南京大学
NANJING UNIVERSITY

API误用的文献综述报告

小组成员：惠天宇 陈恒 唐鹏

报告时间： 2020年5月20日

大纲

- 一、API误用的相关概念
- 二、研究现状
- 三、各论文内容总结
- 四、目前存在的问题
- 五、可进一步研究的问题
- 六、可探讨的研究思路

一、API误用的相关概念

- 应用程序编程接口（API）通常具有使用约束，例如对调用顺序或调用条件的约束。API误用（即违反这些约束）可能会导致软件崩溃，错误和漏洞。
- API误用有多种情况，比如多余的API调用、缺少了API调用或者错误的API调用等。例如在java中打开一个文件进行写入，如果写入完成后没有关闭文件就会导致文件写入失败，这就属于API误用缺陷。

一、API误用的相关概念

- 开发环境应帮助开发人员实现正确的用法以及发现和解决现有的API误用问题。
- 为了解决这个问题，提出了许多用来自动推断API使用规范并通过静态代码分析来识别API误用行为的工具以及相应的评估基准

二、研究现状

当前对于API误用问题的研究主要集中于三方面：

- API误用模式的检测提取方法[1][4][6]；
- 训练API误用检测器的数据来源[3]；
- 评估不同API误用检测器的标准指标[2][5]。

二、研究现状

（1） API误用模式的检测提取方法

- 早期的静态误用检测器通常都是通过从API文档规范或大量正确的API使用代码中挖掘使用模式（usage patterns），如[1][6]。它们通常认为API的使用方式越常见越频繁，则越有可能是正确使用模式。其具体的区别在于，针对不同的编程语言语法特点和应用场景，使用了不同的提取模式和发现使用违反的方法。

二、研究现状

- [4]提出的基于Mutation的方法避免了从API正确用例中进行挖掘，而是将API误用看作正确用法的变异。通过少量的正确用法可以mutate出大量API用法的变种，然后对mutate出来的用法可以通过执行测试用例、分析执行信息来验证是不是API误用。这一方法有效提高了API误用检测器准确率和召回率低的问题。

二、研究现状

（2）训练API误用检测器的数据来源

API的误用的训练数据来源通常有以下几种：

- 开源软件项目的API误用commit（通过关键词过滤commit）
- API的官方规范使用代码示例
- Stack Overflow等Q&A网站上的相关API使用回答
- …

- [3]说明了来自于Stack Overflow这类Q&A网站上的API用例的不可靠性，即使该代码用例是被采纳的答案或高票答案。

二、研究现状

（3）评估不同API误用检测器的标准指标

- [2]首先提出了一个用于评价API误用检测器的基准数据集MUBench，该数据集来自于33个真实的Java项目和一个调查报告，它可以用来检测API误用检测器可以检测到多少种API误用。
- [5]在[2]的基础上进行了扩展，提出了API误用分类法框架（MUC）和自动化的用于API误用检测器比较的pipeline流水线（MUBENCHPIPE）。

三、各论文内容总结

MUBench: a benchmark for API-misuse detectors

- 该文章主要工作是提出了一个名为MUBench的数据集，用来作为API误用检测器的评价标准，该数据集标准是后面相关API误用检测论文评估标准的起始点。
- 同时，本文还使用该数据集对API误用进行了分析，并讨论了如何使用该数据集对API误用检测器进行评估。

三、各论文内容总结

MUBench: a benchmark for API-misuse detectors

使用的鉴别API误用的三种方法

- 分析现存的bug数据集
  - BugClassify（由来自五个开源项目的问题跟踪程序的7401个tickets组成）
  - Defects4J（由在一次提交中修复的357个源代码bug组成）
  - iBugs（包含来自三个开源项目的390个fix提交）
  - QACrashFix（由16个GitHub项目的24个源代码错误组成）
- 分析SourceForge和GitHub中的Java Cryptography Extension (JCE) APIs
  - 识别了至少有10个stars的使用JCE API的项目
  - 对每个fix，提取并用抽象语法树分析源代码更改
  - 手动检查了candidates
- 进行了一项调查，询问开发人员因Java API误用而引起的问题
  - 收集了16个调查回复，指出了17种API误用情况

三、各论文内容总结

MUBench: a benchmark for API-misuse detectors

数据集中的元数据说明

- Source：误用的数据集位置信息
  - Name：当前分析的数据集
  - url：原始bug数据集的url
- Project：误用发生的项目位置信息
  - Name：项目名称（来自survey的无）
  - url：项目url
- Report：指明ticket误用报告
- Description：误用的具体描述
- Crash：是否会导致crash
- Internal：是否是项目内部的API
- Api：api属于误用或正确用法的所有类型

```
source:
  name: BugClassify
  url: https://www.st.cs.uni-saarland.de/softevo//bugclassify/
project:
  name: Mozilla Rhino
  url: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
report: https://bugzilla.mozilla.org/show_bug.cgi?id=286251
description: >
  IRFactory.initFunction() is called twice along one possible execution path,
  which causes an infinite loop.
crash: yes
internal: yes
api:
  - org.mozilla.javascript.IRFactory
characteristics:
  - superfluous call
pattern:
  - single object
challenges:
  - path dependent
fix:
  description: >
    Remove duplicated call.
  commit: https://github.com/mozilla/rhino/commit/ed00a2e83de1e768918604a65def097...
  files:
    - name: src/org/mozilla/javascript/Parser.java
```

Figure 1: Meta Data of API Misuse RHINO-286251

三、各论文内容总结

MUBench: a benchmark for API-misuse detectors

- Characteristics：该误用具有的特征
  - 可能的取值包括：superfluous call、missing call、wrong call、missing precondition、missing catch、missing finally、ignored result
- Pattern：该误用发生提取的方式
  - 可能的取值包括：single node、single object multiple object
- Challenges：识别该误用的难点
  - 可能的取值包括：multi-method、multiple usages、path dependent
- Fix：该误用的fix信息
  - Description：来自补丁和API文档的描述
  - Commit：fixing commit的url
  - Files：列出修改的相关文件

```
source:
  name: BugClassify
  url: https://www.st.cs.uni-saarland.de/softevo//bugclassify/
project:
  name: Mozilla Rhino
  url: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Rhino
report: https://bugzilla.mozilla.org/show_bug.cgi?id=286251
description: >
  IRFactory.initFunction() is called twice along one possible execution path,
  which causes an infinite loop.
crash: yes
internal: yes
api:
  - org.mozilla.javascript.IRFactory
characteristics:
  - superfluous call
pattern:
  - single object
challenges:
  - path dependent
fix:
  description: >
    Remove duplicated call.
  commit: https://github.com/mozilla/rhino/commit/ed00a2e83de1e768918604a65def097...
  files:
    - name: src/org/mozilla/javascript/Parser.java
```

Figure 1: Meta Data of API Misuse RHINO-286251

三、各论文内容总结

MUBench: a benchmark for API-misuse detectors

- 使用该数据集评估了API误用检测器的两个方面：
  1. 在一个误用发生的特定上下文中识别该误用的能力
  2. 一般情况下识别误用的能力
- 使用方法：
  - 为解决1，提取了该误用最初产生的代码，MUBench中的commit URL可以完成这一点
  - 为解决2，提取与该误用有关的最小实例
  - 还提取了各实例的不同fixed版本以增加评估的负样本点

三、各论文内容总结

MUBench: a benchmark for API-misuse detectors

- 局限不足之处：
  - 数据集仅包含来自实际项目的77个API误用和来自调查的12个误用，数据量较小
  - 识别的API误用来自于33个项目，可能无法代表通用的API误用情况
  - 识别出的实例可能无法代表实际会出现的代码误用（但注意：该数据集仅用来评估API误用检测器而非评估实际代码）

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- 该文章主要针对C语言的SSL API库，提出了一种约束导向的静态分析技术SSLDoc用来检测代码中的API误用错误。此外，通过研究真实的API误用错误，提出了一种涵盖大多数类型的API使用约束的规范Ispec。
- 本文提出的技术主要的工作流程是设计并实现了SSLDoc来自动将ISpec解析为验证对象，然后使用一个静态分析引擎来识别潜在的API误用，并使用语义信息来删除错误的识别。

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- 文章首先从一些开源项目中分析了API误用的一般模式，这些项目包括：Linux kernel、OpenSSL、FFmpeg、Curl、FreeRDP和Httpd。
- 分析步骤：
  - 提取与bug-fix有关的commit，使用关键字过滤的方法
  - 在上一步基础上收集与API误用有关的commit

Table 1. Empirical study subjects.

Project	Loc	Studied period	Commit	Bug fix	API misuse
Linux	12.96 M	20170901–20171231	24651	6401	868
OpenSSL	454 K	20150701–20171231	7564	2391	529
FFmpeg	915 K	20160701–20171231	8162	2783	610
Curl	113 K	20130101–20170630	7082	2043	499
FreeRDP	259 K	20130701–20171231	7565	3535	495
Httpd	203 K	20130701–20171231	6072	1323	149
Total	14.90 M	—	61096	18476	3150

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- 根据收集到的3150个相关的API误用实例，手工调查了其中的830的具体情况，得到三种通用的API误用分类：
  - 不恰当的参数使用——Improper Parameter Using (IPU)
  - 不恰当的错误处理——Improper Error Handling (IEH)
  - 不恰当的成对因果调用——Improper Casual Calling (ICC)

Table 2. Investigated API misuse bugs and patches.

		API misuse		Investigated		IPU		IEH		ICC		Other	
Project	# of Bug Fix	# of	$Rate_1$	# of	$Rate_2$	# of	$Rate_3$	# of	$Rate_3$	# of	$Rate_3$	# of	$Rate_3$
Linux	6401	868	13.56%	283	32.60%	43	15.19%	96	33.92%	77	27.21%	67	23.67%
OpenSSL	2391	529	22.12%	127	24.00%	21	16.54%	42	33.07%	49	38.58%	15	11.81%
FFmpeg	2783	610	21.92%	126	20.66%	18	14.29%	43	34.13%	52	41.27%	13	10.32%
Curl	2043	499	24.42%	134	26.85%	23	17.16%	38	28.36%	57	42.54%	16	11.94%
FreeRDP	3535	495	14.00%	119	24.04%	22	18.49%	30	25.21%	48	40.34%	19	15.97%
Httpd	1323	149	11.26%	41	27.52%	8	19.51%	8	19.51%	16	39.02%	9	21.95%
Total	18476	3150	17.05%	830	26.35%	135	(16.27%)	257	(30.96%)	299	(36.02%)	139	(16.75%)

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- 文章提出的ISpec可以描述API的使用约束，其中包含了上下文的大多数语义信息
- 针对每种API的ISpec包含两部分：①参数使用的前置（pre）条件②错误处理和因果调用的后置（post）条件

$$
\begin{aligned}
Specs &:= Spec^* \\
Spec &:= \texttt{Spec:}\ Target\ Pre\ Post \\
Target &:= \texttt{Target:}\ FunSig \\
Pre &:= \texttt{Pre:}\ Cond^+ \\
Post &:= \texttt{Post:}\ (Cond, Action^*)^+ \\
Cond &:= \texttt{true} \mid Opd\ CmpOp\ Opd \mid Opd\ MemberOp\ (Set) \\
Action &:= Return \mid Call \\
Return &:= \texttt{RETURN(}n\texttt{)} \mid \texttt{RETURN(NULL)} \\
Call &:= \texttt{Call(}FunName\texttt{:}Cond^*\texttt{)} \\
Opd &:= Arg \mid UnOp\texttt{(}Arg\texttt{)} \mid \texttt{NULL} \mid n \\
FunSig &:= FunName\texttt{(}Type^*\texttt{) -> }Type \\
Arg &:= FunName\texttt{\_arg\_}i \\
UnOp &:= \texttt{LEN} \mid \texttt{TYPE} \mid \texttt{MEMTYPE} \\
CmpOp &:= \texttt{!=} \mid \texttt{==} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{<} \\
MemberOp &:= \texttt{IN} \mid \texttt{NOTIN} \\
Set &:= id^+ \mid n^+ \\
FunName &:= id \\
Type &:= id
\end{aligned}
$$

Fig. 5. Abstract syntax of IMSpec.

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- Pre描述了API调用前需要满足的前置条件。每个Pre由一系列Conds条件组成，表示正确的前置用法。
- Post描述了用于错误处理和因果调用的用法约束，有两部分：
  - 用于API返回的错误码检查的Cond条件
  - 当该Cond为true时的动作Actions，Actions可以分为：
    - Return语句
    - 调用其他特定的API

```
1   // ISpec for fopen, which opens the file specified in the first
        parameter. If fails, a NULL pointer will be return.
2   Spec:
3   Target: fopen(char*, char*) -> FILE*
4   Pre:
5   - fopen_arg_1 != NULL,
6   - fopen_arg_2 IN (r, w, a, r+, w+, a+)
7   Post:
8   // failure status and error handling actions specific in example
9   - fopen_arg_0 == NULL, RETURN(FILEERR);
10  // success status, close file handler
11  - fopen_arg_0 != NULL, CALL(fclose: fopen_arg_0 == fclose_arg_1)
12
13  // ISpec for fgets, which reads charactors and stores them. If a read
        error occurs, a NULL pointer is returned.
14  Spec:
15  Target: fgets(char*, int, FILE*) -> char*
16  Pre: // omit single parameter validation
17  - LEN(fgets_arg_1) >= fgets_arg_2
18  Post:
19  // failure status
20  - fgets_arg_0 == NULL, CALL(log: true), RETURN(IOERR)
```

Fig. 6. ISpec instances for misused APIs in Fig. 1.

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- SSLDoc使用ISpec自动分析检测源代码中的bug。分为三步：
  - 解析源代码和Ispec构建控制流图
  - 对规范中的每个目标API，选择它的调用处作为分析入口
  - 对每个调用处，执行符号和范围分析产生带有语义信息的一系列程序路径

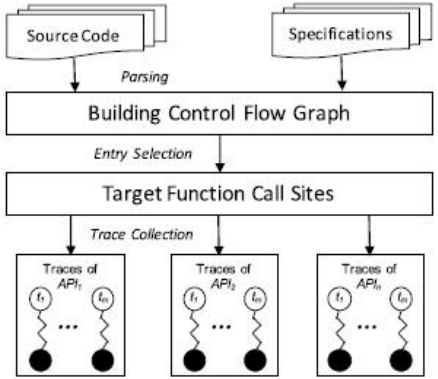


Fig. 7. Workflow of building analysis contexts.

```
t1 : 1_Call fopen((1_fopen_arg1), (1_fopen_arg2));
     2_Assume(1_fopen_arg_0 == NULL) ;
     3_Call Log(...);
     4_Return (-1);
t2 : 1_Call fopen((1_fopen_arg1), (1_fopen_arg2));
     5_Assume(1_fopen_arg_0 != NULL) ;
     6_Call fgets((6_fgets_arg1), (6_fgets_arg2), (1_fopen_arg0));
     7_Assume(6_fgets_arg_0 < 0) ;
     8_Call Log(...);
     9_Return (-2);
t3 : 1_Call fopen((1_fopen_arg1), (1_fopen_arg2));
     5_Assume(1_fopen_arg_0 != NULL) ;
     6_Call fgets((6_fgets_arg1), (6_fgets_arg2), (1_fopen_arg0));
     10_Assume(6_fgets_arg_0 >= 0) ;
     11_Call fclose((1_fopen_arg0));
     12_Return (1);
```

Fig. 9. Program path traces of the code in Fig. 1.

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- 使用Specs中定义的约束和生成的程序路径来检测API误用bug

Algorithm 1. Algorithm for checking API misuse bugs

Input: program path traces T, specifications $Specs$
Output: bug report R

```
1:  R ⇐ ∅
2:  APISet ⇐ extractTargetAPISet(Specs)
3:  for each API f ∈ APISet do
4:    spec ⇐ extractSpec(Specs)
5:    T' ⇐ extractPathTraces(spec, T)
6:    for each trace t ∈ T' do
7:      result ⇐ satisfy(t, spec)
8:      if (!result) then
9:        R ⇐ addBug(t, spec)
10:     end if
11:   end for
12: end for
13: return R
```

针对检测到的API误用报告，首先使用基于语义的过滤，然后基于使用进行排名。最高可能值意味着最正确的用法

三、各论文内容总结

API Misuse Detection in C Programs: Practice on SSL APIs

- 在实验部分中，作者使用OpenSSL的API来实例化SSLDoc，然后将其应用于大型C程序，特别是那些使用SSL API的程序。
- 在OpenSSL的实现和Ubuntu上进行实验后，得到的结果发现了先前未发现的45个未知bug。

三、各论文内容总结

Exposing Library API Misuses via Mutation Analysis

A、存在问题

- 1.第三方API库使用文档不规范，导致API误用
- 2.API库的误用会导致软件崩溃或者系统脆弱。
- 3.目前存在的静态分析工具因为过于简化的假设，系统精度较低

B、解决方法

观察依据：

- 1.API误用可以表示为其正确用法的变体
- 2.可以通过针对测试套件对突变体测试，并分析反馈信息来验证突变是否会导致误用

根据以上观察，提出通过突变分析检测API误用的系统——MUTAPI，通过本论文中提出的8种运算子+目标API生产API误用变体，分析变体总结出API误用的patterns，最终根据patterns识别出现有系统中的API误用。

三、各论文内容总结

Exposing Library API Misuses via Mutation Analysis

B、解决方法——MUTAPI框架

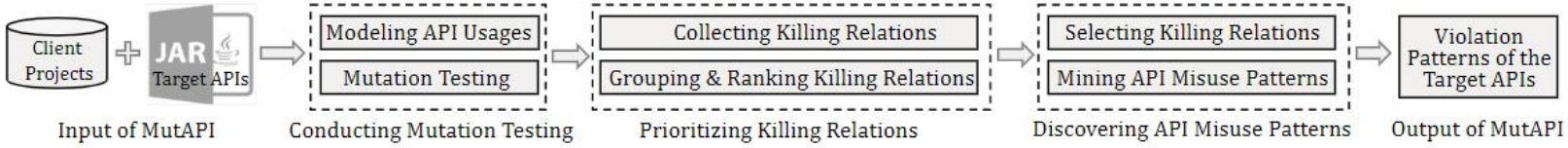


Fig. 4: Overview of MUTAPI

框架组成：

1.输入：客户端项目(源代码及相关测试套件)+目标API

2.MUTAPI对一组目标API的程序执行运算子生产变体，并在测试套件中测试

3.收集测试bug结果，针对每个目标API对这些结果排序

4.选择排名最高的bug信息，并从中挖掘目标API误用的patterns

B、解决方法——三项挑战

1.如何有效生成模仿API误用的突变体是关键（API三种误用：缺失，冗余和不正确）

解决办法：设计了八种类型的变异算子，其目的是以系统的方式主动违反API用法

2.如何验证突变体是否确实引入了API误用

解决办法：MUTAPI 通过分析终止测试的失败堆栈追踪来解决这一难题。

3.如何有效地提取API误用模式

解决办法：MUTAPI 通过从大量已经确定API误用的变体中提取。

C、评估实验

评估数据源：

a.Target API：引文[4]中的73种流行Java API，其中43种是Stack OverFlow种讨论最频繁的API，其余30个API来自MUBENCH (API误用的基准数据集)。

主要来自于四个领域：Common Library, GUI , Security , Database.

b.Client Projects：

TABLE V: Selected Client Projects

Category	Project	#Covered API	#Source	#Test	#KLOC
GUI	Apache **FOP** [42]	35	1577	391	363.2
	SwingX [43]	28	551	340	215.4
	JFreeChart [44]	19	637	350	294.8
	iTextPdf [45]	34	894	609	298.6
Library	Apache **Lang** [34]	24	153	174	144.1
	Apache **Math** [46]	19	826	498	307.1
	Apache **Text** [47]	17	85	63	40.8
	Apache **BCEL** [48]	23	417	75	75.7
Security	Apache **Fortress** [49]	22	214	74	122.2
	Santuario [50]	24	467	198	135.6
	Apache **Pdfbox** [51]	28	598	103	154.1
	Wildfly-Eytron [52]	36	763	151	161.3
Database	Jack**Rabbit** [53]	34	2443	646	611.4
	Apache **BigTop**	22	176	52	20.4
	H2Database [54]	20	566	317	305.4
	Curator [55]	18	197	52	20.4

C、评估实验——三个评估问题及结果

1.MUTAPI可以发现API误用模式吗？ 它可以更容易地发现哪些误用模式?

MUTAPI可以高精度地发现实际的API误用模式(图6，表7)。MUTAPI在检测那些会引发未检查异常的API的误用方面更有效，这是因为代表此类误用的突变体更有可能以运行时异常的形式被杀死。

2.MUTAPI是否可以检测最新基准数据集MUBENCH上的API误用实例?

MUTAPI能够检测出53种实际API误用中的26种。 它实现了0.49的最高召回率。

3.与传统的变异算子相比，新提出的在检测API误用模式方面表现如何?

使用两个指标将建议的突变算子与PIT [27]中使用的传统算子进行比较：（1）效率（即，突变分析所需的时间）和（2）有效性（即，API的数量）。

D、结论与展望

在这项研究中提出了MUTAPI， 与现有方法相比有两个优点。首先，它不需要大量正确的API使用示例。 因此，它可以用于检测新发布的API的误用模式。 其次，它摆脱了以往简单的假设，即偏离最常见的模式是一种API误用。 针对73个流行的API在16个客户端项目上应用了MUTAPI。 结果表明，MUTAPI可检测大量的API误用模式。 它还在MUBENCH基准数据集[6]上实现了0.49的调用率，大大优于现有方法。

1. 计划将MUTAPI应用于不那么常用的API（例如，来自新发布的库的API），而不是流行的API，以调查MUTAPI是否可以检测未知的API误用模式。

2.计划系统地研究测试套件质量对方法有效性的影响。

三、各论文内容总结

A Systematic Evaluation of StaticAPI-Misuse Detectors

A、存在问题及解决办法

- 1.第三方API库使用文档不规范，导致API误用
- 2.API库的误用会导致软件崩溃或者系统脆弱。
- 3.尽管存在许多API误用检测器，但API误用现象仍很普遍

本文在误用数据集MUBENCH的基础上，开发了API误用分类器MUC和用于检测器比较的自动化基准MUBENCHPIPE。

B、MUC——Classification

两个维度表示API误用种类 ：
violation type 和 **API-usage element**

API-usage element：Method calls , Conditions,Iterations, 和Exception handling

violation type：Missing和Redundant

TABLE 2
The Misuse Classification (MuC), with the Number of Misuses with a Particular Violation in MuBench

API-Usage Element	Violation Type	
	Missing	Redundant
Method Call	30	13
Condition	48	6
`null` Check	25	3
Value or State	21	2
Synchronization	1	1
Context	1	1
Iteration	1	1
Exception Handling	10	1

C、实验——数据集

设计三个实验，以测量探测器的精度和召回率。

API误用检测器：JADET，GROUMINER，TIKANGA和DMMC

API误用数据集[24]：

TABLE 1
Datasets Used Throughout this Paper, with the Number of Hand-Crafted Misuses (#HM), the Number of Real-World Projects (#P), Project Versions (#PV), and Misuses (#RM), and the Total Number of Misuses (#M)

	Dataset	#HM	#P	#PV	#RM	#M
1	Original MuBench [3]	17	21	55	73	90
2	Extended MuBench	27	21	55	73	100
3	Experiment P	n/a	5	5	n/a	n/a
4	Experiment RUB	25	13	29	39	64
5	Experiment R	0	13	29	53	53

"n/a" denotes that the number is not relevant for the use of the dataset.

C、实验——结果

1.实验P用于评估检测器精度：所有探测器的精度都极低（低于12%）。 平均而言，他们在前20个调查结果中报告的实际误用少于1.5个。

2.实验RUB用于评估检测器检测能力：所有检测器的实际召回上限都比其理论召回上限低得多；检测器的发现常常与其理论能力有所不同。

3.实验R用于评估检测器召回率：JADET仅发现了在实验P中已经发现的三种误用情况。GROUMINER找不到任何误用情况。TIKANGA发现了在实验P中已经发现的五种误用，一种是在实验P中DMMC识别出的误用，另一种是在实验P中JADET发现的误用。DMMC在实验R中显示出最好的召回率。

三、各论文内容总结

SAFEWAPI: Web API Misuse Detector for Web Applications

- 这篇论文主要提出了一个铭文SAFEwapi的工具用来分析web API和JavaScript应用，使得更好的使用web API，以及发现网络应用中使用api可能导致的问题。
- SAFEWAPI是一个静态分析器，用于检测JavaScript web应用程序中的API误用。
- 为解决分析JavaScriptweb应用程序的问题
  - 1.使用常用接口语言（IDL）声明的web API规范作为定义API误用的标准。
  - 2.从API规范中推断出连接web应用程序和API实现之间的控制流的重要遗漏信息。（idl：交互式数据语言）
- SAFEwapi从API规范中收集有关API功能和对象的信息，根据规范进行建模，分析JavaScriptweb应用程序以发现其中的API误用。

- 根据在设计工业Web API和使用Web API评估JavaScript Web应用程序方面的实际经验，确定了6种常见的Web API误用模式：
  - 1.访问缺失属性的平台对象（AbsProp）
  - 2.API函数调用的参数数目错误（ArgNum）
  - 3.缺少错误回调函数（ErrorCB）
  - 4.可能引发异常的未处理的API调用（ExnHnd）
  - 5.API函数调用的参数类型错误（ArgTyp）
  - 6.访问字典对象的缺失属性（AbsAttr）

- 技术细节
  - 为通过web API函数的规范来填充缺失信息，采用如下方法分析执行流
    - 1.为分析从API函数到JavaScript代码的显式执行流，检查到API函数的JavaScript参数值是否满足相应API规范的参数声明。
    - 2.为分析从JavaScript代码到API函数的显式执行流，检查JavaScript代码是否正确的使用API函数调用的返回值。
    - 3.为分析隐含在API函数之间的隐式执行流，检查JavaScript回调函数是否正确使用了平台生成的对象。

- **技术细节**
  - **基于类型的JavaScript分析**
    - 论文中的工具分析JavaScript web应用的方法建立在现有的静态分析框架之上，该框架检测JavaScript程序中类型相关的错误。框架设计是可插入的（pluggable），因此研究人员可以使用自己的接口来替换默认的抽象域和分析器。
  - **JavaScript值和web IDL类型**
    - 为检查web API函数是否接收有效的JavaScript值作为参数，应该检查值是否满足web IDL中相应的参数类型要求。JavaScript只有5个原始类型，web IDL却有更丰富的类型，应该考虑在JavaScript中的一组隐式类型转换规则

技术细节

平台生成对象的自动建模

- 为检查JavaScript web应用是否正确的使用了API函数调用的返回值，论文中对平台生成对象的自动建模来表示从API调用中的返回值，并且将它们作为抽象值来分析JavaScript应用
- 使用API规范中定义的类型生成的抽象类型值，我们可以分析从API函数到JavaScript代码码之间的显式执行流，以此可以检查JavaScript程序是否正确的使用了平台生成对象。

API函数的自动建模

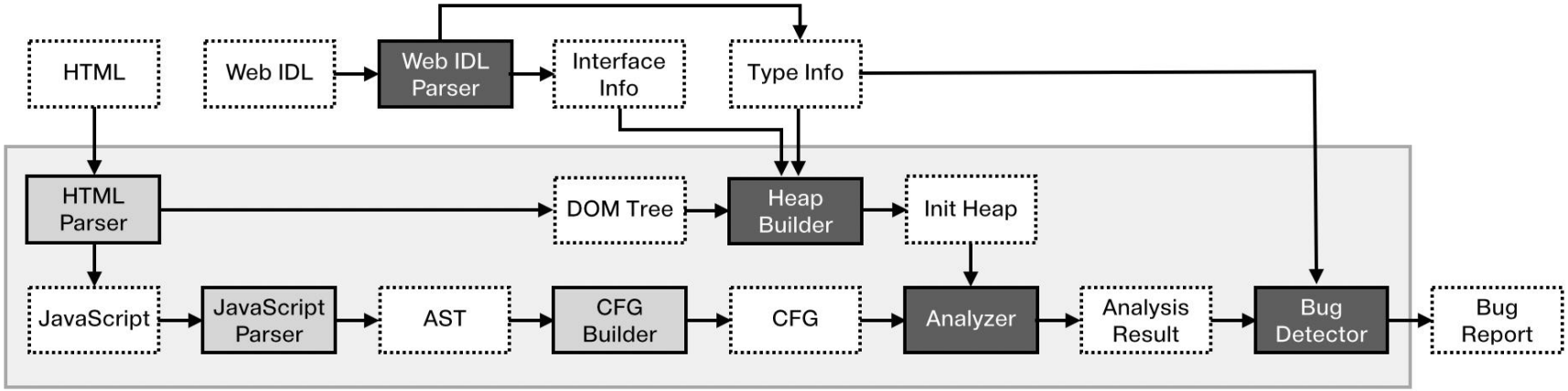
- 通过回调函数的调用来分析JavaScript代码和API函数之间的隐式执行流，对自动调用回调函数的API函数调用进行建模

- **实现**
  - **SAFE：基于类型的JavaScript分析**
    - 通过拓展SAFE来建立SAFEWAPI
    - SAFE对HTML文档或目录进行分析，通过常规编译步骤将其转换为cfg（控制流图），最后分析cfg中的JavaScript程序。并且建立DOM树，并通过堆生成器构建一个包含全局信息的初始堆，以使用DOM树分析web应用程序
  - **SAFEWAPI（添加了Web IDL解析器）**
    - web IDL解释器解释web IDL中给出的API规范并且抽取必要的信息来分析API函数。其次，使用web API中额外的接口和类型信息生成一个初始堆，添加了根平台对象webapis，包含了web IDL解释器建立在初始堆上的所有接口对象。然后，拓展的分析器通过使用web API函数使用初始堆中的抽象类型值建模的函数来分析web API函数调用。拓展的bug检测器使用分析器的结果来检测web API误用的模式。

三、各论文内容总结

SAFEWAPI: Web API Misuse Detector for Web Applications

- **SAFEWAPI结构图**



三、各论文内容总结

SAFEWAPI: Web API Misuse Detector for Web Applications

App. Id	AbsProp	ArgNum	ErrorCB	ExnHnd	ArgTyp	AbsAttr	Total
tv-1	**1**	0	0	0	0	0	**1**
tv-4	0	0	1	11	0	0	**12**
tv-8	1	0	0	1	0	0	**2**
tv-15	1	0	0	0	0	0	**1**
tv-17	0	0	1	0	0	0	**1**
tv-19	0	0	1	0	0	0	**1**
tv-20	2	0	0	0	0	0	**2**
mb-2	0	0	0	2	0	0	**2**
mb-3	0	0	0	2	0	0	**2**
mb-5	0	0	0	1	0	0	**1**
mb-6	0	**2**	0	51	0	0	**53**
mb-7	0	0	0	4	0	0	**4**
mb-12	0	0	0	1	0	0	**1**
mb-15	0	0	0	0	0	0	**0**
mb-19	4	0	0	3	0	0	**7**
mb-20	0	0	**1**	**10**	0	0	**11**
Total	**9**	**2**	**4**	**86**	**0**	**0**	**101**

- 评估
  - 用实际web应用和测试用例评估了SAFEWAPI，收集了43 个实用Samsung web API的JavaScript web应用并使用SAFEWAPI来检测web API的误用。包括23个智能TV程序和20个安卓手机web应用。
  - 在7个TV应用和9个移动应用中发现了API误用。如表，在实际应用中，发现了其中四种误用。其中最主要的误用是可能引发异常的未处理的API调用（ExnHnd），大约10%检测到的是通过使用不推荐的API函数访问缺失属性。尽管有两个模式没有在已完成开发的软件中发现，但从各种测试用例中发现了它们，所以期望可以在正在开发中的应用中发现

三、各论文内容总结

Are Code Example on an Online Q&A Forum Reliable

- 这篇论文将对stack overflow中API误用的严重性和普遍性进行研究。为减少手动评估工作，设计了examplecheck，一个API 使用挖掘框架，它从github中超过380k个java库中总结模式，并且在stack overflow的帖子中提出潜在的API使用冲突。
- 从100个常用的javaAPI中，根据频率然后删除简单的API，选出了70 个常用API方法，剩下的30个API来自API误用基准MUBench，排除了没有在stack overflow（SO）帖子提出的模式的和无法推广到其他项目的模式。

- API用法挖掘以及模式集
  - ExampleCheck需要三个步骤来推断API用法。
    - 第一阶段，给定感兴趣的API方法，ExampleCheck在GitHub中搜索调用给定的API方法的代码片段，通过程序切片删除不相关的陈述并且提取API调用序列。
    - 第二阶段，ExampleCheck从API调用的各个序列中查找公共子序列。
    - 第三阶段，为保留可调用每个API的条件，examplecheck挖掘与各个调用相关的guard condition。为了准确估计唯一guard condition的频率，examplecheck使用SMT求解器，Z3来检查guard condition的语义等价性，而不是仅仅考虑它们的语义相似性

- API用法挖掘以及模式集
  - GitHub上的结构化调用序列提取与切片
    - 给定一个感兴趣的API方法，examplecheck在github语料库中搜索调用同样方法的单独代码片段。为了将代码搜索拓展到更多的语料库，examplecheck使用一个分布式的软件挖掘基础结构来转换Java文件的抽象语法树AST。
    - 为了提取API用法的本质，examplecheck将每个代码片段建模成结构化调用序列，它抽象出某些语法细节，但依然以紧凑的方式保留API调用的时间顺序、控制结构以及guard condition。
    - examplecheck进行程序内（intra-procedural）程序切片来过滤掉一些与感兴趣的API方法不相关的语句

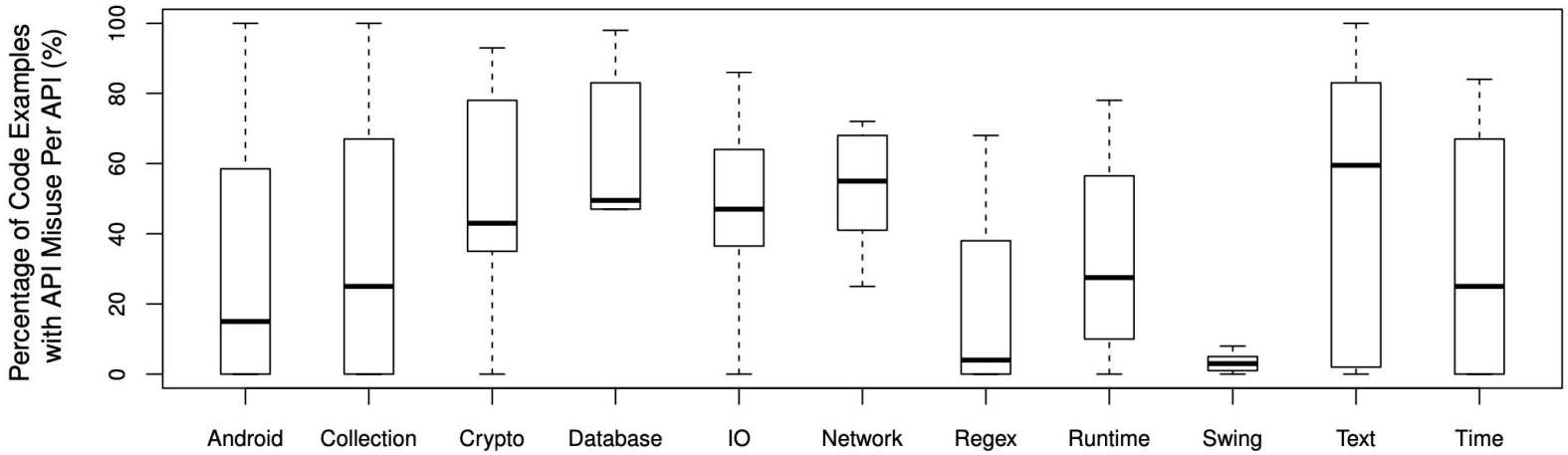
- API用法挖掘以及模式集
  - 频繁子序列挖掘
    - 从阶段1开始，给定一组结构化调用序列，examplecheck使用BIDE算法查找常见子序列。在此阶段，examplecheck主要注重于API调用的时间顺序。
  - Guard conditions挖掘
    - 给定一个来自阶段二的公共子序列，examplecheck挖掘序列中每个API调用的公共的guard conditions。examplecheck从阶段一的每个调用中收集guard conditions，并基于语义等价对它们进行聚类。
    - examplecheck使用SMT求解器（SMT solver）Z3来检查在合并过程中两个guard逻辑上的等价性

人工检查

- examplecheck从研究范围内的100个API中推断出245个API使用模式。这个初始的模式集可能包含无效的或者不准确的模式。因此对推断出的245个模式进行仔细的人工观察，并且基于在线文档和模式频率来排除不准确的模式。
- 在观察过程中，每个模式都被标注为可选（alternative）或必需（required）。一个代码片段必需满足一个可选模式和所有的必需模式。两种模式确保在获得第一个键之前有序映射不为空来避免NoSuchElementException。它们被认为相互替代。
- examplecheck在400个被检查的stack overflow帖子中有289个(72%)含有经两位作者证实的真实的API滥用。在64篇文章中错误地检测到的API的滥用问题。最主要的原因是ExampleCheck通过序列比较来检查API滥用，而没有深入了解它的规范，由于帖子数量限制。有36个是正确的，但是是不常见的替代，ExampleCheck没有学习这些替代的使用模式，因为它们通常不会出现在GitHub上。

三、各论文内容总结

Are Code Example on an Online Q&A Forum Reliable



不同领域API误用比较

Stack overflow中的高赞帖子并不一定可靠

- 将检测到的API使用违规行为分为三类
  - 缺少控制结构
    - 缺少异常处理
    - 缺少if检查
    - 缺少最终处理
  - 缺少或不正确的API调用顺序
  - 不正确的guard condtions
- 局限
  - 仅限于100个经常出现在stack overflow的Java API，因此可能无法推广到其他Java API或不同语言，并且仅限于在stack overflow找到的代码段。
  - 准确率72%，精度不是很高

三、各论文内容总结

Bugram: Bug Detection with N-gram Language Models

A、问题

- 1.第三方API库使用文档不规范，导致API误用。
- 2.API库的误用会导致软件崩溃或者系统脆弱。
- 3.依赖于规则的bug识别方法存在缺陷，忽略不经常使用的API用法。

三、各论文内容总结

Bugram: Bug Detection with N-gram Language Models

B、解决办法

Bugram模型 —— 使用n-gram语言模型检测代码而不是基于规则的方法。通过马尔可夫模型将API方法调用序列进行按顺序统计，常出现的序列被认为是正常的API用法，概率低的用法就是bug或者是不常见的API调用序列。

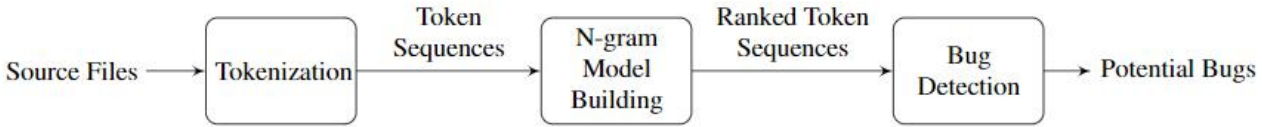


Figure 3: **Overview of Bugram**

本方法假设 —— 程序中的低概率方法调用序列是不正常的，这可能表示检测出了开发人员想知道的bugs，不良代码或对代码的非正常/特殊使用。

三、各论文内容总结

Bugram: Bug Detection with N-gram Language Models

C、具体办法 —— Tokenization

Bugram使用 **Eclipse JDT Core**作为将源码信息token化(ASTs)的工具；

源tokentoken化内容：method calls+ control flow+ 构造器+初始化器；

其中control flow包含以下类型：if/else 分支; for/do/while/foreach 循环; break/continue状态; try/catch/finally模块; return状态; synchronized 模块; switch状态和case/default分支, 以及assert状态.

C、具体办法 —— N-gram Model Building

本论文中使用n元模型学习各序列上的概率分布；对于Tokenization工作中提取的所有序列以及所有的子序列统计进模型中。

例如：一个token序列为ABC，需要将其以及所有子序列(A、B、C、AB、BC)添加进构建模型中。同时忽略不连续的子序列比如(AC).

本文针对n元模型中n的取值不同对于bug检测结果的影响开展了探究，其中n值取为2-10。

C、具体办法 ——Bug Detection

对于n元模型中获得的序列概率进行排名，最低的概率即为潜在的bugs。

影响因素：

1.Gram Size **n**：即为n元模型中n的值，本文中取2-10进行研究；

2.Sequence Length **l**：即为本文第一节中token序列的长度，通过不同长度对结果的影响进行评估；

C、具体办法 ——Bug Detection

影响因素：

3.Reporting Size **s**：概率排名最低的s个序列为潜在bugs；

4.Minimum Token Occurrence y：软件中token至少出现y次才可以被放进n元模型中，避免排名出现失误。通过[26]中技术过滤。

```
String q[] = qqf.bestQueries("body",20);
  for (int i=0; i<q.length; i++) {
       System.out.println(newline+
       formatQueryAsTrecTopic(i,q[i],null,null));
}
```

Figure 4: The filtering based on Minimum Token Occurrence can help Bugram avoid reporting this false bug from the latest version of Lucene.

序列[bestQueries,println,formatQueryAsTrecTopic]排名较低但却不是bug；原因是bestQueries以及formatQueryAsTrecTopic两个token在软件中仅出现一次。

C、具体办法 ——Bug Detection

修改错误预测：

因为最终的预测结果为真正的bug以及频率较低的序列，需要通过**保留在不同sequence lengths长度预测中均为低概率排名序列**，将其中低频率序列过滤出来。

例如：如果ABCDE和BCD分别位于5tokens序列和3tokens序列均在最低排名列表中，Bugram就将它们标识为重叠（两个序列包含一个公共子字符串BCD），并报告ABCDE和BCD有一个bug。

$$C(n,t) = \bigcup_{\forall i,j \in M, i \neq j} (Bottom(n,t,i) \cap Bottom(n,t,j)) \quad (4)$$

三、各论文内容总结

Bugram: Bug Detection with N-gram Language Models

D、评估实验——数据集

16个广泛使用的开源java项目

Table 1: **Projects evaluated in our experiments**

Project	Version	Files	LOC	Methods
Elasticsearch	1.4	3,130	272,261	28,950
GeoTools	13-RC1	9,666	996,800	89,505
jEdit	5.2.0	543	110,744	5,548
Proguard	5.2	675	69,376	5,919
Vuze	5500	3,514	586,510	37,939
Xalan	2.7.2	907	165,248	8,965
Hadoop	2.7.1	4,307	596,462	46,104
Hbase	1.1.1	1,392	465,456	42,948
Pig	0.15.0	948	121,457	9,323
Solr-core	5.2.1	1,061	146,749	9,938
Lucene	5.2.1	2,065	293,825	18,078
Opennlp	1.6.0	603	36,328	2,954
Struts	2.3.24	2,022	157,499	15,254
Zookeeper	3.5.0	492	61,708	5,034
Nutch	2.3.1	409	198,560	2,309
Cassandra	2.2.0	1,616	280,716	15,233

三、各论文内容总结

Bugram: Bug Detection with N-gram Language Models

D、评估实验 ——参数实验

三个有代表性的项目：**Pig,Hadoop, 以及 Solr**

测试九种不同gram sizes，九种不同的sequencelengths，五种不同的reporting sizes。

1.gram sizes

结果显示使用3元模型中发现的bug最多，之后的实验通过3元图模型实现。

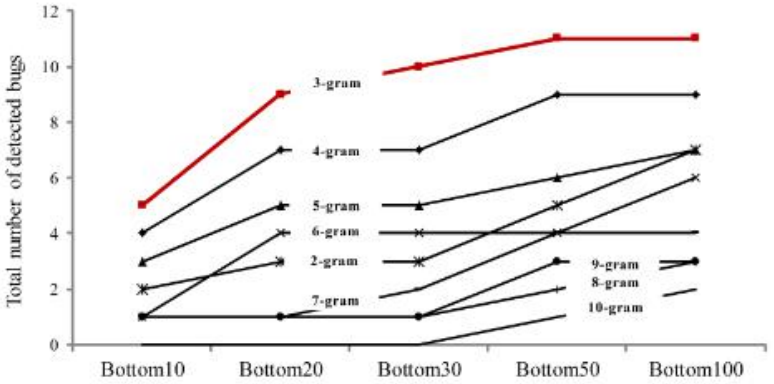


Figure 5: **Impact of the gram size on the number of true bugs detected**

D、评估实验——参数实验

2.Setting Sequence Length

使用3元模型不同的sequence长度，检查概率较低的前50个序列，结果显示当序列长度很短（例如两个）或很大（例如九个和十个）时，Bugram在三个检查的项目中有两个没有检测到错误。在本文中，序列长度范围从3到8。

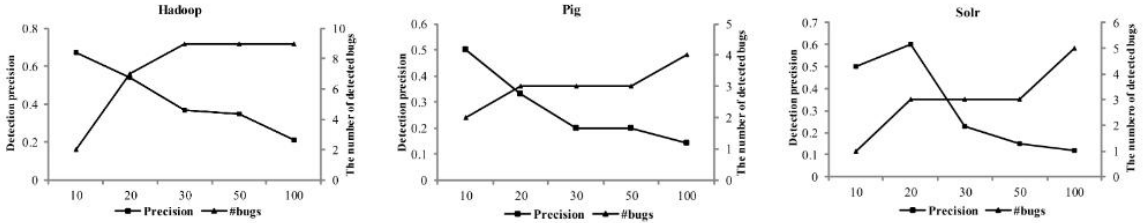
Table 2: **Detected true bugs in the bottom 50 token sequences with different sequence lengths**

Project	Sequence Length								
	2	3	4	5	6	7	8	9	10
Pig	0	1	1	3	2	3	1	1	1
Hadoop	0	3	4	7	3	3	2	0	0
Solr	0	1	1	1	2	0	0	0	0

D、评估实验——参数实验

3.Setting Reporting Size

使用3元模型，序列长度范围设置为3到8。结果显示reporting size设置为20时检测精度最高。



(a) Results of Hadoop (b) Results of Pig (c) Results of Solr

Figure 7: Detection precision and number of detected bugs in the overlaps of bottom s token sequences with low probability

4.Setting Minimum Token Occurrence

根据以往经验取值为3，即删除在软件中出现少于三次的tokens。

三、各论文内容总结

Bugram: Bug Detection with N-gram Language Models

D、评估实验——模型比较及结果

1.Comparison with FSM and FIM

将本文模型Bugram与FIM和FSM对比，其中FIM即为原始PR-Miner，FSM为考虑token顺序和控制流之后的改进PR-miner。

结果:

Bugram的recall率为54.5%，precision为71.2%，F1为61.7%。

FSM的recall率为26.0%，precision为29.9%，F1为27.8%。

FIM的recall率为40.3%，precision为2.4%，F1为4.5%。

Table 3: **Bug detection results. Reported is the number of reported bugs, TBbugs is the number of true bugs, and Refs is the number of refactoring opportunities. We manually inspect all reported bugs except for FIM whose 'Inspected' column shows the number of bugs inspected.** ***Numbers in brackets are the numbers of true bugs detected by Bugram that are detected by neither FIM nor FSM.***

Project	Bugram			FSM			FIM			
	Reported	TBugs	Refs	Reported	TBugs	Refs	Reported	Inspected	TBugs	Refs
Elasticsearch	3	1(1)	0	0	0	0	987	80	0	2
GeoTools	4	2(2)	1	4	0	2	1,203	80	1	2
JEdit	3	0	1	0	0	0	451	80	0	2
Proguard	1	1(1)	0	1	1	0	665	80	1	0
Vuze	2	0	2	0	0	0	435	80	0	4
Xalan	3	2(2)	0	0	0	0	378	80	0	0
Hadoop	13	7(6)	4	13	3	0	869	80	2	3
Hbase	1	1(1)	0	10	2	3	774	80	1	0
Pig	9	3(2)	4	6	0	2	605	80	1	3
Solr-core	5	3(3)	1	0	0	0	787	80	0	1
Lucene	2	0	0	10	2	2	676	80	1	0
Opennlp	6	2(2)	2	3	0	0	806	80	0	2
Struts	5	1(1)	2	9	1	0	232	80	0	0
Zookeeper	0	0	0	3	0	0	442	80	0	1
Nutch	2	2(2)	0	1	0	1	253	80	1	1
Cassandra	0	0	0	7	0	1	324	80	0	2
Total	**59**	**25(23)**	**17**	**67**	**9**	**11**	**9,887**	**1,280**	**8**	**23**
Relative Recall	**54.5%**			**26.0%**			**40.3%**			
Precision	**71.2%**			**29.9%**			**2.4%**			
F1	**61.7%**			**27.8%**			**4.5%**			

三、各论文内容总结

Bugram: Bug Detection with N-gram Language Models

D、评估实验——模型比较及结果

2.Comparison with JADET，Tikanga，and GrouMiner

将本文模型Bugram与FIM和FSM对比，其中FIM即为原始PR-Miner，FSM为考虑token顺序和控制流之后的改进PR-miner。

结果:

Bugram检测精度优于三个工具

Table 4: **Comparison with JADET, Tikanga, and GrouMiner. 'Fixed' denotes the number of true bugs detected by Bugram that have already been fixed in later versions. * *denotes the number of unique true bugs detected by Bugram that the tools in comparison failed to detect.***

Project	Graph-based tools	Bugram		
	JADET	Reported	TBugs	Fixed
AZUREUS 2.5.0	1	8	4	4
columba-1.2	0	4	1*	1
aspectj-1.5.3	2	5	0	0
	3	17	5	5
Project	Tikanga	Bugram		
aspectj-1.5.3	9	5	0	0
tomcat-6.0.18	0	13	4*	2
argouml-0.26	1	3	1	1
Vuze_3.1.1.0	0	8	0	0
columba-1.4	1	6	1	1
	11	35	6	4
Project	GrouMiner	Bugram		
columba-1.4	1	6	1	1
ant-1.7.1	1	3	1	1
log4j-1.2.15	0	7	2*	2
aspectjrt-1.6.3	1	12	2	1
axis-1.1	0	6	3*	1
jedit-3.0	1	5	1	1
jigsaw-2.0.5	1	1	1	1
struts-1.2.6	0	8	0	0
	5	48	11	8

E、结论及未来工作

Bugram通过基于项目中程序token序列的概率分布来计算和排序来检测潜在的错误。通过两种方式评估Bugramin。 首先，将其与针对16个项目的两种基于规则的错误检测方法进行比较。 结果表明，Bugram可检测到25个真实的错误，而PR-Miner无法检测到其中的23个错误。其次，将Bugram应用于在基于图和基于规则的工具（即JADET，Tikanga和GrouMiner）中评估的14个项目中。 Bugram检测出的21个错误，这三个工具无法检测到至少10个。 结果表明Bugram是对现有基于规则的错误检测方法的补充。

将来，计划从多个项目构建n-gram模型以执行跨项目错误检测，这可能有助于更准确地发现更多错误。 还计划在构建n-gram模型时探索分割序列的不同方法。当前，Bugram将序列从一种方法分解为固定长度的序列。 此外，将Bugram扩展到C / C ++项目，并结合Bugram和基于规则的方法来检测更多错误。

三、各论文内容总结

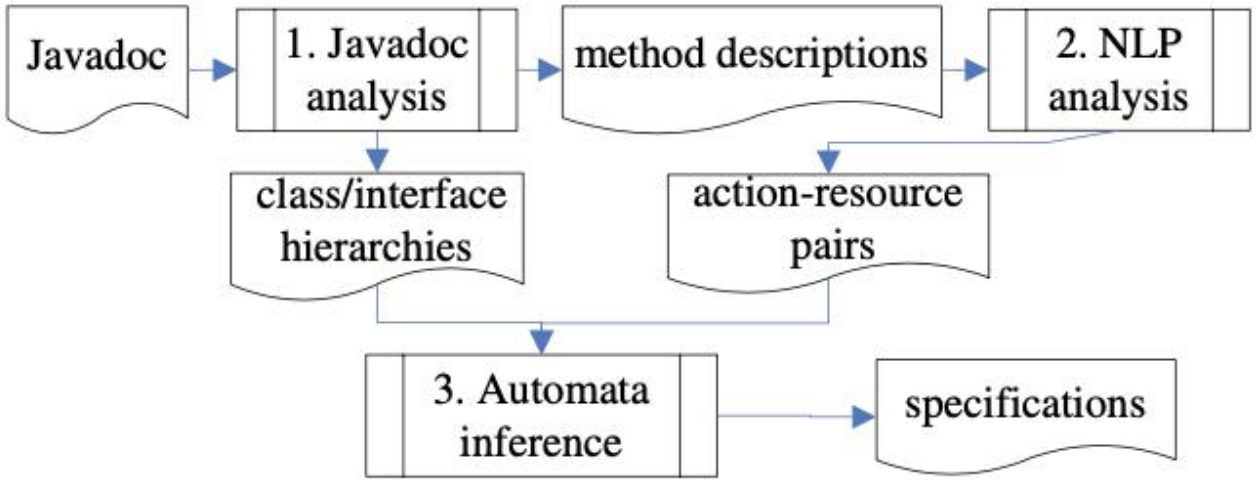
Inferring Resource Specifications from Natural Language API Documentation

- 这篇论文提出了一种称为Doc2Spec的方法，它从API文档推断使用规范。对该方法实现了一个工具，并对五个库的Javadoc进行了评估。
- 可行性
  - 由于API文档包含大量关于资源使用的信息，因此可以从API文档中推断资源规范
- 挑战性
  - (1)由于API文档是用自然语言编写的，因此需要进行准确的语言分析;
  - (2)由于资源的使用通常隐含在对多种方法的描述中，因此需要综合来自多方法描述的信息。

- 为什么基于API文档
  - 之前的大多数方法使用现有代码作为输入，当一个库不流行或者不新颖时，它的相关代码很难找到，那么随机生成的测试用例可能不能反映库的实际使用。而基于API不需要代码，只要API文档中具有方法的相关描述，就能够推断出规范，从而补充了前面的方法

- 过程
  - 1、推断规范
    - 第一步是从API文档中提取方法描述和类/接口层次结构
    - 第二步是根据每个方法描述构建一个操作资源对
    - 最后一步是基于操作资源对和类/接口层次来构建推断资源的自动机
  - 2、检测bug

- 方法步骤图


Javadoc
1. Javadoc analysis
method descriptions
2. NLP analysis
class/interface hierarchies
action-resource pairs
3. Automata inference
specifications

- Javadoc分析：忽略没有描述的方法
- NLP分析
  - 使用基于隐马尔可夫模型（HMM）的命名实体识别（NRE）实现action和resource。
  - 首先使用Baum-Welch算法从人工标记的方法描述中训练参数
  - 在训练之后，我们的方法使用Viterbi算法根据训练模型用分数标记方法描述
  - 最后选择得分最高的action和resource来构建方法的动作资源对

- 自动机推断
  - 1、将由类/接口或类/接口的超类/超接口声明的方法归为一类，在每个类别中，方法的资源具有相同的名称
  - 2、根据资源名称和接口继承将这些方法归类
  - 3、根据操作将每个类别中的方法进一步映射到我们预定义的类型中。预定义了五种类型的方法：创建，锁定，操作，解锁和关闭
  - 4、基于预定义的规范模板为每个类别构建一个自动机。

- 规范模版

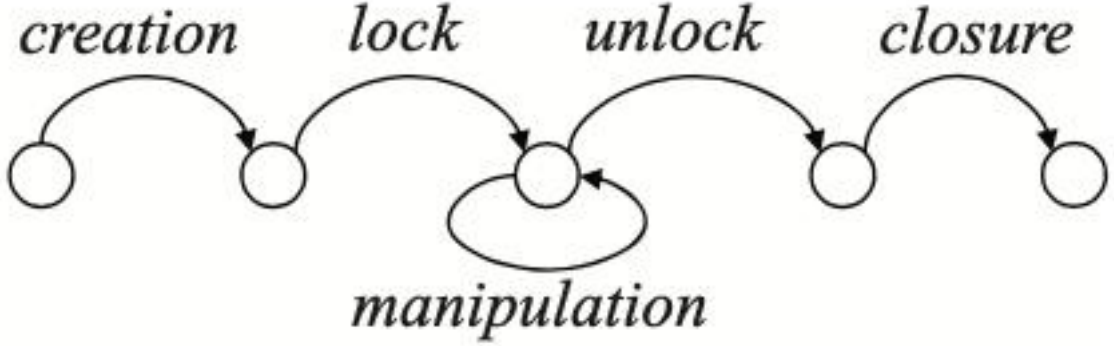


Figure 1. Specification template

- 表现

PERFORMANCE OF DOC2SPEC

Lib	Version	#M	#D	#Spec	DT	ST
J2SE	5.0	25675	23829	3250	400.2	2357.2
J2EE	5.0	5670	5611	83	95.7	151.4
JBoss	4.0.5	26053	13869	373	430.4	140.0
iText	2.1.3	5846	4299	243	112.2	52.4
Oracle	10.1.0.5	2140	1916	32	28.3	22.4
Total		65384	49524	3981	1067.0	2723.4

- Lib：库的名称。
- ＃M：方法数量。
- #D：带有说明的方法数量。
- #Spec：推断出的资源规范的数量。
- DT”：用于提取方法描述和类/接口层次结构的时间（以秒为单位）
- ST：基于提取的信息来推断规格的时间（以秒为单位）。
- Total：总计

三、各论文内容总结

Inferring Resource Specifications from Natural Language API Documentation

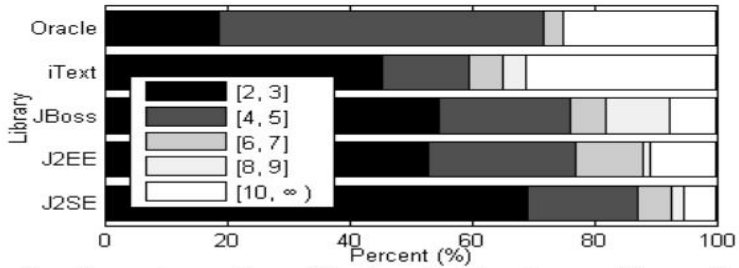


Figure 7. Percentages of specifications that involve specific numbers of methods

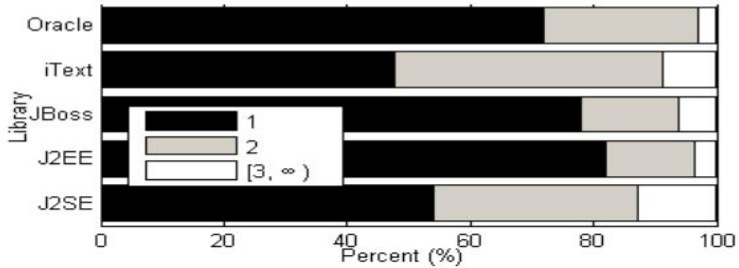


Figure 8. Percentages of specifications that involve specific numbers of classes/interfaces

- 总的来说，尽管大多数推断出的规范只涉及到一个或两个类/接口和少于五个方法，方法能够推断出各种复杂的规范。

PRECISIONS, RECALLS, AND F-SCORES OF INFERRED SPECIFICATIONS

Library	#H	#S	Precision	Recall	F-score
J2SE	8	41	80.2%	82.2%	81.2%
J2EE	7	30	70.7%	79.3%	74.8%
JBoss	8	37	81.5%	74.0%	77.6%
iText	6	22	86.5%	85.2%	85.8%
Oracle	5	17	82.3%	86.2%	84.2%

方法在所有库上都取得了较高的精确度、召回率和F-Score

- 局限：
  - 仅对有限库的javadoc进行了评估
  - False positive率不理想
  - 受API文档影响
- 优势
  - 不需要大量的代码，也不受代码使用频率等的影响，只需要API文档，补充了之前的方法
  - 准确度，召回率以及F-Score表现良好

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

- 该文章主要工作是改善了API误用检测的方法，提出了一种名为MUDETECT的API误用检测器。该检测器基于现有检测器的优势，使用API Usage的图表示法来捕获不同类型的API误用。
- 同时，本文还提出了一种可以有效提高准确性的排名策略，可针对检测器检测到的真实正例进行有效排名。

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

- 先前各类研究提出的API误用检测器准确率和召回率较低的原因，以及本文策略进行相应改进的部分：

1. Representation（本文的表示结合了对控制流、异常流、方法调用顺序、同步和数据流的跟踪）
2. Matching（MUDETECT匹配调用时考虑类型层级信息，使用排名策略而不是距离阈值）
3. Uncommon Usages（MUDETECT删除模式中纯方法的调用，除非使用其返回值）
4. Alternative Patterns（MUDETECT对可替换的模式进行过滤）
5. Self- ans Cross-method Usages（MUDETECT忽略自身和字段上的使用）
6. Ranking
7. Usage Examples（在单项目和提供更多训练示例的跨项目中进行评估）

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（1）API用法图

本文使用AUGs图（带有标记结点和边的有向连通图）作为API用法的表示方法：

- 结点：数据实体（如变量）和操作（如方法调用）
- 边：实体和操作之间的控制和数据流

组件分类：

- Usage Actions（方形结点）：表示方法调用、操作符和API用法中的指令
- Data Entities（椭圆形结点）：表示对象、值和字面值
- Control Flow and Data Flow：有receiver、parameter、definition、order、condition、synchronize、throw和handle八种类型的边，每种边用各自类型的缩写标记。

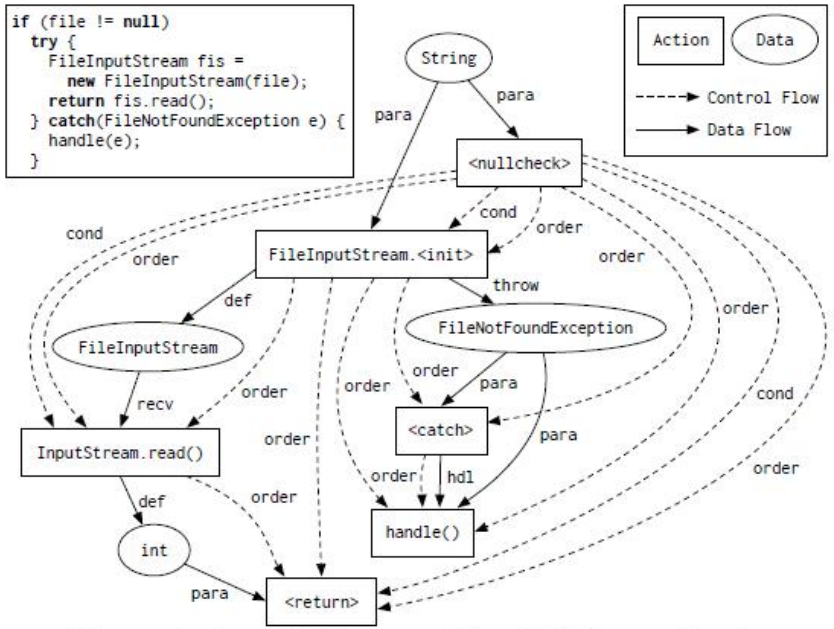


Figure 1: An API Usage and its API-Usage Graph.

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（2）模式匹配

- 本文提出的模式挖掘算法使用了AUGs的集合A、一个频率度量f和一个频率阈值σ。生成的模式是在A中频繁出现的sub-AUG。该算法的三个关键思想：

1. 基于apriori算法挖掘频繁子图
2. 基于代码语义进行扩展
3. 贪心的搜索策略

```
1  def mine(A: Set[AUG], f: Pattern → int, σ: int)
2    P_0 = {p | p ∈ single_call_patterns(A) and f(p) ≥ σ}, P = ∅
3    for p in P_0: extend(p, P, f, σ)
4    return P
5
6  def extend(p: Pattern, P: Set[Pattern],
7      f: Pattern → int, σ: int)
8    E = {e | i ∈ p and e ∈ generate_extensions(i)}
9    PC = {c | c ∈ isomorphic_clusters(E) and f(c) ≥ σ}
10   UC = PC \ P
11   if UC ≠ ∅:
12     c = most_frequent(UC)
13     extend(c, P, σ)
14   else: P = P ∪ {p}
15   ip = {i | i ∈ p and ∀c ∈ PC. generate_extensions(i) ∩ c = ∅}
16   if f(ip) ≥ σ: P = P ∪ {ip}
17
18 def generate_extensions(i: Instance)
19   extensions = ∅
20   for n in adjacent_nodes(i):
21     if has_non-order_connection(n, i) and (
22         is_non-pure_call(n)
23         or (is_pure_call(n) and has_out_connection(n, i))
24         or (is_operator(n) and has_in&out_connection(n, i))
25         or (is_data(n) and has_out_connection(n, i))):
26       extensions = extensions ∪ {i ⊕ n}
27   return extensions
```

Listing 1: MuDetect's Pattern-Mining Algorithm

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（3）违反检测

- 本文提出的检测算法使用了目标AUGs的集合T、模式的集合P和排名函数r。生成的是违反列表，其中每一个违反都是模式的严格子图。该算法包含四个步骤：

1. Graph Matching（图匹配）
2. Alternative-Pattern Instances（提取可选模式实例）
3. Violation Ranking（违反排名算法）
4. Alternative Violations（提取可选的违反）

```
def find_violations(T: Set[AUG], P: Set[Pattern],
    r: (Set[Violation], Set[Instance], Set[Pattern]) →
        List[Violation]):
  V = ∅, I = ∅
  for target in T:
    for pattern in P:
      for overlap in common_subgraphs(target, pattern):
        if overlap = pattern:
          I = I ∪ {Instance(target, pattern)}
        elif overlap ⊏ pattern:
          V = V ∪ {Violation(target, pattern, overlap)}
  V_A = filter_alternatives(V, I)
  V_R = r(V_A, I, P)
  return filter_alternative_violations(V_R)

def common_subgraphs(t: AUG, p: Pattern):
  S = single_call_node_overlaps(t, p)
  return {lcs | lcs ∈ extend_subgraph(s, t, p) and s ∈ S}

def extend_subgraph(o: Overlap, t: AUG, p: Pattern)
  e = next_extension_edge(o, t, p)
  return extend_subgraph((o ⊕ e), t, p) if e ≠ none else o

def next_extension_edge(o: Overlap, t: AUG, p: Pattern):
  e_best = none, w_min = inf
  for e in adjacent_edges(o, t):
    w = count_equiv_edges(e, t) * count_equiv_edges(e, p)
    if 0 < w and w < w_min:
      e_best = e, w_min = w
  return e_best
```

Listing 2: MUDETECT's Detection Algorithm

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（4）排名策略

- 文章首先调查了现有的排名策略并讨论各因素，然后根据这些因素组成了新的排名策略。
- 1）以前的排名策略：
  - 一些检测器使用模式与用法之间的最大距离来将用法分类为违规
  - 一些检测器通过违反模式（ps）的支持对他们的发现进行排名
  - 一些检测器将它们的发现按其稀有性排序，将ps和违规再次发生的次数（即违规支持（vs））组合为（ps -vs）/ ps
- 2）MUDETECT的排名策略：从各文献中考虑排名策略的候选者，然后通过惩罚考虑各个排名因子的所有组合。

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（5）单项目和跨项目的设置

- MUDETECT分离了模式挖掘和违反检测两步，使得可以在不同的设置上分布运行它。
- （1）在单项目的设置中，使用来自其目标项目的AUGs作为模式挖掘和违反检测的共同输入；
- （2）在跨项目的设置中，使用来自其目标项目的AUGs作为违反检测的输入，而使用来自其他项目的AUGs作为模式挖掘的输入。

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

- 比较了MUDetect和其他四个检测器JADET、GROUMINER、TIKANGA和DMMC。使用的数据集来自于MUBench，并对该数据集进行了扩展。

Table I: MUBENCH: Number of Misuses (#MU) and Number of Misuses with Corresponding Correct Usages (#CU).

Dataset	#MU	#CU
Original MUBENCH [18]	84	64
Our Extension	107	107
Extended MUBENCH	191	171

- 使用MUBenchPIPE进行了五种实验：
- Experiment P.目的是测量探测器的精度
- Experiment RUB. 目标是在给定完美训练数据的情况下评估探测器的召回上限
- Experiment R.目的是测量探测器的召回率
- Experiment RNK.目标是找到MUDETECT的最佳排名策略
- Experiment XP.目标是测量MUDETECTXP的精度和召回率（在跨项目设置中）

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（1）实验RNK

- 表II列出了最佳和最差排名策略，这些策略按排名前20位的发现中的命中次数（@ 20），命中次数（#H）和平均命中排名（AHR）进行排序
- 结果表明，排名对MUDETECT如何排名误用有很大影响

Table II: Experiment RNK: Number of Hits (#H), Average Hit Rank (AHR), and Number of Hits in the Top-20 (@20).

#	Strategy	@20	#H	AHR	#	Strategy	@20	#H	AHR
1.	$p_s/v_s \times v_o$	19	34	91.6		...			
2.	p_s/v_s	17	34	91.8	9.	p_s	14	34	305.5
3.	$p_s/v_s \times v_o \times p_v$	16	34	90.1		...			
4.	Rareness	16	33	94.3	33.	p_u/v_s	2	18	53.2
	...				34.	v_o	1	26	1187.4

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（2）实验P、RUB和R

- 实验P的结果表明，从实验RNK中识别出的排序策略成功地将真正例推到最前面，使我们的性能优于其他检测器。仍存在误报的原因：
  - FP1：uncommon usages
  - FP2：Intra-procedural Analysis
- 实验RUB的结果表明，AUG捕获正确用法和误用之间的差异要比所有其他检测器更好，并且本文的检测算法成功地识别了这些差异。仍存在误报的原因：
  - FN1：Self-Usages
  - FN2：Redundant
- 实验R的结果表明，MUDETECT的召回率高达20.9%，几乎是其他探测器召回率的两倍。

Table III: Results: Experiment P measures precision in the top-20 findings. Experiment RUB measures recall upper bound. Experiment R measures recall. Experiment XP measure precision and recall of MUDETECTXP.

	Experiment P			Experiment RUB				Experiment R		
Detector	Confirmed Misuses	Precision	Kappa Score	Hits	Recall Upper Bound		Kappa Score	Hits	Recall	Kappa Score
JADET	8	8.8%	0.64	29	16.9%		0.79	15	6.7%	0.64
GROUMINER	4	2.6%	0.49	88	51.2%		0.85	7	3.1%	1.00
TIKANGA	7	8.2%	0.52	15	8.8%		0.73	17	7.6%	0.69
DMMC	12	7.5%	0.72	28	16.3%		0.88	24	10.7%	0.91
MUDETECT	32	21.9%	0.90	124	72.5%		0.89	47	20.9%	1.00
MUDETECTXP	30	33.0%	0.88					95	42.2%	0.93

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

（3）实验XP

- 表III的最后一行显示了MUDETECTXP与实验P和实验R中其他检测器相比的精度和召回率。MUDETECTXP可以识别MUDETECT遗漏的65种误用。
- 实验XP的结果表明，来自其他项目的挖掘模式显著提高了准确性和查全率

Table III: Results: Experiment P measures precision in the top-20 findings. Experiment RUB measures recall upper bound. Experiment R measures recall. Experiment XP measure precision and recall of MUDETECTXP.

	Experiment P			Experiment RUB			Experiment R		
Detector	Confirmed Misuses	Precision	Kappa Score	Hits	Recall Upper Bound	Kappa Score	Hits	Recall	Kappa Score
JADET	8	8.8%	0.64	29	16.9%	0.79	15	6.7%	0.64
GROUMINER	4	2.6%	0.49	88	51.2%	0.85	7	3.1%	1.00
TIKANGA	7	8.2%	0.52	15	8.8%	0.73	17	7.6%	0.69
DMMC	12	7.5%	0.72	28	16.3%	0.88	24	10.7%	0.91
MUDETECT	32	21.9%	0.90	124	72.5%	0.89	47	20.9%	1.00
MUDETECTXP	30	33.0%	0.88				95	42.2%	0.93

三、各论文内容总结

Investigating Next Steps in Static API-Misuse Detection

可能影响有效性的因素：

- **过度拟合。**文章根据在MUBENCH上的研究观察结果设计了MUDETECT，又在相同基准上评估了MUDETECT，因此扩展了基准数据集。
- **内在有效性。**文章使用其他检测器进行评估对比时直接使用了相关文献中的配置而没有进行调整。此外，对于跨项目的设置，仅仅评估了MUDETECTXP。
- **外部有效性。**使用的评估API误用数据集可能不具有代表性，因此使用了MUBENCH。

三、各论文内容总结

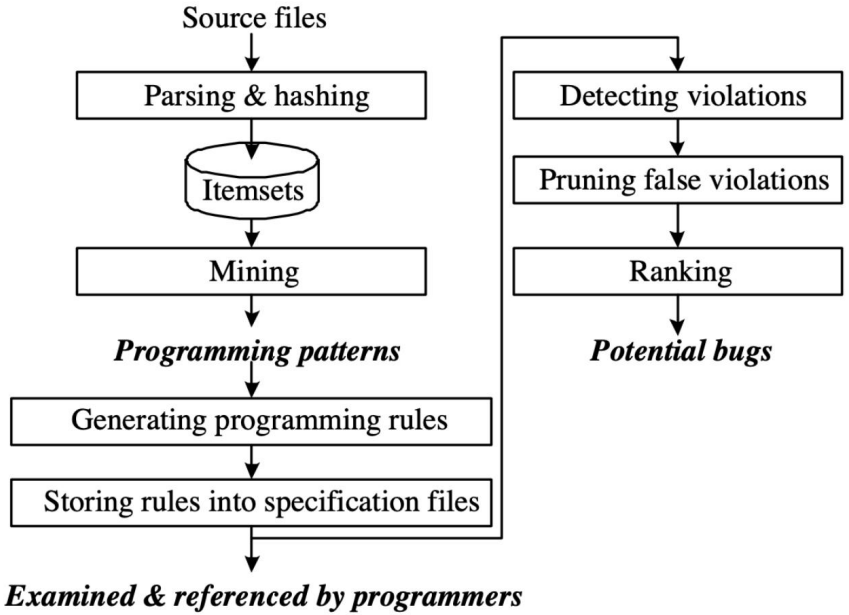
PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- PR-Miner使用一频繁项集挖掘，提取一般形式的编程规则，并且可以自动检测对提取的编程规则的违反情况
  - 不受任何固定规则模板的约束
  - 包含各种类型的多个程序元素，例如函数，变量和数据类型
- (1)提出了一种从大型软件代码中自动提取隐式编程规则的通用方法
- (2)提出了一种高效的算法来检测对所提取的编程规则的违反情况
- PR-Miner有两个主要的功能:自动提取隐式编程规则，以及自动检测对提取的编程规则的违反情况

三、各论文内容总结

PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- 流程图



三、各论文内容总结

PR-Miner：Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- 自动提取规则
  - 在自动规则提取中，PR-Miner的高级思想是通过查找源代码中经常一起使用的元素来查找元素之间的关联（使用GCC编译器）
  - 为了有效地找到程序元素的相关性，PR- Miner首先解析软件源代码，将问题转换为频繁项集挖掘问题。对于挖掘算法发现的频繁子项集，我们将对应的程序元素集合称为一个编程模式（programming pattern）
  - 在使用频繁项集挖掘技术提取编程模式之后，PR-Miner需要从提取的模式中生成编程规则
  - 在生成编程规则之后，PR-Miner将它们存储在规范文件中，以便程序员可以检查它们，并在以后将它们用作引用

三、各论文内容总结

PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- 使用闭频繁项集挖掘算法FPclose：
  - (1)通用性。闭频繁项集挖掘算法不限制频繁子项集中的项数，也不需要任何规则模板。
  - (2)时间效率。像FPclose这样的数据挖掘算法通常是非常高效的，因为它们尽量消除冗余计算，以避免扫描数据太多
  - 3)空间效率。从封闭频繁子项集中可以推出多种封闭规则（closed rule）

三、各论文内容总结

PR-Miner：Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- 为了进一步减少输出规则的数量并加快生成和违规检测过程，PR-Miner以压缩格式存储封闭规则
- 闭合频繁子项目集I的压缩格式为:

$$I : s|\{C_1 : s_1|s_1 > s\} \ldots \{C_m : s_m|s_m > s\}$$

- 提出CLOSEDRULES 算法生成压缩格式类型的封闭规则

三、各论文内容总结

PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

Algorithm: CLOSEDRULES($\mathcal{I}$)
Input: $\mathcal{I} = \{I_k | 1 \leq k \leq n\}$,
I_k has 3 fields $\langle F_k, s_k, E_k \rangle$;
Output: The closed rules $\mathcal{R}$ in condensed format.
1: Sort $\mathcal{I}$ by supports in descending order such that
$s_1 \geq s_2 \geq ... \geq s_n$
2: Mine common closed frequent sub-itemsets from $\mathcal{I}$:
$\mathcal{C} \leftarrow$ FPCLOSE($\{F_i | i = 1, 2, ..., n\}, 2$),
where $\mathcal{C} = \{C_i | 1 \leq i \leq m\}$ and
C_i has 3 fields $\langle F'_i, s'_i, E'_i \rangle$
3: **for** $i = 1, 2, ..., m$
4: Denote $E'_i = \{i_j | 1 \leq j \leq s'_i\}$
5: **for** $j = 2, 3, ..., s'_i$
6: **if** $s_{i_1} > s_{i_j}$
7: Insert $F'_i : s_{i_1}$ to sub-itemset I_{i_j} in $\mathcal{R}$

- 检测对提取所得规则的违反情况
  - 检测对提取的编程规则的违反情况
  - 对检测出错的违规情况进行修改
    - 通过执行一个过程间检查。通过检查每个包含违规的函数的调用和被调用者的路径来减少false positive
  - 对违规进行排名。

三、各论文内容总结

PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

用Linux、PostgreSQL和Apache HTTP服务器对PR-Miner进行评估

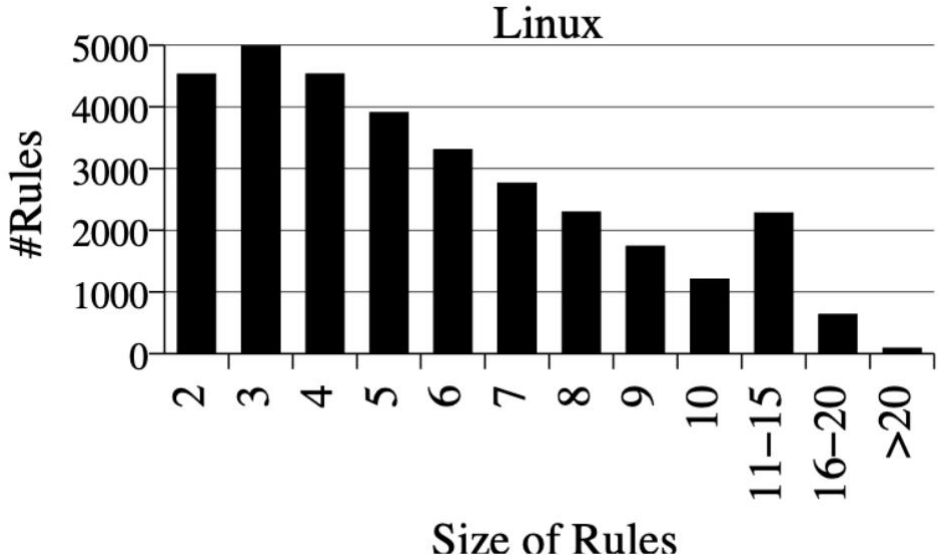
Software	version	#C files	LOC	#functions
Linux	2.6.11	3,538	3,037,403	73,607
PostgreSQL	8.0.1	409	381,192	6,964
Apache	2.0.53	160	84,724	1,912

Software	Total	F-F	V-V	F-V
Linux	32,283	1,075	8,883	22,325
PostgreSQL	6,128	379	687	5,062
Apache	283	33	92	158

封闭规则可以分为三类:函数-函数(F-F)规则、变量-变量(V-V)规则和函数-变量(F-V)规则

三、各论文内容总结

PR-Miner：Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code



说明PR-Miner没有限制规则中的元素数量

三、各论文内容总结

PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- 检测结果

Software	Inspected (top 60)			Uninspected
	Bugs	Specification	False Positives	
Linux	16	20	24	1387
PostgreSQL	6	9	45	87
Apache	1	0	6	0

- 开销

Software	Extracting rules		Detecting violations	
	Time(s)	Space(MB)	Time(s)	Space(MB)
Linux	42	441	46	303
PostgreSQL	5	25	4	14
Apache	1	7.3	1	6.2

三、各论文内容总结

PR-Miner：Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- 局限
  - 对复制粘贴造成的影响没有很好的进行处理。由于复制粘贴，对编程规则的违反可能会传播到多个模块，这将导致对规则的大量违反。
  - C中的宏定义可能会导致错误的编程规则以及检测违规时的错误错误。由于GCC首先通过在创建中间表示之前展开宏来对源代码进行预处理，所以一个宏中的信息可以像复制粘贴一样复制很多次。
  - 函数名称冲突
  - 不能检测跨越多个函数定义的编程规则
  - 控制语句中的违规情况可能会遗漏

三、各论文内容总结

PR-Miner：Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code

- PR-Miner是一种有效的、实用的工具，可以提取隐含的、无文档的编程规则，并检测大型软件代码中的违规行为。此外，通过替换GCC前端解析器，PR Miner可以很容易地应用于其他编程语言（如Java）中的程序

三、各论文内容总结

Code2seq: generating sequences from structured representations of code

A、问题及方案

为了解决有结构的代码段与自然语言描述之间的关系，本文推出CODE2SEQ模型——一种利用编程语言的语法结构来更好地编码源代码的方法，是一种最新的ＮＭＴ模型。

模型可用于对**代码摘要**、**代码序列化文本**和**代码评论**。

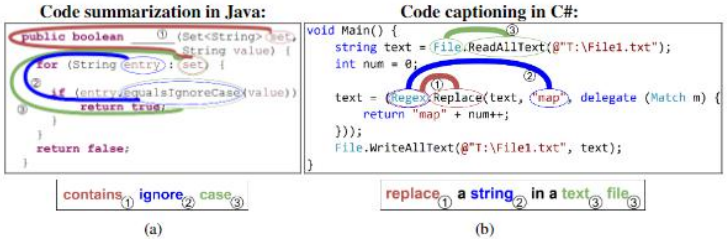


Figure 1: Example of (a) code summarization of a Java code snippet, and (b) code captioning of a C# code snippet, along with the predictions produced by our models. The highlighted paths in each example are the top-attended paths in each decoding step. Because of space limitations we included only the top-attended path for each decoding step, but hundreds of paths are attended at each step. Additional examples are presented in Appendix B and Appendix C.

三、各论文内容总结

Code2seq: generating sequences from structured representations of code

B、REPRESENTING CODE AS AST PATHS

AST的叶节点代表标识符和变量名，非叶节点代表语言中结构的限制集，比如循环，变量声明，正则式。
论文将成对叶节点之间的路径当做包含路径上所有节点的sequence，然后用这些带有叶节点的路径表示代码片段本身。将代码片段当做token的sequence则会忽略掉体现方法相似的语法路径。

右图中红色路径即为相同的token序列表示不同的源码。

一个AST的路径可以是任意个，本文取值 k 为 **200**

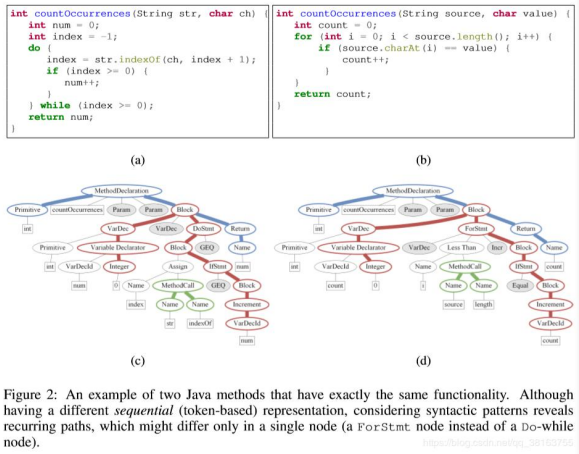


Figure 2: An example of two Java methods that have exactly the same functionality. Although having a different *sequential* (token-based) representation, considering syntactic patterns reveals recurring paths, which might differ only in a single node (a `ForStmt` node instead of a `Do`-while node).

三、各论文内容总结

Code2seq: generating sequences from structured representations of code

C、模型框架——ENCODER-DECODER FRAMEWORK

采用经典NMT模型，解码层选用LSTM为主要单元。

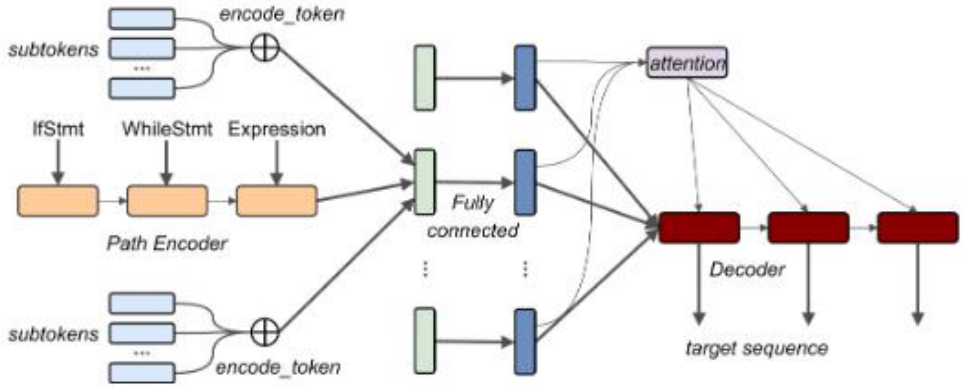


Figure 3: Our model encodes each AST path with its values as a vector, and uses the average of all of the k paths as the decoder's start state. The decoder generates an output sequence while attending over the k encoded paths.

三、各论文内容总结

Code2seq: generating sequences from structured representations of code

C、模型框架——AST Encoder

Path Representation：一段源代码AST中的各个路径都是由节点以及子节点组成，使用可学习嵌入矩阵 E^{nodes} 代替路径中的节点，使用bi-LSTM获得整个路径序列的编码。

$$h_1, ..., h_l = LSTM(E^{nodes}_{v_1}, ..., E^{nodes}_{v_l})$$
$$encode\_path(v_1 ... v_l) = [\overrightarrow{h_l}; \overleftarrow{h_1}]$$

Token Representation：AST路径中第一以及最后一个节点是值为token的终端节点。使用$E^{subtokens}$ 代表每个token的子token（通过分割获取）。可以使用LSTM模型在每个时间步预测子token（当预测方法名时）

$$encode\_token(w) = \sum_{s \in split(w)} E^{subtokens}_s$$

C、模型框架——AST Encoder

Combined Representation：为了完整表示一条路径，将路径的表示与每个叶节点的表示拼接起来并应用一个全连接层：

$$z = tanh\left(W_{in}\left[encode\_path(v_1 ... v_l); encode\_token(value(v_1)); encode\_token(value(v_l))\right]\right) \quad (7)$$

Decoder Start State：decoder的起始状态为 z 的平均值：

$$h_0 = \frac{1}{k}\sum_{i=1}^{k} z_i$$

Attention：注意机制用于在解码时动态选择这些组合表示中的分布，就像NMT模型处理已编码的源token一样。

三、各论文内容总结

Code2seq: generating sequences from structured representations of code

D、实验

通过两个任务评估本模型：

代码总结：通过给定程序预测Java源码的名字；

代码标题：通过给定的代码片段为源码生成自然语言描述。

实验设置：

损失函数：交叉熵损失动量为0.95

学习率：0.01

输入层Dropout：0.25（代码总结任务）；0.7（代码标题任务）

D、实验

实验设置：

编码AST层Dropout： 0.5

代码总结任务LSTM单元数：128（encode层）；320（decode层）

代码标题任务LSTM单元数：256（encode层）；512（decode层）

k 值的选择：即为每个源码AST采样的路径数，本实验中选用200.k = 100结果较差，而k大于300结果无持续变好。此外，由于每个示例中Java大型训练集中的平均路径数为220个路径，因此对于一些大型方法而言，多达200个的路径数是有益的。

D、实验——代码总结

数据集：

Java-small：共包含11个项目，其中９个项目用于训练，一个项目用于验证另一个作为测试集。本数据集包含700K个示例。

Java-med：来自GitHub中前1000个顶级Java项目组成，随机选择其中800个项目进行训练，100个用于验证，100个用于测试，约包含4M个示例。

Java-large：来自GitHub中前9500个Java项目，随机选择9000个项目进行训练，250个用于验证，300个用于测试，该数据集包含约16M个示例。

三、各论文内容总结

Code2seq: generating sequences from structured representations of code

D、实验——代码总结

本实验中在相同数据集上与baseline模型比较结果

Table 1: Our model significantly outperforms previous PL-oriented and NMT models. Another visualization can be found in Appendix E.

Model	Java-small			Java-med			Java-large		
	Prec	Rec	F1	Prec	Rec	F1	Prec	Rec	F1
ConvAttention (Allamanis et al., 2016)	50.25	24.62	33.05	60.82	26.75	37.16	60.71	27.60	37.95
Paths+CRFs (Alon et al., 2018)	8.39	5.63	6.74	32.56	20.37	25.06	32.56	20.37	25.06
code2vec (Alon et al., 2019)	18.51	18.74	18.62	38.12	28.31	32.49	48.15	38.40	42.73
2-layer BiLSTM (no token splitting)	32.40	20.40	25.03	48.37	30.29	37.25	58.02	37.73	45.73
2-layer BiLSTM	42.63	29.97	35.20	55.15	41.75	47.52	63.53	48.77	55.18
TreeLSTM (Tai et al., 2015)	40.02	31.84	35.46	53.07	41.69	46.69	60.34	48.27	53.63
Transformer (Vaswani et al., 2017)	38.13	26.70	31.41	50.11	35.01	41.22	59.13	40.58	48.13
code2seq	**50.64**	**37.40**	**43.02**	**61.24**	**47.07**	**53.23**	**64.03**	**55.02**	**59.19**
Absolute gain over BiLSTM	+8.01	+7.43	+7.82	+6.09	+5.32	+5.71	+0.50	+6.25	+4.01

D、实验——代码总结

输出长度对结果的影响：本模型在所有代码长度上均优于所有baseline。对于短代码段（即少于3行），所有模型均能提供最佳结果。 随着输入代码大小的增加，所有检查的模型都显示自然下降，并且显示长度为9及以上的稳定结果。

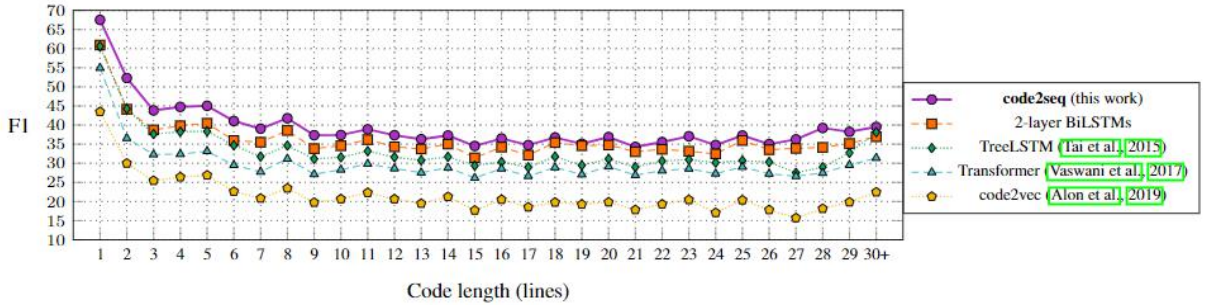


Figure 4: F1 score compared to the length of the input code. This experiment was performed for the code summarization task on the Java-med test set. All examples having more than 30 lines were counted as having 30 lines.

三、各论文内容总结

Code2seq: generating sequences from structured representations of code

D、实验——代码标题

数据集：使用CodeNN中的数据集，该数据集由来自StackOverflow的66,015对问题和答案组成。

评估脚本：使用与Iyer等人相同的评估脚本对BLEU评分进行了平滑处理。

Baseline：CodeNN, TreeLSTMs with attention, 2-layer bidirectional LSTMs with attention,以及 the Transformer

Table 2: Our model outperforms previous work in the code captioning task. †Results previously reported by Iyer et al. (2016), and verified by us. Another visualization can be found in Appendix D.

Model	BLEU
MOSES† (Koehn et al., 2007)	11.57
IR†	13.66
SUM-NN† (Rush et al., 2015)	19.31
2-layer BiLSTM	19.78
Transformer (Vaswani et al., 2017)	19.68
TreeLSTM (Tai et al., 2015)	20.11
CodeNN† (Iyer et al., 2016)	20.53
code2seq	**23.04**

E、验证实验及结果

1.No AST nodes：无需使用LSTM编码AST路径，仅使用第一个和最后一个终端值来构造输入矢量。

2.No decoder：而是使用单个softmax层将目标序列预测为单个符号。

3.No token splitting：无子token编码，而是嵌入完整token。

4.No tokens：仅使用AST节点，而不使用与终端关联的值

5.No attention：给定初始解码器状态，对目标序列进行解码，无需使用Attention。

6.No random：每次迭代均不重采样路径； 预先取样，并在整个培训过程中为每个示例使用相同的路径

Table 3: Variations on the code2seq model, performed on the validation set of Java-med.

Model	Precision	Recall	F1	ΔF1
code2seq (original model)	**60.67**	**47.41**	**53.23**	
No AST nodes (only tokens)	55.51	43.11	48.53	-4.70
No decoder	47.99	28.96	36.12	-17.11
No token splitting	48.53	34.80	40.53	-12.70
No tokens (only AST nodes)	33.78	21.23	26.07	-27.16
No attention	57.00	41.89	48.29	-4.94
No random (sample k paths in advance)	59.08	44.07	50.49	-2.74

F、总结

本文提出了一种代码到序列模型，该模型考虑了源代码的独特句法结构以及自然语言的顺序建模。 核心思想是在代码段的“抽象语法树”中对路径进行采样，使用LSTM编码这些路径，并在生成目标序列时对其进行关注。

通过使用本模型来预测三种大小不同的数据集上的方法名称，使用给定的部分代码和简短代码段来预测自然语言标题。

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

- 本文将深度学习中的循环神经网络模型应用于API 使用规约的学习及API 误用缺陷的检测.在大量的开源 Java 代码基础上,通过静态分析构造API 使用规约训练样本,同时利用这些训练样本搭建循环神经网络结构学习API 使用规约。在此基础上,本文针对API 使用代码进行基于上下文的语句预测,并通过预测结果与实际代码的比较发现潜在的API 误用缺陷。本文对所提出的方法进行实现,并针对Java 加密相关的API 及其使用代码进行了实验评估。结果表明,该方法能够在一定程度上实现API 误用缺陷的自动发现.

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

- PR-Miner[10]使用频繁项集挖掘技术，关注于挖掘代码中数据子项的频繁共现关系,但对API 使用的顺序信息没有加以利用
- Doc2Spec[7]通过分析自然语言编写的API文档，提取相关的API使用规约
- Bugram[8] 使用N-gram 语言模型，利用到了API 使用的顺序信息,但是对多样和变长的代码上下文存在模型的合成能力不足的问题.
- 循环神经网络能有效解决长距离依赖问题，被广泛应用于模式识别等领域。因此本文使用该模型也能有效学习API调用组合、顺序及控制结构等方面的使用规约

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

- 本文提出的API误用检测方法分为3个阶段：
- （1）训练数据构造：基于开源Java代码数据集,对源代码进行静态分析,构造抽象语法树,并进一步构造为API 语法图,用于API 调用序列的抽取,最后,将API 调用序列构造为深度学习模型需要的训练数据.

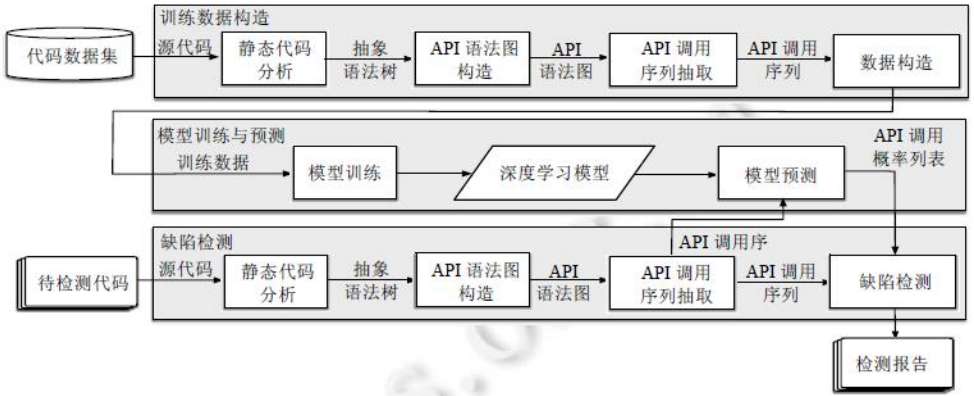


Fig.1 Overview of our approach

图 1 本文方法概览

- （2）模型训练与预测：训练深度学习模型,基于该模型和API 使用的上下文对API 调用序列中某处 的API 调用进行预测,得出该位置的API 调用概率列表.
- （3）缺陷检测：从待检测代码中抽取用于预测的API 调用序列,基于模型训练与预测阶段的模型预 测对每个位置进行API 调用预测,并获取每个位置的API 调用概率列表,将原代码中的API 调用序列 与预测得出的概率列表进行比对,得出最终的API 误用检测报告.

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（1）训练数据构造

- 该阶段分为4个子阶段：静态代码分析、API 语法图构造、API 调用序列抽取、数据构造.
- （1）静态代码分析

  本文使用JavaParser对源代码文件进行解析,并获取表示源代码语法结构的AST

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（1）训练数据构造

- （2）API 语法图构造（在AST基础上进一步解析构造）

相关定义：

- 节点
  - 方法节点,表示API 语法图来源(文件路径、类、方法).
  - API 节点,表示API对象创建、方法调用、类变量访问.
  - 控制节点,表示控制结构
- 边
  - 顺序边，表示流(控制流)的方向
  - 控制边,表示控制结构之间的顺序关系

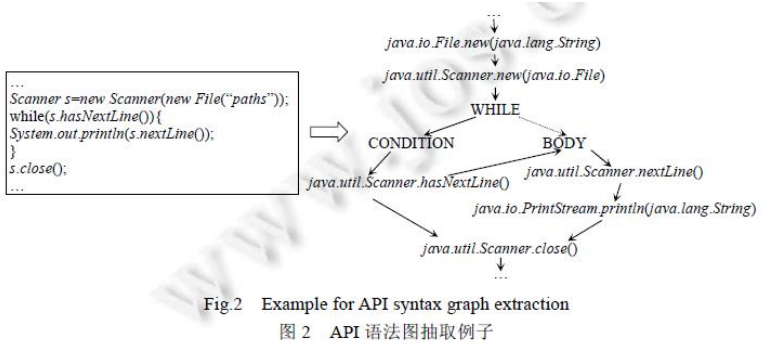


Fig.2 Example for API syntax graph extraction

图 2 API 语法图抽取例子

- API语法图：一个根节点,能构成一个最简的API 语法图。边从父节点指向子节点,没有子节点的节点称为叶子节点。图中没有环路。两个API 语法图,从一个API 语法图的叶子节点出发,向另一API 语法图的根节点加边,称为移植。

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（1）训练数据构造

- 对API语法图相关定义进行UML建模
- Node 表示API 语法图上的节点
- Edge 表示父节点连接子节点的边
- Graph 表示API 语法图,该图上的节点由Node 表示,其中,一个特殊的根节点root表示一个API 语法图 的开始.从Graph a 的叶子节点到Graph b 的根节点都加上边,这个操作称为移植(transplant).

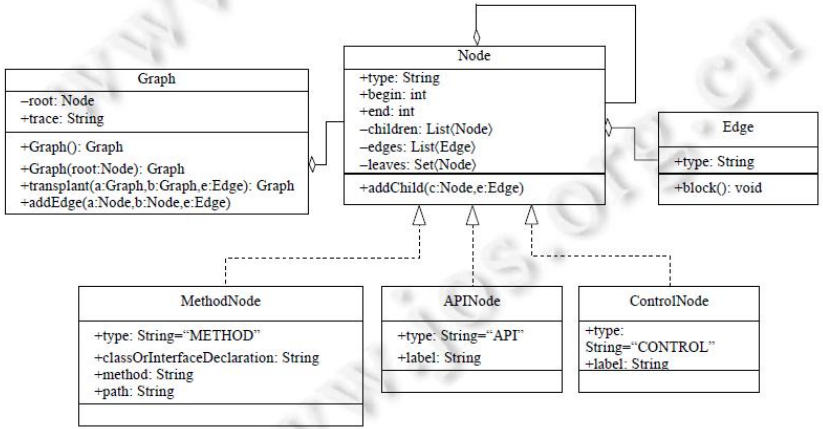


Fig.3 Class diagram of API syntax graph

图 3 API 语法图相关类图

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（1）训练数据构造

- 从静态代码到构建API语法图的具体步骤：
- （1）基于JavaParser分析代码为AST
- （2）基于 JavaParser 对每个类中的方法进行抽取,将信息存储到方法节点中,并以该节点为根节点,构建 API 语法图g
- （3）对抽取的每个方法根据不同规则，迭代地构建API语法图
- （4）将 API 语法图g’移植到API 语法图g 中,获得每个类中方法对应的API 语法图.

（1）训练数据构造

- （3）API 调用序列抽取

从API 语法图根节点出发,根据顺序边以及表示顺序的控制边对API 语法图进行深度遍历得到的节点标签序列,称为这个API 语法图上的API 调用序列.表示顺序的控制边有以下几种:IF→CONDITION、WHILE→CONDITIO、TRY→TRYBLOCK、FOR→INITAILIZATION、FOREACH→VARIABLE.

主要思想是对API 语法图 进行深度遍历,提取所有存在API 调用节点的API 调用序列作为与该API 语法图对应的API 调用序列集合.所 有提取的API 调用序列以EOS 控制节点结束.

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（1）训练数据构造

- （4）产生训练数据
- 包括了词汇表的建立以及训练数据的构造两部分：

1. 构建词汇表：统计调用序列中出现的API调用词频,建立API调用与API调用编号一一对应的词汇表,并持久化到本地的词汇表文件中.
2. 构造训练数据：基于词汇表将API 调用序列转换为API调用编号序列，对构造好的编号序列进行遍历, 构造<API 调用前文编号序列,API 调用编号>形式的数据,并作为训练数据持久化至本地训练数据文件中。API 调用前文指的是在该序列中API调用之前的所有API调用组成的序列,API调用前文的编号序列形式称为API调用前文编号序列

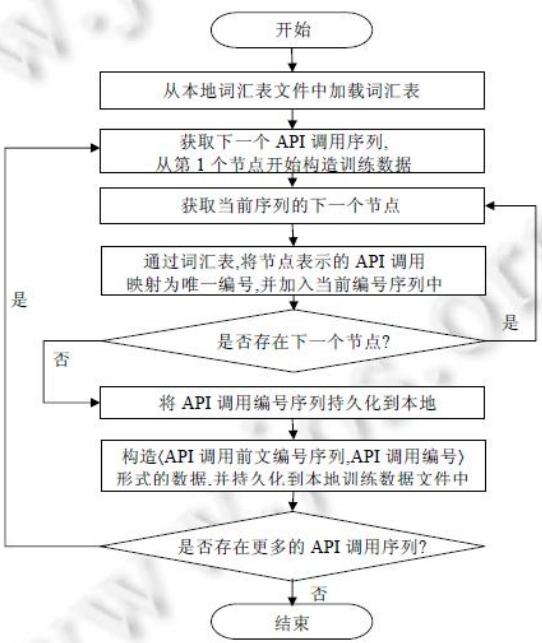


Fig.4 Flowchart of training data build algorithm

图 4 构造训练数据算法流程图

（1）模型训练与预测

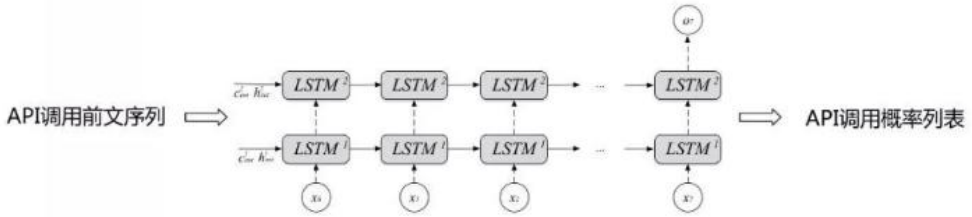
1. 模型训练

- 模型选用TensorFlow 1.6.0作为实现框架，使用深层循环网络结构配合长短时记忆网络。计算流程如下:

1. 经过词向量层，将输入的API调用前文编号序列中的每个API调用编号嵌入到一个实数向量中
2. 输入向量经过 dropout 层,丢弃部分信息以增强模型的健壮性
3. 使用 TensorFlow 的动态RNN 接口实现神经网络主要结构
4. 循环神经网络的输出经过一个全连接层转化,得到与词表大小一致的输出称为logits;随后,logits 经过 Softmax 处理及交叉熵计算,可以分别得到API 调用的概率分布以及损失值;TensorFlow 将基于给定的优化函数,对参数进行优化.
5. 在训练过程中加入预测正确率(accuracy)作为训练时的反馈,同时也作为优化训练模型的参考

（1）模型训练与预测

- 2. 模型预测
- 基于训练好的深度学习模型,将用于预测的API 调用前文序列作为输入, 预测下一位置上的API 调用的概率分布,将这个概率分布进行排序,得出下一位置上的API 调用概率列表.该 API 调用概率列表将作为之后缺陷检测的重要部分.



模型（Deep LSTM）训练/预测示意图

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（2）缺陷检测

- 将原代码中每个API 调用序列上的每个位置的API 调用与预测得出的API 调用概率列表进行比对, 应用定义好的代码缺陷检测规则,报告出候选的代码缺陷,得出最终的API 误用检测报告。
- 基于Python 语言中的Web 端框架Flask搭建Web 服务端,将API调用概率列表的预测包装为Web API 供客户端调用.

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（2）缺陷检测

- 具体步骤如下：

1. 从待检测的源代码中抽取 API 语法图
2. 从 API 语法图中抽取API 调用序列
3. 对每一个 API 调用序列,从第2 个位置开始,进行缺陷检测
4. 基于检测位置的 API 调用前文,预测该位置可能出现的 API 调用概率列表
5. 判断该位置的 API 调用在API 调用概率列表中的排名是否在可接受范围，如果超出则报告缺陷
6. 结束检测后,可根据缺陷报告进行进一步的评估

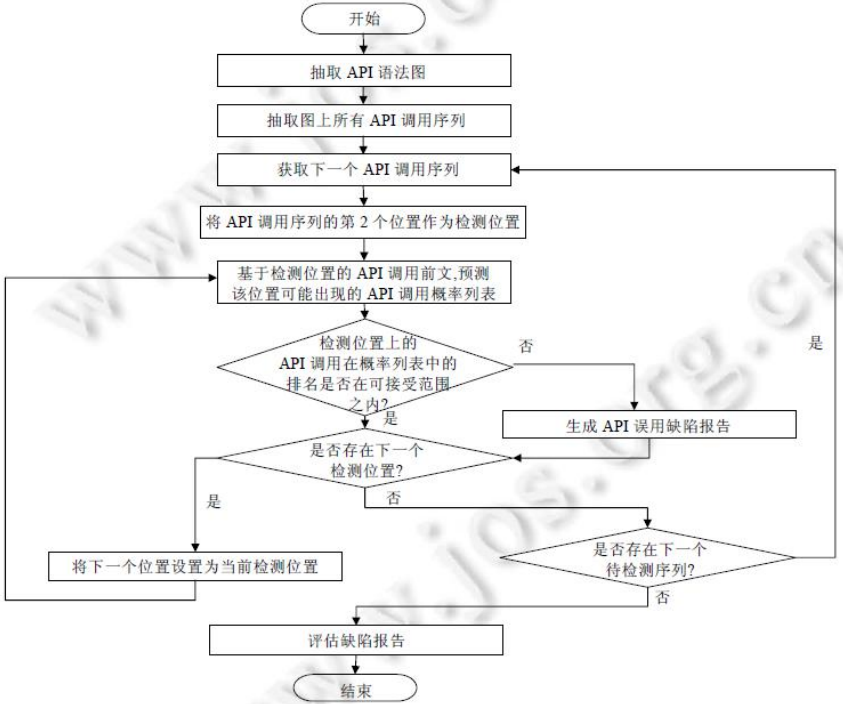


Fig.6 Flowchart of bug detection algorithm

图 6 缺陷检测算法流程图

（3）实验评估

- 分为深度学习模型训练和代码缺陷检测实验两部分
- 实验对象：JCE(Java cryptography extension)是JDK 提供的一组包,它们提供用于加密、密钥生成和协商及消息验证码 (MAC)算法的框架和实现,JCE 提供的Java 密码学APIs(Java cryptography APIs)在javax.crypto 包下

数据来源	GitHub
关键词	javax.crypto
时间限制	最后修改日期在2018年1月1日前
数量	14422个java代码文件
训练原始文件总大小	50MB
API调用序列	38602条
训练数据	388973条
词表大小	1564

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（3）实验评估

- 1. 深度学习模型训练实验
- 实验探究隐层大小(HIDDEN_SIZE)、深层循环网络层数(NUM_LAYER)、学习率(LR)以及迭代次数(NUM_EPOCH)对模型效果的影响.对模型效果的评判效果以验证集上的分类准确率(accuracy)为准,分类准确率为模型预测Top-1 为目标词的准确率.通过调整深度学习模型的部分参数,对模型进行调整训练,训练集为总数据集的90%,验证集为总数据集的10%.

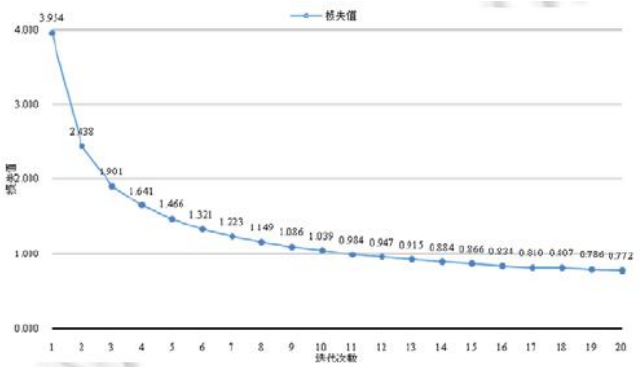


Fig.7 Loss of DL model

图 7 深度学习模型损失

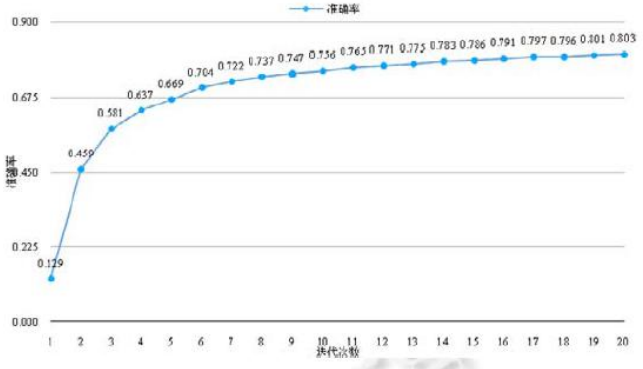


Fig.8 Accuracy of DL model (*HIDDEN_SIZE*=250,*NUM_LAYER*=2,*LR*=0.002)

图 8 深度学习模型准确率(*HIDDEN_SIZE*=250,*NUM_LAYER*=2,*LR*=0.002)

（3）实验评估

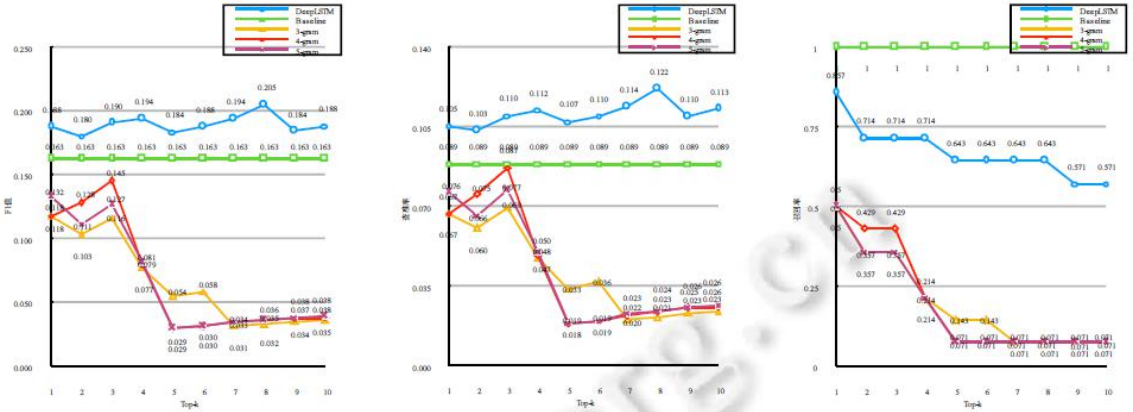
- 2. 代码缺陷检测实验
- 测试标准：
  - 定义 TP 为检测文件API 误用缺陷位置正确的不重复缺陷报告数.
  - 定义 FP 为检测文件API 误用缺陷位置错误的不重复缺陷报告数.
  - 定义 FN 为未进行报告的缺陷报告数.
- 实验中采用准确率和召回率的调和均值F1 进行API 误用缺陷检测的实验评价.
- 测试数据包含四种类型地API误用：
  a. 使用了多余的 API 调用;
  b. 使用了错误的 API 调用;
  c. 遗漏了关键的 API 调用;
  d. 忽略了对 API 调用中可能抛出的异常的处理.

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（3）实验评估

- 为验证模型的有效性,设置以下模型作为实验的对比模型
  - 基准模型（Baseline）
  - N-gram检测模型（N-gram）
- 探究可接受阈值的取值对代码缺陷检测模型的F1值的影响.



(a) $F1$ 值　(b) 查准率　(c) 召回率

Fig.9　Result of API misuse bug detection experiment

图 9　API 误用缺陷检测实验结果

三、各论文内容总结

API Misuse Bug Detection Based on Deep Learning

（3）实验评估

Fig.10 Analysis of relationship between acceptable thresholdand type of API misuse

图 10 可接受阈值与 API 误用类型关系分析

- 本实验中,模型对检测错误的API 调用类型的API 误用效果较好,而在发现遗漏API 调用类型的API 误用上能力稍有欠缺

四、目前存在的问题

- 虽然[4]中提出的较新的API误用检测方法，可以避免仅从最频繁的API用法中挖掘，从而提高检测的准确率和召回率。但现有的大多数检测方法其仍是基于特定规则的方法，并且方法对于特定编程语言的依赖性强，可移植性差。
- 缺乏不同编程语言之间可共用的检测方法，每种API误用检测方法和相关基准数据集都是基于特定语言的。
- 现有方法大多是静态方法，不能使用在运行的动态环境中。

五、可进一步研究的问题

- 可以在已有基于规则的API误用检测器基础上进行改进，增强它识别的精确率和召回率
- 引入深度学习到API误用检测任务中的是否可行
- 提出具有普适性的API误用检测方法，以应用于不同编程语言、不同应用场景的代码上
- 研究有关动态API误用检测的方法

六、可探讨的研究思路

- （1）在文献[4]的基础上，引入深度学习的方法。
- 首先使用[4]中提出的突变子对各API代码进行突变生成相应突变体，然后将各突变体代码转换为AST抽象语法树，之后根据各API调用位置从AST中提取该API调用序列。使用某种方式进行编码后输入RNN或LSTM之类的用于处理时序数据的神经网络中，输出的结果为对应各API误用种类的概率分布。

六、可探讨的研究思路

- 这个思路上可以深入探讨的部分有两个地方：

1. 将生成的API突变体以何种方法转换为可输入神经网络的输入类型，现阶段考虑的是使用AST，但具体做法还不是十分明确
2. 应对我们的问题，使用什么样的RNN神经网络结构是最合适的

六、可探讨的研究思路

- （2）文献[4]中突变子可以考虑使用聚类的方法获得
- [4]中提出了八种突变子用于对API的正常使用模式进行突变，自动获得可使用的API误用的代码。其中的8种突变体是作者人工选取的，我们可以借用机器学习中的相关聚类方法，对包含API误用信息的训练数据进行训练分类，自动获得不同的API误用种类，这里每种种类都可对应不同的突变子。

参考文献

- [1] SungGyeong Bae, Hyunghun Cho, Inho Lim, Sukyoung Ryu: SAFEWAPI: web API misuse detector for web applications. SIGSOFT FSE 2014: 507-517
- [2] Sven Amann, Sarah Nadi, Hoan Anh Nguyen, Tien N. Nguyen, Mira Mezini: MUBench: a benchmark for API-misuse detectors. MSR 2016: 464-467
- [3] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, Miryung Kim: Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. ICSE 2018: 886-896
- [4] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, Zhendong Su: Exposing library API misuses via mutation analysis. ICSE 2019: 866-877
- [5] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, Mira Mezini: A Systematic Evaluation of Static API-Misuse Detectors. IEEE Trans. Software Eng. 45(12): 1170-1188 (2019)
- [6] Chi Li, Zuxing Gu, Min Zhou, Jiecheng Wu, Jiarui Zhang, Ming Gu: API Misuse Detection in C Programs: Practice on SSL APIs. International Journal of Software Engineering and Knowledge Engineering 29(11&12): 1761-1779 (2019)

参考文献

- [7] Hao Zhong, Lu Zhang, Tao Xie, Hong Mei: Inferring Resource Specifications from Natural Language API Documentation. ASE 2009: 307-318
- [8] Song Wang, Devin Chollak, Dana Movshovitz-Attias, Lin Tan: Bugram: bug detection with n-gram language models. ASE 2016: 708-719
- [9] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, Mira Mezini: Investigating next steps in static API-misuse detection. MSR 2019: 265-275
- [10] Zhenmin Li, Yuanyuan Zhou: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. ESEC/SIGSOFT FSE 2005: 306-315
- [11] Uri Alon, Shaked Brody, Omer Levy, Eran Yahav: code2seq: Generating Sequences from Structured Representations of Code. ICLR (Poster) 2019
- [12] WANG Xin,CHEN Chi,ZHAO Yi-Fan,PENG Xin,ZHAO Wen-Yun.API Misuse Bug Detection Based on Deep Learning.Journal of Software,2019,30(5):1342-1358

下一步计划阅读的参考文献

- [1] Sven Amann: A Systematic Approach to Benchmark and Improve Automated Static Detection of Java-API Misuses. Darmstadt University of Technology, Germany, 2018
- [2] Sharmin Afrose, Sazzadur Rahaman, Danfeng Yao: CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses. SecDev 2019: 49-61
- [3] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, Tien N. Nguyen: Graph-based mining of multiple object usage patterns. ESEC/SIGSOFT FSE 2009: 383-392
- [4] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Rosu, Darko Marinov: How good are the specs? a study of the bug-finding effectiveness of existing Java API specifications. ASE 2016: 602-613

谢谢!