Checking temporal patterns of API usage without code execution

Erick Raelijohn, Michalis Famelis, Houari Sahraoui

FormaliSE 2021

Université de Montréal DIRO geodes

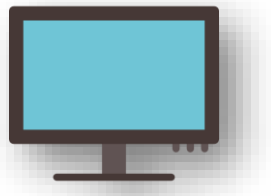
IDEs are Awesome!

```
10  public class CrHelper extends SQLiteClosable {
11      private WeakHashMap<SQLiteClosable, Object> mPrograms;
12      Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
13  ›   public void onCreate(SQLiteDatabase db) {…
20  ›   public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {…
49
50      public void close() {
51            Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
52              while (iter.hasNext()) {
53                Map.Entry<SQLiteClosable, Object> entry = iter.next();
54                SQLiteClosable program = entry.getKey();
55                if (program != null) {
56                    program.onAllReferencesReleasedFromContainer();
57                }
58              }
59      }
60  }
61
```

Code

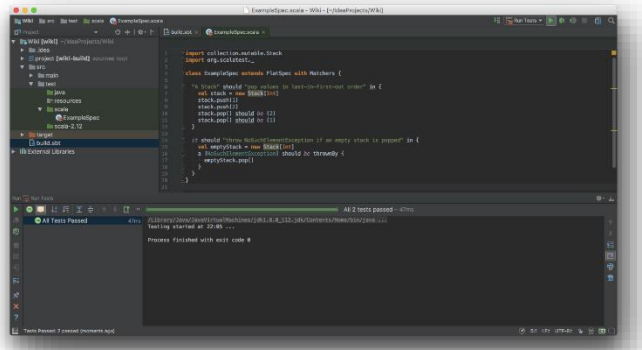
Adele

Productivity!



IDE

Feedback



- Automatic code generation
- Organize imports
- “On the fly” compilation

But not as much as they could be

Adele

API Knowledge

- Read and understand the documentation
- Look for examples
- Seek community support

```
10  public class CrHelper extends SQLiteClosable {
11      private WeakHashMap<SQLiteClosable, Object> mPrograms;
12      Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
13  ›   public void onCreate(SQLiteDatabase db) {=
20  ›   public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {=
49
50      public void close() {
51            Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
52              while (iter.hasNext()) {
53                Map.Entry<SQLiteClosable, Object> entry = iter.next();
54                SQLiteClosable program = entry.getKey();
55                if (program != null) {
56                    program.onAllReferencesReleasedFromContainer();
57                }
58              }
59      }
60  }
61
```

Code

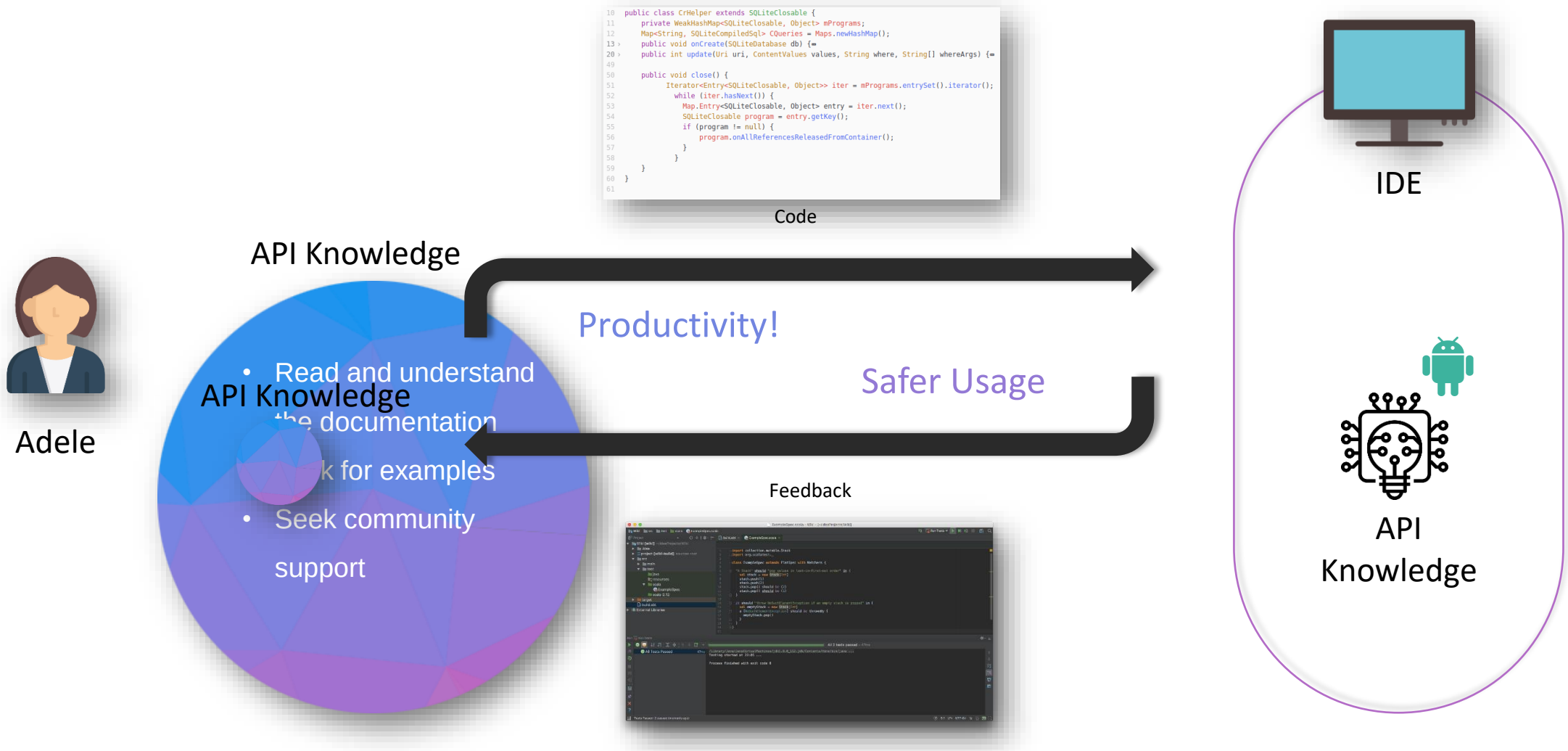
~~Productivity~~!

IDE

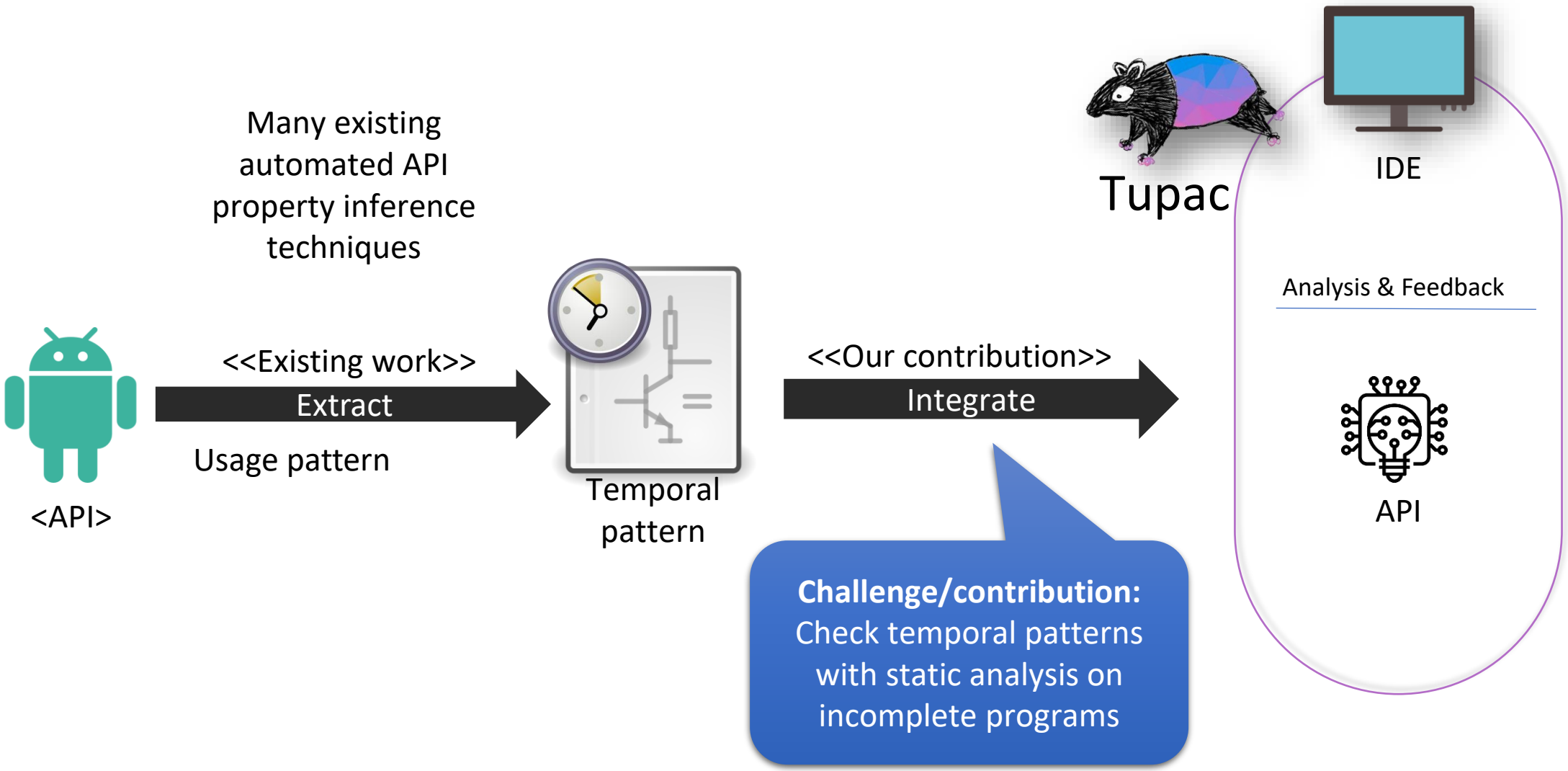
Feedback

Feedback from other sources
(e.g., execution, bug reports)

Our Vision


Code
IDE
API Knowledge
Adele
API Knowledge
Read and understand the documentation
k for examples
Seek community support
Productivity!
Safer Usage
Feedback
API Knowledge

Temporal Usage PAttern Checker (**Tupac**) for APIs



Outline

- Motivation

- Approach
- Preliminary Evaluation
- Conclusion

Basic Assumptions

- Object Oriented (OO) paradigm
- Check pattern properties ***in the IDE*** as part of regular coding rhythm
- Be able to check patterns on **incomplete code**
- Static analysis: no need to run the code, no reliance on tests, traces
- Temporal API usage properties **as external inputs**
  (from documentation, or an extraction tool)
- Properties language: Linear Temporal Logic - LTL

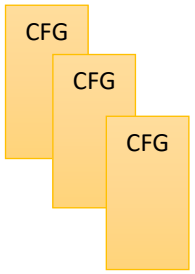
Tupac's Approach

Temporal API usage properties (LTL)

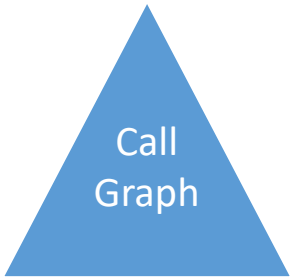
Code in IDE

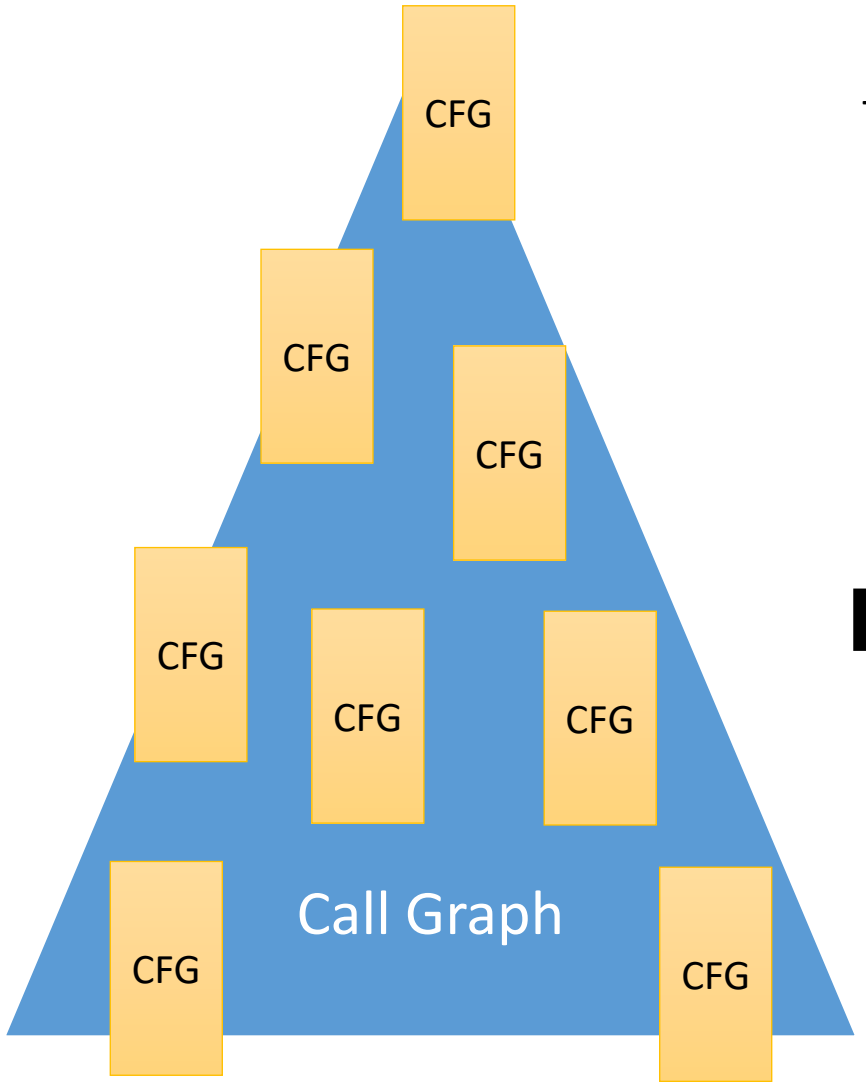
```
public class CrHelper extends SQLiteClosable {
    private WeakHashMap<SQLiteClosable, Object> mPrograms;
    Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
    public void onCreate(SQLiteDatabase db) {…
    public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {…

    public void close() {
          Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
            while (iter.hasNext()) {
              Map.Entry<SQLiteClosable, Object> entry = iter.next();
              SQLiteClosable program = entry.getKey();
              if (program != null) {
                  program.onAllReferencesReleasedFromContainer();
              }
            }
    }
}
```



Control Flow Graphs



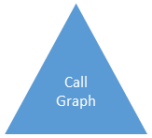


"Deep Graph"

Core Concepts

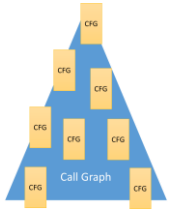
- Control Flow Graph (CFG) *of method*
  Can be used for intraprocedural analysis

Instructions, predicates (ifs, loops, …)



- Call Graph (CG) *of OO system*
  Shows intraprocedural dependencies

Nodes: methods in the system,
Edges: caller-callee relationships



- Deep Graph (DG) *of OO system*
  Combines the CG and all CFGs of a system
  Each control transition annotated with the method call triggered it

- "Trace": a path on the DG
  Executing the code *might* produce this sequence of API calls
  I.e., the DG may contain more behaviours than the code

Deep Graph Construction

Inputs: a set of control flow graphs, a call graph

Output: a deep graph

1. Prune and slice inputs to only keep API calls and calls to other system methods
2. Following the CG, annotate each edge of CFGs with the caller method
3. Connect the forest of annotated CFGs based on the CG

System method calls

SQLite API calls

Example

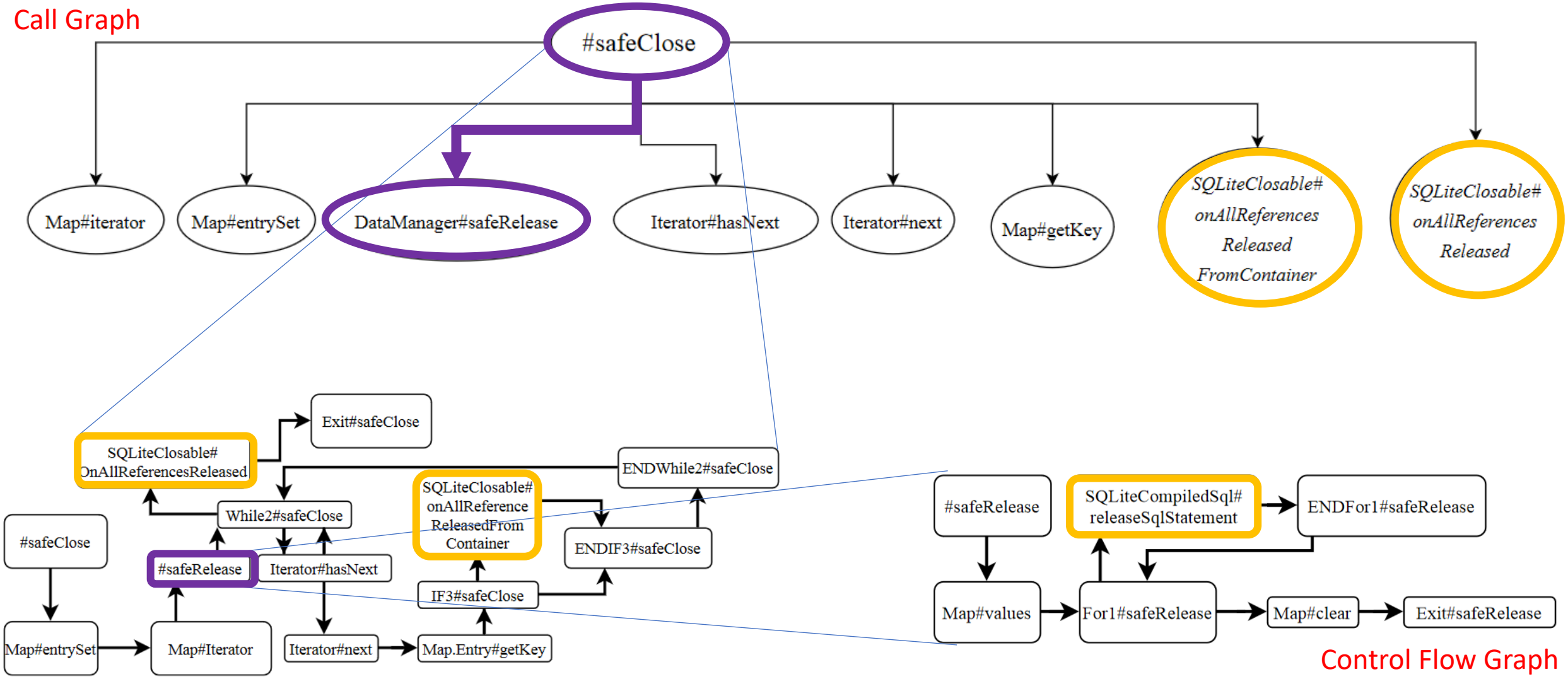
```
1  public class DataManager extends SQLiteClosable {
2     ...
3     public void safeClose() {
4       Iterator<Entry<SQLiteClosable, Object>> iter =
                mPrograms.entrySet().iterator();
5       this.safeRelease();
6       while (iter.hasNext()) {
7         Map.Entry<SQLiteClosable, Object> entry = iter.next();
8         SQLiteClosable program = entry.getKey();
9         if (program != null)
10          program.onAllReferencesReleasedFromContainer();
11      }
12      onAllReferencesReleased();
13    }}
14    public void safeRelease(){
15      for (SQLiteCompiledSql compiledSql : this.CQueries.values())
16        compiledSql.releaseSqlStatement();
17      CQueries.clear();
18  }}
```

System method calls

SQLite API calls

Intuition: Follow the calls

Call Graph



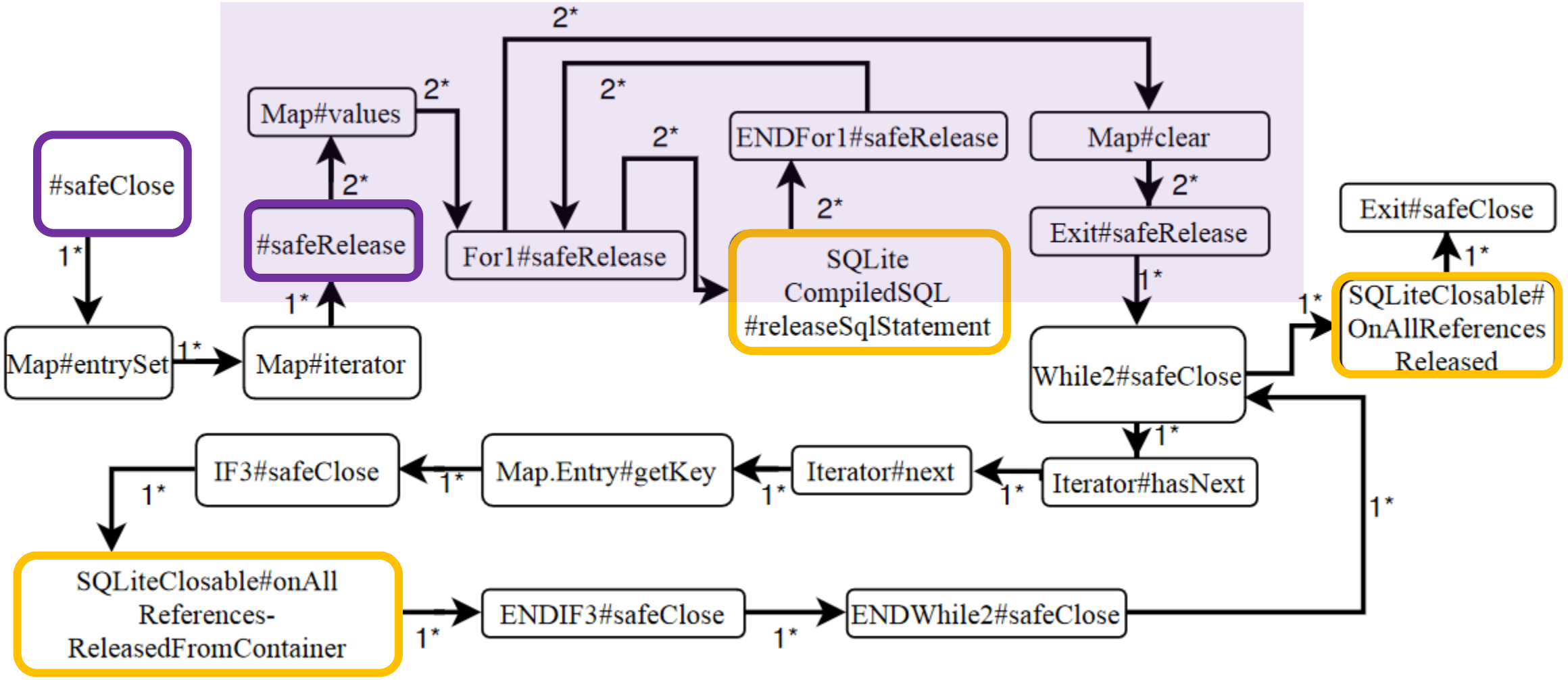
Control Flow Graph

Control Flow Graph

System method calls

SQLite API calls

Result: Deep Graph



Model Checking the Deep Graph

- DG translated to a NuSMV module
  - State: method and API calls
  - Transition relation: caller annotations +constraints to enforce determinism
- API property patterns in LTL
  - Assumed as given externally
- Possible results:

TRUE	The pattern is respected
FALSE + counterexample	DG path where the pattern is violated. - True positive: API misuse - False positive: (a) pattern irrelevant for code fragment (b) imprecision of DG

Tupac produces a visualization to help make sense

Outline

- Motivation
- Approach
- Preliminary Evaluation
- Conclusion

Research Questions

🎯 RQ1 : How good is Tupac for detecting pattern violations?

🌐 RQ2: Interprocedural vs intraprocedural analysis

🏃 RQ3: Is Tupac fast enough be usable without stalling devs' workflow?

Setup

- 4 open source programs
  HtmlCompressor, doc-to-pdf-converter, jar2Java, JTar
- 4 commonly used APIs
  util.Map, util.List, util.Set, io.File
- Typical use cases → Trace collection → Sequences of API calls
- API pattern mining using an existing technique [1]
- 🎯RQ1, 🌐RQ2: only keep 26 shared patterns
- For 🥷RQ3: sampled 124 patterns of various AST complexities

Independent from Tupac!
Any extraction technique might be used

[1] M. Saied, E. Raelijohn, E. Batot, M. Famelis, and H. Sahraoui. Towards assisting developers in api usage by automated recovery of complex temporal patterns. Information and Software Technology 2020

RQ1 Results

Manual analysis

By Tupac

		Interprocedural analysis			
Metric	Description	p1	p2	p3	p4
CGT	Ground Truth Correct	17	1	15	16
VGT	Ground Truth Violated/absent	9	25	11	10
VTP	True Positive Violated/absent	9	24	11	10
VFP	False Positive Violated/absent	6	0	7	9
Precision		0.6	1	0.61	0.53
Recall		1	0.96	1	1
F-score		0.75	0.98	0.76	0.69

Precision = VTP/(VTP + VFP), Recall = VTP/VGT, F−score = 2∗(Precision∗Recall)/(Precision+Recall)

p1: HtmlCompressor
p2: doc-to-pdf-converter
p3: jar2Java
p4: JTar

1. Results consistent for p1-p4
2. Great recall
3. Precision better than random (>50%)

RQ2 Results

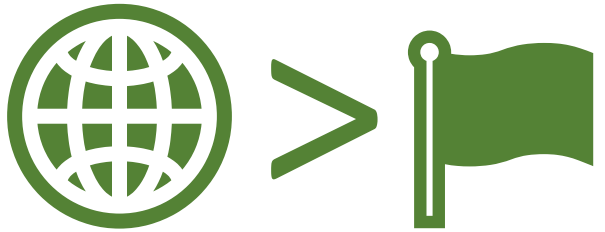
Intraprocedural analysis: within a single method; not following method calls
Might it be a cost-effective shortcut to the expensive DG construction?

Using only the Control Flow Graph

		Interprocedural analysis				Intraprocedural analysis			
Metric	Description	p1	p2	p3	p4	p1	p2	p3	p4
CGT	Ground Truth Correct	17	1	15	16	17	1	15	16
VGT	Ground Truth Violated/absent	9	25	11	10	9	25	11	10
VTP	True Positive Violated/absent	9	24	11	10	9	25	11	10
VFP	False Positive Violated/absent	6	0	7	9	10	0	9	16
Precision		0.6	1	0.61	0.53	0.47	1	0.55	0.38
Recall		1	0.96	1	1	1	1	1	1
F-score		0.75	0.98	0.76	0.69	0.64	1	0.71	0.56

Precision = VTP/(VTP + VFP), Recall = VTP/VGT, F−score = 2∗(Precision∗Recall)/(Precision+Recall)

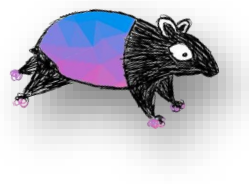
1. Similar recall
2. Bad precision (worse than random)



RQ3 Results

Independent from Tupac

TABLE VI: Average DG creation time per project in ms



Project	Size(LOC)	DG(ms)
p1	5309	592
p2	592	183
p3	2879	758
p4	1314	270

TABLE VII: Average verification time per single pattern

API	Average(ms)	Av. per single pattern(ms)
io#File	58679,50	814,99
util#List	10783,00	770,21
util#Set	8541,50	711,79
util#Map	12645,00	486,35

All patterns

1 pattern

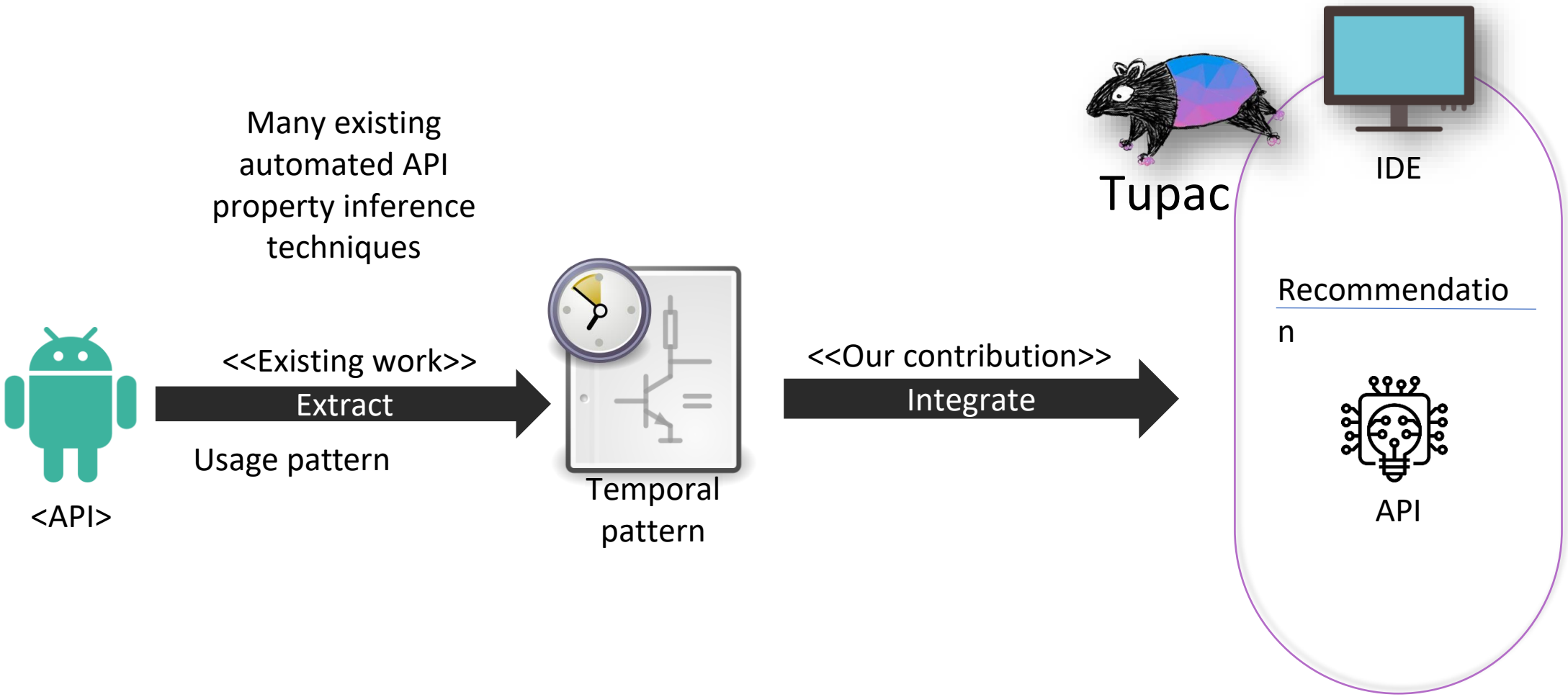
1. On average, less than 1 second to create DG
   (DGs are reusable!)

2. On average, 0.7 seconds to check a single pattern

Outline

- Motivation
- Approach
- Preliminary Evaluation
- Conclusion

Temporal Usage PAttern Checker (**Tupac**) for APIs



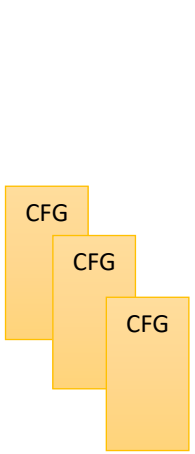
Tupac's Approach

Temporal API usage properties (LTL)

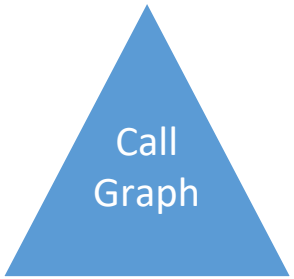
Code in IDE

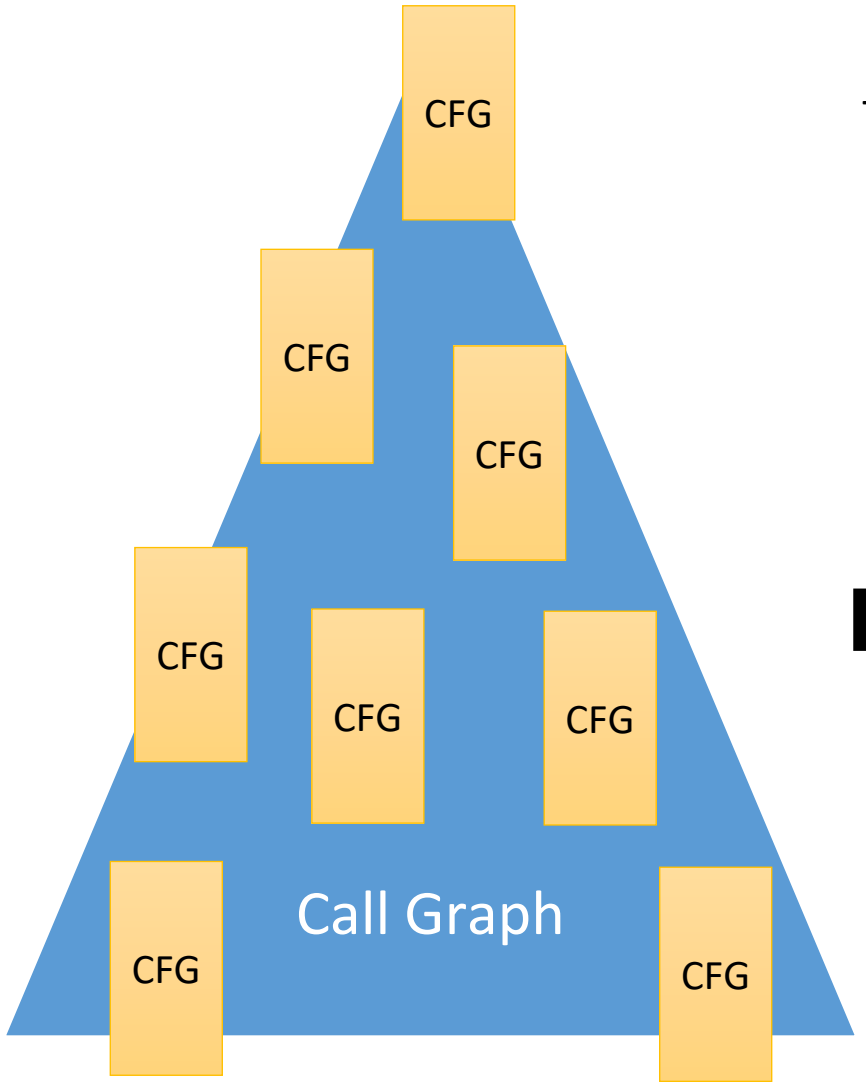
```
public class CrHelper extends SQLiteClosable {
    private WeakHashMap<SQLiteClosable, Object> mPrograms;
    Map<String, SQLiteCompiledSql> CQueries = Maps.newHashMap();
    public void onCreate(SQLiteDatabase db) {…
    public int update(Uri uri, ContentValues values, String where, String[] whereArgs) {…

    public void close() {
          Iterator<Entry<SQLiteClosable, Object>> iter = mPrograms.entrySet().iterator();
            while (iter.hasNext()) {
              Map.Entry<SQLiteClosable, Object> entry = iter.next();
              SQLiteClosable program = entry.getKey();
              if (program != null) {
                  program.onAllReferencesReleasedFromContainer();
              }
            }
    }
}
```



Control Flow Graphs





"Deep Graph"

Preliminary Evaluation Findings

Tupac is reasonably good at detecting pattern violations

Interprocedural is better than intraprocedural analysis

Tupac is fast enough to be usable without stalling devs' workflow

Limitations and Future Work

- Does not handle complex code structures (e.g., chained instructions)
- Many false positives
  - No vacuity testing: what patterns are even relevant?
  - DG encodes more behaviours than the code
  - Do not distinguish between different objects making the call
- Need more thorough evaluation, validation with users, and deeper comparison with similar approaches
  - Challenge: no common static analysis benchmark

Checking temporal patterns of API usage without code execution

Erick Raelijohn

Michalis Famelis

Houari Sahraoui

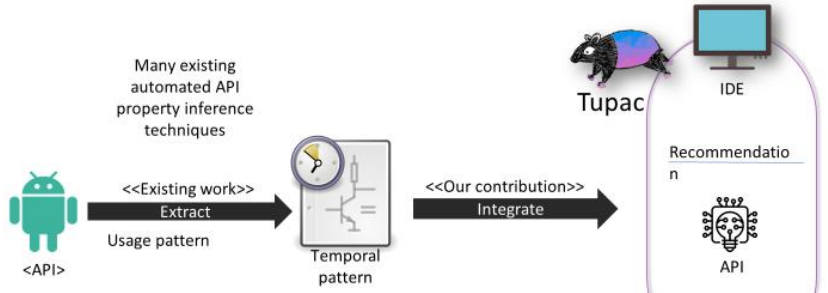
Université de Montréal

DIRO

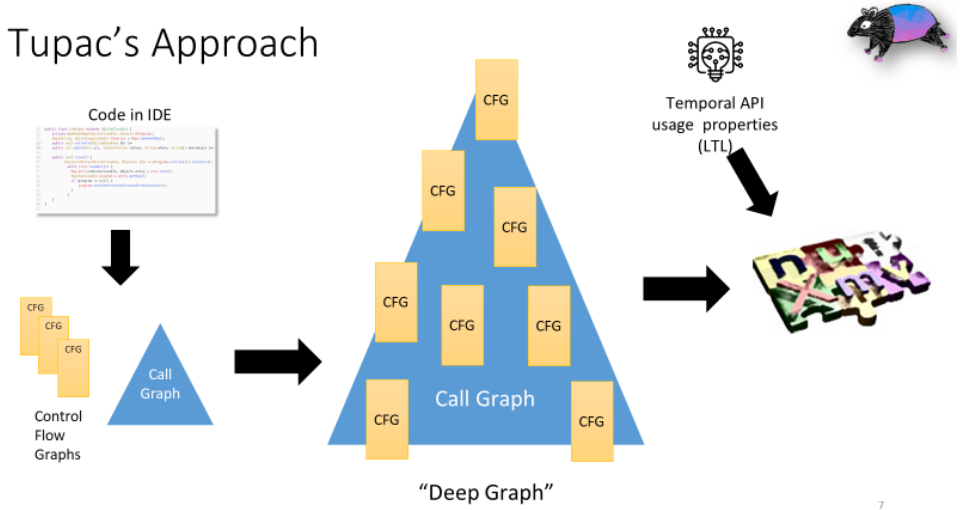
geodes

Our Vision

Temporal Usage PAttern Checker (**Tupac**) for APIs



Tupac's Approach



Findings

How good is Tupac for detecting pattern violations?

Interprocedural vs intraprocedural analysis

Is Tupac fast enough be usable without stalling devs' workflow?