traces from over 600 open source Android project. The goal of this study was to understand bug hazards related to exception handling code. The study also surveyed their findings with 71 real App developers to understand how developers perceive the
exception handling that leads to bug hazards (i.e. the potential of introducing bugs).

VI. CONCLUSION

In this paper, we have investigated the question whether information harvested from online data sources can be used to check and improve the soundness of static program models. The short answer is yes. Even with reflection analysis enabled, Doop will find only 4 out of the 15 edges we extracted. While we have to be cautious to generalise this, it is obvious that there are many usages of dynamic language features that even the most advanced static analysis cannot capture.

We argue that while our analysis does not provide a large number of call graph edges that can enhance the static analysis, it is useful to retrieve interesting (and in this sense, high-quality) edges that can point to the weaknesses of static analysis tools. This is hardly surprising: it seems more likely that a programmer encounters an exception or error if the software is used in a way that was not intended by the programmer, e.g. by bypassing program invariants or boundary checks. But those are exactly the cases of interest to static analysis as it has the ambition to discover those cases in order to reveal bugs and vulnerabilities.

One could argue that the use of hybrid analysis [4], [16] addresses problems with (un-)soundness. The main challenge is to create drivers (harnesses) that exercise the “unsound parts” of a program. The use of test case generation / fuzzing for this purpose is promising [12]. It seems that hybrid analysis techniques can mitigate, but not solve the problem. Extending this study by cross-referencing the call graphs with the call graphs produced by tamiflex or similar tools is an interesting and relevant topic.

An extended study with a larger dataset is an interesting topic for future research. One particular interesting issue is the study of call graphs that cover multiple projects and libraries, including frameworks known for their heavy use of reflection (plugin-based systems, dependency injection) and the Java core libraries. We have noticed a large number of reflective invocation where call site and target were in different libraries. One potential problem here is that it is still challenging to build comprehensive and sufficiently precise static models for real-world programs that include all library dependencies, although new algorithms and tools are under development to address scalability issues [11].

ACKNOWLEDGEMENT

The authors would like to thank Georgios Gousios and Davy Landman for their comments. This work was funded by a New Zealand NSC-SEED Project (PROP-52515-NSCSEED-MAU).

REFERENCES