Mining Framework Usage Graphs from App Corpora

Sergio Mover, Sriram Sankaranarayanan, Rhys Braginton Pettee Olsen, Bor-Yuh Evan Chang
University of Colorado Boulder, USA

***Abstract*—We investigate the problem of mining graph-based usage patterns for large, object-oriented frameworks like Android—revisiting previous approaches based on *gr*aph-based *o*bject *u*sage *m*odels (*groum*s). Groums are a promising approach to represent usage patterns for object-oriented libraries because they simultaneously describe control flow and data dependencies between methods of multiple interacting object types. However, this expressivity comes at a cost: mining groums requires solving a subgraph isomorphism problem that is well known to be expensive. This cost limits the applicability of groum mining to large API frameworks.**

In this paper, we employ groum mining to learn usage patterns for object-oriented frameworks from program corpora. The central challenge is to scale groum mining so that it is sensitive to usages horizontally across programs from arbitrarily many developers (as opposed to simply usages vertically within the program of a single developer). To address this challenge, we develop a novel groum mining algorithm that scales on a large corpus of programs. We first use frequent itemset mining to restrict the search for groums to smaller subsets of methods in the given corpus. Then, we pose the subgraph isomorphism as a SAT problem and apply efficient pre-processing algorithms to rule out fruitless comparisons ahead of time. Finally, we identify containment relationships between clusters of groums to characterize *popular* usage patterns in the corpus (as well as classify less popular patterns as possible anomalies). We find that our approach scales on a corpus of over five hundred open source Android applications, effectively mining obligatory and best-practice usage patterns.

I. Introduction

We consider the problem of mining graph-based descriptions of application programming interface (API) usage patterns from large software repositories. Such usage patterns are invaluable in understanding how the API is typically used by developers and help highlight anomalous usage. This work can in turn lead to automatic approaches to detecting potential defects, automatic code completion, and automatic repair. The API usage mining problem has been well-studied in the past two decades with numerous proposed approaches (e.g., [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14]). The work continues, as each approach has an *expressivity* versus *efficiency* tradeoff. As the API usage patterns get more expressive, the problem of capturing common patterns from large repositories becomes ever harder. The ideal usage pattern describes (a) the control flow of the various methods called, including control structures such as branches and loops; and (b) the data flow between how an object created by one method call gets utilized in another call. Commonly used APIs such as Android involve a large collection of methods and API usage protocols, each involving multiple object types as well as constraints on both the control and data flows.

Graph-based models of API usage are therefore very natural for such applications. A previous work [15] considered the problem of mining program-specific usage patterns using GRaph-based Object Usage Models (*groums*), an abstracted control-flow graph with data dependency edges. In a groum, nodes represent API calls (or control structures) and edges represent data dependencies or control flow between the nodes.

While groums are a natural and promising representation for expressing API usage, the main unsolved challenge in applying groum mining for finding framework usage patterns is to efficiently scale on a large corpora of Android applications (apps). Efficiently mining groums from a large corpus is challenging because of the cost of computing subgraph isomorphisms, the basic operation used to compute graph patterns. To illustrate this challenge, we tried to run the GrouMiner tool [15] on over five hundred Android projects with over 70,000 methods downloaded from GitHub. While this was never the intended application of GrouMiner, this approach also did not produce results despite our running it over 3 days.

To address this challenge, we make the following contributions:

- We combine frequent itemset mining with groum mining to restrict the search for relevant groums to smaller subsets of API methods (Section II). At a high level, we first use the mined itemsets as a basis for partitioning the entire corpus of groums into smaller clusters.
- We use a SAT-based encoding, coupled with filtering techniques to avoid unnecessary solver calls (Section IV), for computing subgraph isomorphisms, the main operation used to construct the lattice-ordered bins of groums.
- We organize the groums of each sub-corpus defined by a itemset cluster into *lattice-ordered* bins of groums (*logroum*s), using containment-based subsumption relation to define the order of the lattice. Groum bins can then be labeled as Popular, Anomalous, or Isolated usage patterns (Section V) according to their frequency of occurrence in the corpus *and* the ordering relationship with other bins. Using the mined itemsets, we then further slice the graphs according to the items (API calls), to compute subgraph isomorphisms on smaller instances.
- We evaluate our approach, BigGroum, on a corpus consisting of over five hundred Android apps (Section VI). We first show that BigGroum is *efficient*, being able to mine the patterns for the entire corpus of apps, and that our con-

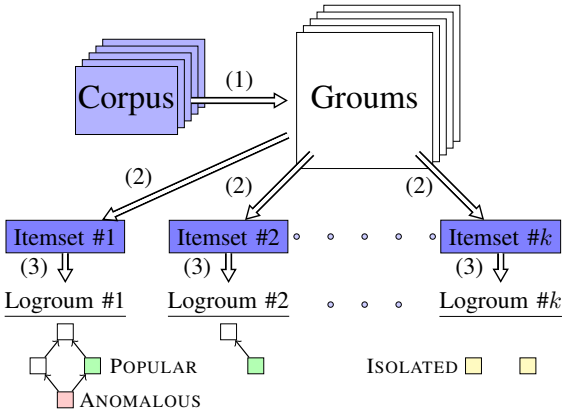


Fig. 1: Overview of the BIGGROUM approach: (1) compile a groum from each app method in the corpus; (2) cluster the groums using frequent itemsets; (3.a) bin the isomorphic groums and order them with the embedding relation ($\preceq$) in a logroum; (3.b) label each bin in the logroum using the frequency of occurrence of groums in each bin and the order relation. The graphs in the picture represents a lattice, where each node is a "bin" and each edge represents the embedding relation $\preceq$. White/green/red/yellow colored bins represent unlabeled/POPULAR/ANOMALOUS/ISOLATED patterns.

tributions, the itemset computation, the lattice-based mining and our subgraph isomorphism computation, are necessary to scale on such large corpora. Then, we show that BIGGROUM is *effective*, by evaluating the precision and recall of the mined patterns. From sampling 15 clusters (out of a total of 194 clusters), we find that 87% of the popular patterns correspond to obligatory or best-practice usage patterns, and none correspond to untrue patterns. Then, out of 15 known Android usage patterns, we can find 11 of them as popular patterns. We also compare BIGGROUM with the GrouMiner tool, in terms of the efficiency and the inferred patterns.

Approach At a Glance: Figure 1 shows a flow diagram of the BIGGROUM approach. BIGGROUM takes as input a corpus of apps that use a common set of APIs. Whereas the approach can generalize to other objected-oriented APIs, we will focus our attention on the Android APIs. Note that the approach directly mines app-code (i.e., the code that uses the APIs), while the Android framework itself *is not an input* of the algorithm. Each app in the corpus consists of multiple classes, and in turn, each class consists of multiple app methods.

(1) BIGGROUM compiles each method in the corpus into a groum (i.e., the approach is intra-procedural), a graph that represents the control flow and data dependencies of the method, sliced and "abstracted" with respect to the API method calls: a groum only contains nodes that represent the API method calls, the control structure of the method and the program variables used in the method calls.

(2) The corpus of groums is clustered by using frequent itemset mining in order to perform the pattern computation on subsets of the entire corpus. Frequent itemset mining computes the set of API method calls (itemsets) where the number of groums (i.e., app defined methods) containing all API method calls of the itemsets exceeds some threshold f_l. The corpus of groums can then be clustered based on the computed frequent itemsets. Each itemset selects the groums that share at least some number K_l of API method calls.

(3) BIGGROUM then mines the patterns from each cluster of groums. (3.a) The groums in each cluster first binned according to the embedding relation $\preceq$, computed amongst each pair of groums in the cluster (isomorphic groums belong to the same bin, and a bin is subsumed by another bin if the groums that it contains are embedded in the groums of the other bin). (3.b) The bins in the lattice are labeled POPULAR if the number of groums in the bin exceeds some threshold f; ANOMALOUS and ISOLATED labels are applied to bins below some threshold L subject to a relation with popular bins (formally defined in Section V). The output of BIGGROUM are the isomorphic groums in the labeled bins that represent POPULAR, ANOMALOUS, or ISOLATED API usage patterns.

A key property of our approach is that we can reify our labeled bins as groums that potentially explain a usage pattern. While statistical model-based approaches (e.g., based on hidden markov model [16] or n-grams [17]) may be used for predictive tasks like code completion, they do not readily provide artifacts that could be human interpretable.

II. GROUM MINING

Groums: In this section, we define the groum data structure that will be used to represent usage patterns, the embedding relation between two groums and the groum mining problem.

Our definition of groum mostly follows that of Nguyen et al, with modifications used to ease the presentation of our approach to mining groums [15]. An API signature is defined by a set of object types $O = \{o_1, \ldots, o_k\}$ and methods $M = \{f_1, \ldots, f_K\}$. Each method f_i is associated with a method signature that includes (a) tuple of argument types $(o_{i,1}, \ldots, o_{i,j})$, (b) a return type o_i and (c) a receiver type $o_{i,r}$. In practice, we may also include additional information such as the set of exceptions thrown by a method. However, we will elide these details for simplicity of presentation.

Definition 1 (Groum [15])**.** *A* groum *is a labeled graph G with nodes V and edges $E \subseteq V \times V$.*

The set of nodes V are partitioned into three types, data nodes V_d*,* control nodes V_c *and* method call nodes V_m*:*

1) *Each data node $v \in V_d$ has an associated type $\tau(v) \in O$;*
2) *Each method call node $v \in V_m$ is associated with an API method call $f_i \in M$;*
3) *Each control node $v \in V_c$ is associated with a statement type such as* if*,* for*,* while *and so on.*

The set of edges E are partitioned into three types, use-edges $E_u \subseteq V_d \times V_m$*,* def-edges $E_d \subseteq V_m \times V_d$*, and* control-flow edges $E_c \subseteq (V_c \cup V_m) \times (V_c \cup V_m)$*:*

1) *Use-edges point from a data node to the method call node where the particular data type is used;*

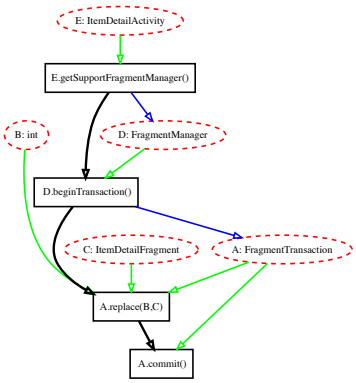


Fig. 2: Groum showing the usage of the Android's object `FragmentTransaction`. The dashed oval nodes represent data nodes. The boxes show method nodes with corresponding API calls. The use edges are shown in green whereas the def edges are shown in blue. Control flow edges are shown in black, while transitive edges are not shown for readability.

2) *Def-edges point from a method call node to a data node whenever the return value of the associated API method is assigned to the data node pointed to;*
3) *Control-flow edges relate each method call or control node to the locations that the control may visit next.*

We will refer to the *transitive closure of control flow edges* as the relation E_t such that $(s,t) \in E_t$ if either $(s,t) \in E_c$, or $(u,t) \in E_c$ and $(s,u) \in E_t$ (i.e., there is a control flow path from s to t). While we do not represent explicitly in a groum G the edges defined by E_t (i.e. $E_t \not\subseteq E$), we will use them and refer to them as *transitive edges*. Figure 2 shows an example of a groum.

Definition 2 (Embeddings)**.** *Given two groums $G_1 : (V_1, E_1)$ and $G_2 : (V_2, E_2)$, we say that G_1 is embedded into G_2, written $G_1 \preceq G_2$ iff there exists a mapping $\pi : G_1 \mapsto G_2$ that maps nodes $v \in V_1$ to $\pi(v) \in V_2$ and edges $e \in E_1$ to $\pi(e) \in E_2$ such that the following conditions hold:*

1) *π is one to one but not necessarily onto. In other words, every node $v \in V_1$ is mapped to a unique node in $\pi(v) \in V_2$ and every edge in $e \in E_1$ is mapped to a unique edge $\pi(e) \in E_2$. However, there may be unmapped nodes in V_2 and unmapped edges in E_2.*
2) *π preserves the type of the nodes: it maps data nodes to data nodes and so on. Furthermore,*
   - *For each data node v, $\pi(v)$ has the same object type.*
   - *For each method node v, $\pi(v)$ is associated with the same API method call as v.*
   - *For each control node v, $\pi(v)$ has the same control label.*
3) *π maps def-edges to def-edges, and use-edges to use-edges.*
4) *For control edges π maps a control-flow edge $e \in E_{1,c}$ to a transitive edge $\pi(e) \in E_{2,t}$.*

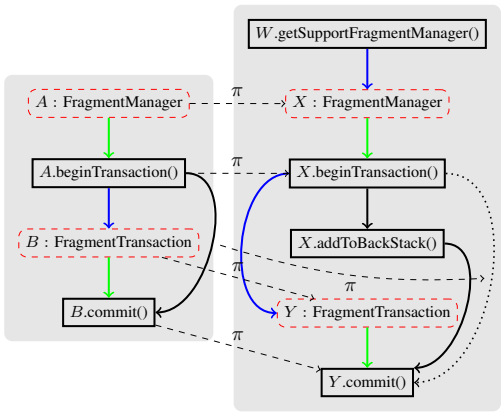


Fig. 3: Example of an embedding between groums: the left groum is embedded in the right groum. The black, dotted arrow in the right groum is a transitive control edge (for clarity we do not show other transitive control edges). Each node of the smaller groum is mapped onto a node of the larger one, the mapping is shown with black dashed arrows labeled with π. In the figure we do not show the mapping on the edges, apart the one from a control edge (the edge from node `A.beginTransaction()` to node `B.commit()`) to a transitive edge (the edge from node `X.beginTransaction()` to node `Y.commit()`), showing that control flow edges can map onto transitive edges.

5) *If $\pi(s_1, t_1) = (s_2, t_2)$ for edges $(s_1, t_1) \in E_1$ and $(s_2, t_2) \in E_2$, we have $\pi(s_1) = s_2$ and $\pi(t_1) = t_2$.*

Two graphs G_1 and G_2 are *isomorphic* written $G_1 \equiv G_2$, iff $G_1 \preceq G_2$ and $G_2 \preceq G_1$.

Figure 3 shows the embedding between the groum on the left side and the groum on the right side.

The problem of deciding if $G_1 \preceq G_2$ is NP-complete, and thus reducible to solving a SAT problem [18]. However, due to the power of modern SAT solvers, and the presence of helpful labels on the graphs, we present a SAT encoding that is effective on groums with hundreds of nodes and edges. The problem of checking if $G_1 \preceq G_2$ is presented in Section IV.

Definition 3 (Groum mining problem)**.** *Stated in its simplest form, the problem of mining groums takes as an input a* corpus *$\mathcal{C} : \{H_1, \ldots, H_{N_g}\}$ of N_g groums, where N_g is assumed to be a large number, and a frequency threshold f. The output of the mining problem is the set of all the groums $\mathcal{P} : \{G_1, \ldots, G_m\}$ such that for each G_i, $|\{H_j \mid G_i \preceq H_j\}| \geq f$ (i.e., an element in $\mathcal{P}$ is a groum that is embedded in at least f groums).*

Similarly, the approach also identifies *anomalous* members of the corpus that contain API usage patterns incompatible with the popular patterns in $\mathcal{P}$ (see Section V).

III. Groum Clustering with Frequent Itemsets

Frequent itemsets are a popular approach in data mining [19] to collect a set of items that occur together in more than f_l records, wherein f_l is a threshold set by the user. Many approaches to mining API usages have utilized this technique

in the past. Here, we briefly summarize what frequent itemsets are, and how they serve to cluster a given corpus of groums into smaller subsets, which can be mined more efficiently.

Let $\mathcal{C}$ be a corpus of groums over an API with methods M. With each $H_i \in \mathcal{C}$, we identify a *record* $m(H_i) \subseteq M$ as the set of API methods calls that occur in H_i. In other words, $m(H_i)$ simply collects the set of methods called, without representing the number of calls or the control flow between them. The groum in Fig. 2 has the itemset with the methods {`getSupportFragmentManager`, `beginTransaction`, `replace`, `commit`}.

Definition 4 (Itemsets and Frequencies)**.** *An* itemset $I \subseteq M$ *is a subset of API methods. Its frequency* $\#(I)$ *is defined as the number of groums* H_j *in the corpus* $\mathcal{C}$ *that contain all the methods in the itemset:* $\#(I) := |\{H_j \in \mathcal{C} \mid I \subseteq m(H_j)\}|$.

Definition 5 (Frequent Itemset Mining problem)**.** *The* frequent itemset mining problem *takes as input a corpus* $\mathcal{C}$*, an API with set of methods* M*, and a frequency threshold* f_l*, and computes the set of all frequent itemsets* $I_1, \ldots, I_k$*, such that* $\#(I_j) \geq f_l$ *for each itemset* I_j *in the list.*

The problem of mining frequent itemsets has been very well studied with efficient algorithms that scale for large corpora. (e.g., see the the original work [19]). We use frequent itemset to cluster co-occurring sets of groums from the corpus.

Definition 6 (Clusters)**.** *A groum* $H_i \in \mathcal{C}$ *belongs to the cluster defined by a frequent itemset* I_j *iff* H_i *has* K_l *or more methods in common with* I_j*, wherein* K_l *is a fixed threshold parameter.*

IV. Computing Embeddings

SAT Encoding: Given two groums G_1, G_2, we seek to check if G_1 is embedded into G_2 ($G_1 \preceq G_2$) as defined in Def. 2. This forms a core *primitive* of our overall approach.

Let $G_1 : (V_1, E_1)$ and $G_2 : (V_2, E_2)$ be the two graphs. We wish to check if $G_1 \preceq G_2$. Furthermore, we partition the vertices and edges as in Def. 1. For convenience, we compute the transitive closure of the control edges $E_{2,t}$ of G_2.

We introduce a series of Boolean variables $p(v_1, v_2)$ for each pair $v_1 \in V_1$ and $v_2 \in V_2$, and $p(e_1, e_2)$ for each pair of edges $e_1 \in E_1$ and $e_2 \in E_2 \cup E_{2,t}$. A variable $p(v_1, v_2)$ ($p(e_1, e_2)$ encodes that the node v_1 (the edge e_1) is mapped to the node v_2 (the edge e_2). Formally, $p(v_1, v_2)$ ($p(e_1, e_2)$ is true if $\pi(v_1) = v_2$ ($\pi(e_1) = e_2$).

In the following, we now define a propositional logic formula that is satisfiable iff $G_1 \preceq G_2$.

Type Matching: First we note that only nodes of the same type can be embedded. Let $\text{compNodes} \subseteq V_1 \times V_2$ represent all the *compatible* nodes, wherein two nodes $v_1 \in V_1$ and $v_2 \in V_2$ are compatible iff the following conditions hold:

1) they are of the same node type;
2) if they are method nodes, they have the same API method;
3) if they are object nodes, they have the same object type;
4) if they are control nodes, they have the same control type.

All other nodes are deemed incompatible. To enforce node compatibility, we define the following formula:

$$\psi_1 := \bigwedge_{(v_1,v_2) \notin \text{compNodes}} \neg p(v_1, v_2)$$

Likewise, we define the set of compatible edge pairs $\text{compEdges} \subseteq E_1 \times (E_2 \cup E_{2,t})$, wherein $(e_1, e_2) \in \text{compEdges}$ iff:

1) if e_1 is a def edge, then so is e_2 (and vice-versa);
2) if e_1 is a use edge, then so is e_2 (and vice-versa);
3) if e_1 is a control edge, then $e_2 \in E_{2,t}$ (and vice-versa);
4) if $e_1 : (s_1, t_1)$ and $e_2 : (s_2, t_2)$ then $(s_1, s_2) \in \text{compNodes}$ and $(t_1, t_2) \in \text{compNodes}$.

We encode edge compatibility with the formula:

$$\psi_2 := \bigwedge_{(e_1,e_2) \notin \text{compEdges}} \neg p(e_1, e_2)$$

One-to-One Mapping: For a set of Boolean variables $P : \{p_1, \ldots, p_m\}$, the formula $\text{One}(P)$ states that exactly one of the variables in P is true. It is defined by two formulas $\text{atLeastOne}(P)$ and $\text{atMostOne}(P)$:

$$\text{One}(P) := \underbrace{\bigvee_{i=1}^{m} p_i}_{\text{atLeastOne}(P)} \bigwedge \underbrace{\bigwedge_{1 \leq i < j \leq m} \neg(p_i \wedge p_j)}_{\text{atMostOne}(P)}$$

We encode that each node in V_1 is mapped to exactly one node in V_2:

$$\psi_3 := \bigwedge_{v_i \in V_1} \text{One}\left(\{p(v_i, v_j) \mid v_j \in V_2\}\right)$$

Likewise, each node in V_2 can be mapped to at most one node in V_1:

$$\psi_4 := \bigwedge_{v_j \in V_2} \text{atMostOne}\left(\{p(v_i, v_j) \mid v_i \in V_1\}\right)$$

We define similar formulas for edges, where each edge $e_i \in E_1$ is mapped to an edge in $e_j \in E_2 \cup E_{2,t}$:

$$\psi_5 := \bigwedge_{e_i \in E_1} \text{One}\left(\{p(e_i, e_j) \mid e_j \in E_2 \cup E_{2,t}\}\right)$$

At most one edge mapped onto each edge $e_j \in E_2 \cup E_{2,t}$:

$$\psi_5 := \bigwedge_{e_j \in E_2 \cup E_{2,t}} \text{atMostOne}\left(\{p(e_i, e_j) \mid e_i \in E_1\}\right)$$

Node/Edge Compatibility: We encode that two edges are mapped only if their nodes are mapped:

$$\psi_6 := \bigwedge_{(e_1,e_2) \in E_2 \cup E_{2,t}} p(e_1, e_2) \Rightarrow (p(s_1, s_2) \wedge p(t_1, t_2))$$

Full SAT Encoding: The overall formula ψ is defined as

$$\psi := \psi_1 \wedge \psi_2 \wedge \psi_3 \cdots \wedge \psi_6$$

Theorem 1. *ψ is satisfiable if and only if $G_1 \preceq G_2$.*

Proof. Proof simply compares the clauses in ψ and Def. 2. □

Effective Filtering: Modern SAT solvers can solve large problems with hundreds of thousands of variables and millions of clauses. Furthermore, widely available implementations such as miniSAT have enabled their use in many application areas [20], [21]. Nevertheless, for large graphs the problem of checking the satisfiability of the embedding encoding is not practical. However, there are many optimization that can be performed to avoid the use of a solver in the first place, and reduce the size of the encoding. To this end, we design *filters* that can check if $G_1 \not\preceq G_2$. When these checks determines that $G_1 \not\preceq G_2$, we can avoid the expensive call to the SAT solver. Furthermore, these checks allow us to simplify the size of the SAT problem, by eliminating variables and clauses.

We propose the following filters and simplifications. **(a) Node count:** If G_1 has more data nodes than G_2 then $G_1 \not\preceq G_2$. The same considerations hold for control and method nodes and similarly for edges of various types. Thus, a simple count of number of nodes and edges of various types can sometimes eliminate the possibility of an embedding. **(b) Node compatibility:** For each node $v_1 \in V_1$, we compute all compatible nodes $v_2 \in V_2$ such that $(v_1, v_2) \in$ COMPNODES. If no such nodes v_2 can be found for some v_1, then we conclude that $G_1 \not\preceq G_2$. Furthermore, we do not need to create Boolean variables corresponding to the pairs $(v_1, v_2) \notin$ COMPNODES. **(c) Edge compatibility:** For each edge $e_1 \in E_1$, we compute all compatible edges $e_2 \in E_2 \cup E_{2,t}$. If no compatible edges can be found for a given e_1, then $G_1 \not\preceq G_2$, and we avoid to create the Boolean variable corresponding to the pairs $(e_1, e_2) \notin$ COMPEDGES.

Thus, a SAT solver need be called only if all the filters above are unable to rule out an embedding. Furthermore, doing so also drastically simplifies the SAT encoding in our experience.

V. PATTERN MINING AND CLASSIFICIATION

Overview of the mining and classification algorithm: We mine groums from a given sub-corpus C defined by an itemset I, consisting of groums $\{H_1, \ldots, H_n\}$ that have at least some number K_l of methods in common with the itemset I.

The process of mining proceeds in three phases:
(1) Slicing: we apply a standard slicing algorithm to each groum H_i, using the method nodes in the itemset I as *seed*. Slicing removes all the method call in H_i that do not occur frequently. This slicing retains those data nodes that are defined/used by the itemset seed nodes in H_i, and those control nodes that are control dependent ancestors of the seed nodes. Once sliced, we simplify the graph by removing the empty nodes. Let $\{G_1, \ldots, G_n\}$ be the set of sliced groums obtained from $\{H_1, \ldots, H_n\}$ (i.e., G_i is the groum sliced from H_i).
(2) Binning and lattice construction: We group the groums $\{G_1, \ldots, G_n\}$ into bins of isomorphic graphs. We then build a lattice between bins based on the embedding relation ($\preceq$).
(3) Classification: We classify the bins in the lattice as POPULAR, or ANOMALOUS, or ISOLATED according to their position in the lattice and the frequency of the bins.
Binning and Lattice Construction: Given a set of groums $\{G_1, \ldots, G_n\}$, we compute bins that contain sets of isomorphic groums. Each bin $B_i \subseteq \{G_1, \ldots, G_n\}$ collects graphs that are *isomorphic* to each other. Additionally, graphs in two different bins are not isomorphic. For each bin B_j, we choose any graph $G_j \in B_j$ as its *representative*.

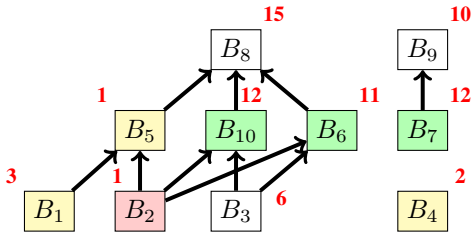


Fig. 4: A logroum: nodes are bins that collect isomorphic groums; edges show the subsumption relation between bins: an edge from the bin B_i to the bin B_j means that $G_i \preceq G_j$, where G_i and G_j are groums contained in B_i and B_j respectively. Transitive edges are not shown for readability. The figure shows the cardinality of each bin in red, and the colors of the nodes represent the classification of the bin: green/red/yellow/white nodes are respectively popular/anomalous/isolated/unclassified patterns.

The groums binning procedure iterates through all the sliced groums $\{G_1, \ldots, G_n\}$. Given a groum G_i, the algorithm:) Compares G_i with the bin representative G_j for each bin B_j created so far (initially, there are no bins).) If G_i is isomorphic to the representative graph G_j of the bin B_j, then G_i is added to B_j. Note that the isomorphic check is performed by the SAT solver by checking if $G_i \preceq G_j$ and $G_j \preceq G_i$.) Otherwise, if G_i is not isomorphic to any existing bin, it forms a new bin on its own. The process terminates when there are no more graphs left to bin, partitioning all the graphs into a set of bins $\{B_1, \ldots, B_l\}$. $|B_i|$ is the cardinality of a bin B_i.

Next we build a *logroum*, a lattice structure where the lattice elements are the bins themselves and two bins B_i and B_j, with representative groums G_i and G_j respectively, are such that $B_i \preceq B_j$ iff $G_i \preceq G_j$. We use the SAT-based encoding presented in the previous section to check if $G_i \preceq G_j$. Figure 4 illustrates a logroum as a directed acyclic graph, showing the cardinality $|B_i|$ for each bin B_i.
Classifying Bins: We then classify the bins in the logroum as POPULAR, ANOMALOUS and ISOLATED. The representative element of a classified bin will then represent a pattern. Our classification scheme uses the exact same principles as the original scheme proposed by Nguyen et al in GrouMiner [15]. However, we use the lattice to formalize our classification.

The classification uses two user provided parameters. f is the minimum desired frequency for a POPULAR patterns. L is the maximum desired frequency for an ANOMALOUS or ISOLATED pattern. We assume $L < f$.

We classify the bins into three categories: (a) POPULAR: patterns that are embedded in at least f groums in the corpus; (b) ANOMALOUS: patterns that are strictly embedded in a POPULAR pattern, but are matched infrequently by at most L other groums in the corpus; (c) ISOLATED: patterns that are

not embedded in a popular pattern and infrequent, matching at most L other patterns in the current corpus.

For each bin B_i, its cardinality is written $|B_i|$ and its *frequency* is defined as $\#(B_i) : \sum_{B_i \preceq B_j} |B_j|$. In lattice terms, the frequency of a bin sums up its own cardinality and that of every bin that is connected to it (for example, in Fig. 4, we calculate $\#(B_1) = |B_1| + |B_5| + |B_8| = 19$ and $\#(B_{10}) = |B_{10}| + |B_8| = 27$).

Definition 7 (Popular, Anomalous and Isolated)**.** *A bin B_j in the lattice is* POPULAR *iff $\#(B_j) \geq f$ and furthermore, for every bin B_k such that $B_j \preceq B_k$ we have $\#(B_k) < f$. In other words, no bin "above" B_j in the lattice is popular. A bin B_k in the lattice is* ANOMALOUS *if $|B_k| \leq L$ and furthermore, it is embedded in a* POPULAR *bin. A bin B_l in the lattice is* ISOLATED *if $|B_k| \leq L$ and furthermore, it is* not *embedded in a* POPULAR *bin or does not embed a popular bin.*

Note that some of the bins may not end up being classified into any of the categories mentioned above: we remain *indifferent* to such bins given the frequency cutoffs chosen.

As an example, we classify the bin of Fig. 4 using $f = 20$ and $L = 4$: the bins $B_{10}, B_6,$ and B_7 are POPULAR. For instance, $\#(B_7) = 22 \geq f$, and furthermore, it is connected to B_9, which is not popular. Likewise, B_2 is ANOMALOUS. For instance, the cardinality $|B_2| \leq L$ and furthermore, B_2 is connected to the popular nodes B_6 and B_{10}. Likewise, B_1, B_4 and B_5 are ISOLATED. Since their cardinalities are below L and they are neither embedded in nor embed a popular bin.

We motivate our choice of POPULAR, ANOMALOUS and ISOLATED patterns. Let us consider a popular and correct usage pattern P and a simplistic "bug model" that mutates a groum applying the following operations, which are commonly seen as the cause of object oriented API misuses [22].

1) *Inserting* an additional method in the groum. An instance of this is the "double free" bug, wherein resources are released twice in some code path.
2) *Deleting* a method in the groum. This mutation is a common source of bugs in many APIs. For instance, forgetting to release a resource.
3) *Rearranging* the order of methods in the groum. These mutations are quite common in use-after-free bugs.
4) *Multiple mutations* (among the three above) can be applied concurrently to the groum.

Table 5 motivates our inclusion of both ANOMALOUS and ISOLATED pattern and summarizes the *expected* effect on the classification, when applying a specific mutation type on a groum corresponding to a popular pattern in the corpus.

Mutation Type	Expected Classification	Expected effect on the logroum The groum is inserted in a bin
Insertion	unlabeled	that is subsumed by P
Deletion	ANOMALOUS	that subsumes P, but is not frequent
Rearranging	ISOLATED	that subsumes P, but is not frequent
Multiple mutations	ISOLATED	that subsumes P, but is not frequent

Fig. 5: Expected classification resulting from specific mutation types of a single groum from a popular pattern.

Performance of BIGGROUM

ID	Description
PERF1	Can BIGGROUM scale on a large corpus better than GrouMiner?
PERF2	What is the impact of the SAT-based embedding on the performance? Is the filtering necessary to achieve scalability?

Quality of the BIGGROUM patterns

ID	Description
COMP	Does BIGGROUM find better POPULAR patterns than GrouMiner?
PREC1	Are the POPULAR patterns mined by BIGGROUM correct?
PREC2	Do the ANOMALOUS and ISOLATED patterns mined by BIGGROUM correspond to real bugs?
REC1	Does BIGGROUM mine known Android patterns?
REC2	Does BIGGROUM mine ANOMALOUS and ISOLATED patterns that correspond to actual bugs to known Android patterns?

Fig. 6: Research questions of the experimental evaluation.

VI. EXPERIMENTAL EVALUATION

We describe our implementation of the ideas presented thus far, our research questions and an evaluation to address them. We then present the results of the evaluation to answer each specific question and the threats to the validity of the results. Throughout this section, we refer to the approach used in this paper as the BIGGROUM, and to the approach of Nguyen et al [15] as GrouMiner. To validate and reproduce the experiments, we provide all the results of the experimental evaluation, the corpus of Android apps, and our tools (with their source code) at the following url https://goo.gl/r1VAgc.

A. Implementation of BIGGROUM

We implement the BIGGROUM approach with different tools, written in Java, Python and C++. BIGGROUM extracts a groum for each method of each class of an Android app. BIGGROUM works directly with Java and Android bytecode using the Soot frontend for Java [23], [24]. The groums are sliced to remove statements and control structures irrelevant to the Android API. BIGGROUM implements the clustering using frequent itemset mining and the construction of bins, the lattice and the resulting classification algorithm as described in Section III and V respectively. The embedding computation uses the Z3 SMT solver [21] as underlying SAT solver. BIGGROUM presents the resulting patterns and anomalies as html pages, to allow a user to examine them.

B. Experimental Evaluation Setup

Research Questions: The main research question we want to address in the experimental evaluation is: "Can we mine patterns of usage as groums for a complex framework such as Android from a heterogeneous, large corpora of apps?". We answer this research question with the questions of Fig. 6.

We first ask if BIGGROUM scales better than GrouMiner when mining a large corpus of groums (PERF1), and the role of the SAT-based embedding check and the filtering on the overall performance of BIGGROUM (PERF2).

We then focus on the quality of the patterns mined by BIGGROUM. The COMP question compares the POPULAR

patterns mined by BIGGROUM and GrouMiner (when it can compute the patterns), in terms of number and quality (e.g., size, frequencies) of the patterns found.

In the PREC1 and PREC2 questions we evaluate the *precision* of the mined patterns. We evaluate the precision of a pattern by assigning it to one of the following categories (i) OBLIGATORY: are common usage patterns that lead to serious defects such as crashes or security vulnerabilities when not respected; (ii) BESTPRACTICES: are common usage patterns that lead to undesirable user experience when not respected; (iii) CUSTOMARY: are common usage patterns that are followed by Android developers to achieve an accepted user experience (e.g., color schemes, windows with titles, notifications and so on); and (iv) UNTRUE: are patterns formed by a purely accidental collocation of methods or weakly related methods. Intuitively, OBLIGATORY, BESTPRACTICES, and CUSTOMARY are all "correct" patterns of usage of the APIs, while UNTRUE patterns are "wrong" patterns of usages. This classification is more fine grained than the one that partitions the patterns as "correct" or "wrong", and further helps to understand how the mined patterns could be used in a client (e.g., a bug detector may only use OBLIGATORY patterns, while code completion may consider all of them).

Finally, questions REC1 and REC2 ask if well documented patterns in Android can be found by BIGGROUM, and is evaluated measuring the recall of the approach. Clearly, we do not know a-priori the patterns represented in our corpus, or all the existing patterns in Android, and hence we compute the recall measure on a subset of known reference patterns.

Setup of the Experiments: We consider a corpus of 542 Android open source apps from GitHub. We crawled GitHub searching for repositories containing Android apps that were rated with at least 5 stars to bias our corpus towards "good quality" apps. Since BIGGROUM works on byte-code, we tried to compile the apps (with the gradlew command), keeping only the ones that compile, for a total of 542 apps. We extracted a total of 70000 groums from each declared method of the 542 apps that contained at least one call to an Android API method.

We obtained the input required by the GrouMiner tool by slicing the app code in the corpus to produce a java code containing only the method of interest and the class members accessed by the method. However, this slicing applied on the source code does not remove calls to app defined methods. In Android apps usually the calls to app methods constitute a small part of a single app, and hence we expected to also obtain patterns for Android APIs from GrouMiner.

We performed a first experiment where we both run GrouMiner and the mining algorithm of BIGGROUM (i.e. BIGGROUM without the clustering phase) on the entire corpus of groums. After 72 hours of execution, both the approaches failed to terminate. This result implies that: (i) the original implementation of GrouMiner [15] *cannot scale* on the corpus of 70000 groums; and (ii) the clustering of groums using the frequent itemset *is necessary* to scale to large corpora of apps.

All the data presented in the rest of the evaluation is

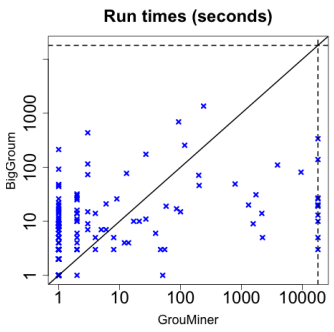


Fig. 7: The plot each point compares the times (seconds) for BIGGROUM (y-axis) and GrouMiner$^+$ (x-axis) to compute a cluster. A point is on the dashed line if the corresponding approach did not terminate before the timeout (300 minutes). GrouMiner$^+$ timed out on 11 clusters.

obtained by first computing the clusters using the frequent itemset computation, and then running the groum mining algorithm of BIGGROUM and GrouMiner on each cluster separately. We refer to this configuration of GrouMiner as GrouMiner$^+$, since it differs from the original approach [15]. We set a timeout of 5 hours for the computation of the patterns of a *single cluster* (i.e., for each cluster, we run GrouMiner$^+$ and BIGGROUM for at most 5 hours). In the experiments, we used the parameters f_l=20, f=20, L=5, K_l=2 (i.e., we create clusters of groums that share 2 or more methods with the corresponding itemsets) for BIGGROUM. We chose these values running BIGGROUM on a smaller corpus of groums.

The frequent itemset computation generated 194 clusters in 60 seconds. The largest cluster had 1730 groums with 22% of the clusters having 100 groums or more. The smallest cluster had 48 groums. Likewise, the largest itemset had 20 Android API methods in it, whereas the smallest itemset had 3 methods.

C. Experimental Results - BIGGROUM Performance

*Performance Comparison with GrouMiner$^+$ (*PERF1*):* In the scatter plot shown in Figure 7 we compare the performance of BIGGROUM and GrouMiner$^+$. Overall, BIGGROUM computed the frequent subgraphs for *all* the clusters in 95 minutes, whereas GrouMiner$^+$ took 413 minutes for 183 out of 194 clusters, timing out for the remaining 11 (the time out is 300 minutes). On average, BIGGROUM computed the patterns for a cluster in 0.5 minutes, while GrouMiner$^+$ took 2.3 minutes (the average for GrouMiner$^+$ is only computed for the clusters where GrouMiner$^+$ did not timed out).

We conclude that BIGGROUM scales better than GrouMiner$^+$. We conjecture that the bottom up mining approach of GrouMiner$^+$ must enumerate a large number of smaller patterns before finding the larger popular patterns, requiring more computational effort. We note that there are also some clusters wherein GrouMiner$^+$ computes the patterns almost immediately (i.e., in less than a second) and faster than BIGGROUM. However, on these clusters BIGGROUM always terminates well within the timeout.

*SAT Solver Performance (*PERF2*):* We compare the total time taken by BIGGROUM for the pattern mining and classification phase with the total time taken by the calls to the SAT solver. We also compare the total number of $\preceq$ checks and the total number of checks that had to call the SAT solver:

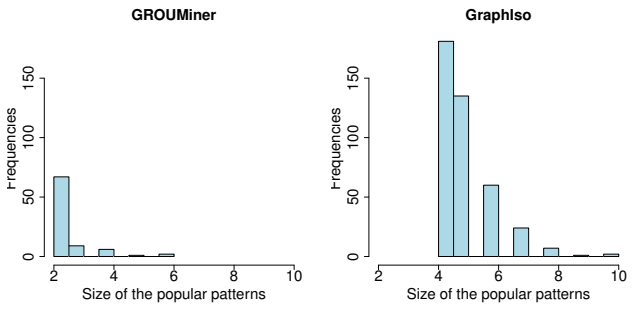


Fig. 9: Distribution of popular pattern sizes (number of API methods) for GrouMiner$^+$ (left) and BIGGROUM (right).

Avg./Max. Graph Sizes	Total Time(s)	SAT Time (s)	Tot. $\preceq$ Checks	$\preceq$ checks Sat solver
186/456	5695	3042	6744259	119250

About 1.77%, a tiny fraction, of the checks had to directly call the SAT solver. At the same time, however, the total time taken by the SAT solver for these calls is about 53.4% of the overall computation time (95 minutes).

D. Experimental Results - Quality of the BIGGROUM *Patterns*

*Pattern Comparison with GrouMiner (*COMP*):* In Fig. 8 we compare the number and sizes of POPULAR patterns found by GrouMiner$^+$ against those found by BIGGROUM, while in Figure 9 we compare the distribution of groum sizes in terms of number of method nodes. GrouMiner$^+$ finds 85 POPULAR patterns, while BIGGROUM finds 410. On average BIGGROUM patterns have 4.9 method nodes versus 2.4 API methods for GrouMiner$^+$. For each GrouMiner$^+$ pattern, we examine if BIGGROUM can find the same or a more complete pattern. BIGGROUM finds 72/85 patterns found by GrouMiner$^+$. We manually examined the remaining 13 patterns to understand why BIGGROUM did not discover them: (i) 7 patterns involved an API method call that *was not part* of a frequent itemset, and thus was sliced away in BIGGROUM (changing the frequency cutoff for popular patterns could address these discrepancies); (ii) 2 patterns contained app specific methods, 2 others contained methods from the Java (and not Android) APIs, and 2 patterns contained methods without a precise type signature.

BIGGROUM finds more patterns than GrouMiner$^+$ for the following reasons: (i) BIGGROUM tracks base types for app classes that inherit from an Android class, enabling us to compare object types across apps, unlike GrouMiner$^+$; (ii) even though we slice the GrouMiner$^+$ input, some of the app specific method calls are left over, nevertheless. These are sometimes popular enough for the given cutoff frequencies.

POPULAR patterns		
Approach	Tot. of patterns	Min./Avg./Max. pattern size
BIGGROUM	410	4/4.9/10
GrouMiner$^+$	85	2/2.4/6

Fig. 8: BIGGROUM vs. GrouMiner$^+$

Precision of the BIGGROUM *Patterns (*PERF1 *and* PERF2*):* To evaluate the precision of the approach we manually inspected the patterns found by BIGGROUM for 30 (out of the 194) randomly selected clusters. We first analyze the POPULAR patterns. We manually assigned the category, OBLIGATORY, BESTPRACTICES, CUSTOMARY and UNTRUE, to the *most* frequent and the *least* frequent POPULAR pattern in the clusters (a cluster may contain several POPULAR patterns). For the OBLIGATORY patterns, we then investigated the other ISOLATED and ANOMALOUS patterns in those clusters to check if they were actual defects.

The bar chart in Figure 10a summarizes the outcome of our manual evaluation for the POPULAR patterns: the first bar in the plot shows the distribution of the patterns with the highest frequency, while the second bar shows the distribution of the patterns with the lowest frequency, both divided in the 4 different categories. The plot shows that the precision of BIGGROUM does not change if we consider patterns with different frequency (i.e., the frequency cutoff f is adequate).

In the following, we discuss the results for the most frequent POPULAR patterns. We found at least one popular pattern in 29 out of the 30 clusters examined. The cluster defined by the methods `setOnClickListener`,`setText` and `setTextColor` failed to have any popular patterns. There is no prescribed order for the three setter methods involved, and further, only a subset of these methods may be called. This yields a large number of possible patterns, none of which exceed our frequency cutoff to be popular.

We found 8/29 OBLIGATORY patterns, and Figure 10b shows one of them: the pattern [25] shows the protocol for opening a database, creating a new value, inserting the value in the database, and closing the database.

Most of the patterns examined (17/29) are BESTPRACTICES that describe code snippets to accomplish a well defined, specific task. Figure 10c shows an example of a BESTPRACTICES pattern to retrieve an Activity toolbar, setting its title and adding a navigation button back to the app's home screen [26]. Clearly, the pattern is used by several apps. A deviation from this pattern does not necessarily cause a serious defect, but may presumably lead to a poorer user experience.

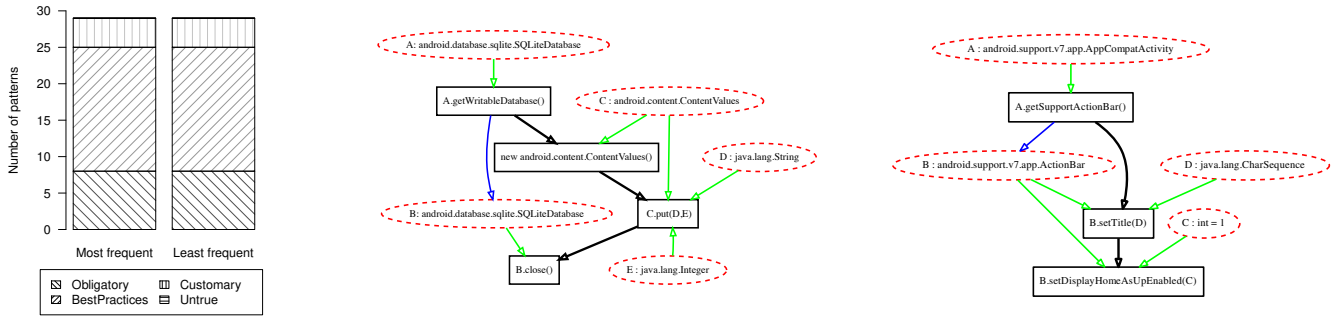
4 patterns out of 29 were categorized as CUSTOMARY. In one of such patterns the `android.util.Log.d` method is frequently called with the method (`putExtra`) used to create and modify an `android.content.Intent` object (used for interprocess communication). It is clear that developers often insert log messages to help them better debug `Intent`s.

We did not find any POPULAR pattern categorized as UNTRUE, although an ongoing thorough examination of all the 410 patterns may provide us such examples.

Next, we manually examined one representative groum (choosen randomly) for each pattern inside the clusters categorized as ANOMALOUS and ISOLATED, searching for violations that could be potential bugs.

Category	Tot. of patterns	Tot. of patterns containing a bug
ANOMALOUS	34	2
ISOLATED	492	21

We see that 6% of the ANOMALOUS and 4% of the ISOLATED patterns correspond to real bugs in the usage of the APIs. We



(a) (b) (c)

Fig. 10: BIGGROUM precision. (a) Categorization of the most and least POPULAR patterns. (b) An OBLIGATORY protocol for working with an `SQLiteDatabase` object in Android. (c) A BESTPRACTICES protocol to set an Activity toolbar.

found several bugs wherein the developer omitted a call to the `close` method on a database object in the protocol shown in Figure 10b [27]. We also encountered several patterns that did not contain a bug: in almost all these cases the database was eventually closed by another method in the same class. These results show a limitation of our current approach. From our manual inspection, the cause of imprecision of our approach is caused by the fact that the groums are obtained from a single method in the app (i.e., the groum extraction is intraprocedural), and hence does not capture the real execution of an app (e.g., what methods are invoked before and after). Considering interprocedural groums is a future research direction.

Recall of the BIGGROUM *Patterns (*REC1 *and* REC2*):* We evaluated the recall of BIGGROUM by considering 15 known "reference patterns". We collected the names of the Android methods contained in our corpus, we selected a subset of them randomly, and we then searched for their usages on the Android documentation and StackOverflow. The list of patterns is reported in Fig. 11, together with their categorization. (we describe them in detail at the evaluation material's link).

To evaluate REC1 we first searched for the occurrence of each reference pattern among the POPULAR patterns discovered by BIGGROUM. Then, we evaluate REC2 by analyzing the ANOMALOUS and the ISOLATED patterns in the same clusters where the reference patterns were found to be popular.

Fig. 11 shows the total number of POPULAR patterns that contains a reference pattern, with their average frequencies, and the number of ANOMALOUS and ISOLATED patterns found in the same clusters (of the POPULAR patterns). We see that BIGGROUM finds at least one POPULAR pattern for 11 out of the 15 reference patterns. However, 4 out of 15 reference patterns did not have any corresponding POPULAR pattern, since there are few instances of these patterns in the corpus and hence they did not pass the frequency cutoff to be labeled as POPULAR. Thus, it seems that we miss 4 reference patterns becaue we do not have enough data in the dataset of apps, and not because BIGGROUM does not mine them. BIGGROUM also finds ANOMALOUS and ISOLATED patterns discovering possible wrong usages of the reference patterns.

	Reference Pattern	Cat.	POP		AN	IS
			T	f	T	T
1	DB Transaction	OBL	1	38	1	0
2	Get/Release Cursor	OBL	8	37.6	7	78
3	Fragment Transaction	OBL	10	66	2	30
4	Show Toast	OBL	19	52	4	99
5	Show/AlertDialog	OBL	20	32	21	250
6	Retrieve/Release Parcel	OBL	1	29	0	0
7	Create/Send Intent	BES	1	26	0	0
8	Retrieve from backstack	BES	0	0	0	0
9	Query ContentProvider	BES	3	35	0	24
10	Insert ContentProvider data	BES	0	0	0	0
11	Update ContentProvider data	BES	0	0	0	0
12	Delete ContentProvider data	BES	0	0	0	0
13	Build/send notification	OBL	3	42.3	0	32
14	Restore Preferences	BES	2	38.5	0	12
15	Edit Preferences	OBL	5	53	2	81

Fig. 11: BIGGROUM recall. Cat.: category of the pattern (OBL: OBLIGATORY, BES: BESTPRACTICES) POP: POPULAR, AN: ANOMALOUS, IS: ISOLATED, T: Total number of patterns matching the reference pattern, f: Average Frequency.

E. Threats to Validity

The choice of the corpus of apps may affect the performance and the quality of the results. We minimized the issue by selecting real, good quality apps (i.e., apps with more than 5 stars on GitHub and that compile). Our evaluation is for Android and we could have different results for other frameworks.

The experiments' settings could also affect the results. First, we observe that GrouMiner was designed to work inside a single project rather than work across a larger corpus. We addressed this by slicing the source code to retain parts relevant to the Android API. Then, we tried to avoid any selection bias on the BIGGROUM parameters choosing the parameters on a small corpus of apps before mining the full corpus and evaluating our technique.

We evaluated the precision on $30/194$ randomly chosen patterns, finding evidence that a vast majority of the BIGGROUM's patterns are OBLIGATORY and BESTPRACTICES. The number of UNTRUE patterns is highly unlikely to be a large percentage, given that none were found in our sample.

We did our best to select the reference patterns in an unbiased way. We recognize that our understanding of the API usage and some of the online sources may in fact be erroneous. Two of the paper's authors collected and validated the reference patterns, consulting additional documentation (e.g. StackOverflow posts, the Android source code) and, in the most uncertain cases, an Android developer. The evaluation of the patterns was manual and hence, prone to the same threats to validity. In this case, three of the paper's authors validated the results. Finally, the classification of the patterns is not formal and hence it is open to interpretations.

VII. Related work

Groum related approaches: Nguyen et al [15] introduce the groum representation and describe the GrouMiner algorithm to mine frequent patterns and anomalous API usages from a dataset of groums. GrouMiner uses an approximate isomorphism check that compares sequences of node labesl in each graph, which is correct in the majority of the cases. Here, we focus just on the differences in mining groums, assuming that GrouMiner's isomorphism check is completely accurate.

GrouMiner's approach builds groums starting from patterns of size 1, and then incrementally extends an existing pattern with a new node and a new edge until any possible extension of a graph does not result in a pattern that is frequent enough. This construction is potentially expensive due to large number of intermediate patterns. Instead, our approach avoids the bottom-up computation by partitioning the groum dataset with the frequent itemsets of API method calls. We then build a precise lattice that describes the groums subsumption relation. Our algorithm scales better in practice, as shown in our results, since it avoids the computation of smaller, non interesting frequent patterns. As GrouMiner, we chose anomalous patterns that are strictly contained inside a popular pattern, but we furhter use a lower cutoff L and consider ISOLATED patterns.

Due to their expressiveness, groums have been successfully used for *API repair* [28], [29], *code completion* [16], [30], [31], [32], and *code migration* [33]. In particular, most of these approaches need as pre-requisite the set of frequent API patterns expressed as groums. BIGGROUM could extends the applicability of these methods to large framework, as Android.

Recent works [31], [32] produce sequences of API calls from groums and use the sequences to train Hidden Markov Models (HMMs) of the API usages. HMMs automate tasks such as code completion, but are inadequate as documentation or code repair (e.g. as in the papers [28], [29]), that needs a model of control flows and data dependencies.

API Mining: According to the survey of Robillard et al [1], API mining techniques are classified by the kinds of properties they produce (*unordered*, *sequential* and *behavioral*). We did not compare experimentally groum with other type of API specifications. In the following, we discuss the main expressive differences between groums and other types of specifications.

Several works [2], [3], [4], [5] produce an unordered set of APIs that is frequently used together by applying frequent association rule mining. These approaches are efficient, but their main weakness is that they cannot capture the control flow (e.g. order of execution of the methods) or data flow in the mined pattern. We use the same frequent itemset computation, but just as a pre-processing step to partition the dataset of groums. Other approaches (e.g. [8], [9], [10], [11], [12]) mine sequences of method calls. Since we focus on groums, we capture expressive patterns that represent the control flow, data dependencies and interaction among multiple objects. In contrast, the previous techniques can only capture sequences of methods that should be called together. Other techniques (e.g. [13], [14]) mine patterns that are specific for a single object, and thus cannot capture the patterns shown in our experimental evaluation (e.g. such as replacing a `Fragment`). The expressiveness of groums increases the cost of mining the patterns: our approach tackles this problem. Some techniques (e.g. [34], [35]) find behavioral patterns, like the preconditions required to invoke a method. These approaches do not capture the control and data dependencies. On the other hand, we do not mine these kinds of invariants.

Tasks solved through API Mining: Several tasks can be solved by first mining the API usages. Examples of these tasks are code completion [16], [16], [30], [17], relevant code search [36], [37], [14], [38], [39] and API repair [28], [29]. In this paper we do not solve these problems, but we provide a more efficient method to compute groum patterns.

Isomorphism computation via SAT: The reductions of the graph isomorphism and embeddings problems to SAT have been explored elsewhere [40], [41], gaining popularity due to the improvements of SAT solvers in the recent years. Our encoding solves the isomorphism problem between groums, where we consider the different kinds of nodes and edges in the definition of the isomorphism. This allow us to obtain a simplified encoding, where we discard possible isomorphisms that do not respect the compatibility of nodes and edges.

VIII. Conclusion

In this paper we tackled the problem of finding framework usage patterns from a large corpora of Android application.

BIGGROUM overcome the scalability issues due to the size of the app corpus clustering the groums using frequent itemset mining, and building a lattice that represents the embedding relationship among groums. Furthermore, we show that simple filtering techniques reduce the cost of the embedding computation. We presented a detailed experimental evaluation of BIGGROUM, demonstrating its scalability and the quality of the mined patterns. Our future work will focus on overcoming some limitations of the approach, like using interprocedural analysis, and on code completion and automatic repair.

Acknowledgments: This material is based on research sponsored in part by DARPA under agreement number FA8750-14-2-0263. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

REFERENCES

[1] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated api property inference techniques," *IEEE Trans. Softw. Eng.*, vol. 39, no. 5, pp. 613–637, May 2013.

[2] A. Michail, "Data mining library reuse patterns in user-selected applications," in *14th IEEE International Conference on Automated Software Engineering*, Oct 1999, pp. 24–33.

[3] T. Zimmerman, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. on Software Engg.*, vol. 31, 2005.

[4] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining source history," *IEEE Trans. on Software Engg.*, vol. 30, no. 9, 2004.

[5] J. Montandon, H. Borges, D. Felix, and M. Valente, "Documenting APIs with examples: lessons learned with the APIMiner platform," in *Working Conference on Reverse Engg. (WCRE)*. IEEE, 2013, pp. 401–408.

[6] H. S. Borges and M. T. Valente, "Mining usage patterns for the android API," *PeerJ Computer Science*, vol. 1, p. e12, 2015.

[7] Y. Lamba, M. Khattar, and A. Sureka, "Pravaaha: Mining android applications for discovering api call usage patterns and trends," in *Proceedings of the 8th India Software Engineering Conference*, ser. ISEC '15. New York, NY, USA: ACM, 2015, pp. 10–19.

[8] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *ICSE*, 2006, pp. 282–291.

[9] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *ESEC-FSE*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 35–44.

[10] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *ECOOP*, ser. Genoa. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 318–343.

[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the api jungle," *SIGPLAN Not.*, vol. 40, no. 6, pp. 48–61, Jun. 2005.

[12] S. Sankaranarayanan, S. Chaudhuri, F. Ivancic, and A. Gupta, "Dynamic inference of likely data preconditions over predicates by tree learning," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, 2008, pp. 295–306.

[13] R. Alur, P. Cerný, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, 2005, pp. 98–109.

[14] H. Peleg, S. Shoham, E. Yahav, and H. Yang, "Symbolic automata for static specification mining," in *SAS*, 2013, pp. 63–83.

[15] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *ESEC/FSE'09*. ACM, 2009, pp. 383–392.

[16] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *ICSE 2012*, 2012, pp. 69–79.

[17] V. Raychev, M. T. Vechev, and E. Yahav, "Code completion with statistical language models," in *PLDI*, 2014, pp. 419–428.

[18] M. R. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-Completeness*. W.H.Freeman, 1979.

[19] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. Very Large Data Bases(VLDB'94)*, 1994, pp. 487–499.

[20] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[21] L. M. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, ser. LNCS, vol. 4963. Springer, 2008, pp. 337–340.

[22] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: a benchmark for api-misuse detectors," in *MSR*, 2016, pp. 464–467.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *CASCON'99*. IBM Press, 1999, pp. 13–.

[24] P. Lam, E. Bodden, O. Lhoták, and L. Hendren, "The soot framework for java program analysis: a retrospective," cf. https://patricklam.ca/papers/11.cetus.soot.talk.pdf.

[25] "StackOverflow on getWritableDatabase," http://stackoverflow.com/questions/35068292/do-i-need-call-close-when-i-use-getwritabledatabase, 2017, accessed: 2017-02-01.

[26] "Android documentation on setDisplayAsUpEnabled," https://developer.android.com/training/implementing-navigation/ancestral.html\#NavigateUp, 2017, accessed: 2017-02-01.

[27] "StackOverflow bug due to missing close on SQLiteOpenHelper," http://stackoverflow.com/questions/4464892/android-error-close-was-never-explicitly-called-on-database, 2017, accessed: 2017-02-01.

[28] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," *SIGPLAN Not.*, vol. 45, no. 10, pp. 302–321, Oct. 2010.

[29] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 315–324.

[30] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *ICSE 2015*, 2015, pp. 858–868.

[31] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning API usages from bytecode: a statistical approach," in *ICSE 2016*, 2016, pp. 416–427.

[32] ——, "Recommending API usages for mobile apps with hidden markov model," in *ASE 2015*, 2015, pp. 795–800.

[33] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining api usage mappings for code migration," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 457–468.

[34] C. Flanagan and K. R. M. Leino, "Houdini, an annotation assistant for esc/java," in *FME*, 2001, pp. 500–517.

[35] S. Sankaranarayanan, F. Ivancic, and A. Gupta, "Mining library specifications using inductive logic programming," in *ICSE*, 2008, pp. 131–140.

[36] A. Mishne, S. Shoham, and E. Yahav, "Typestate-based semantic code search over partial programs," in *OOPSLA*, 2012, pp. 997–1016.

[37] H. Peleg, S. Shoham, E. Yahav, and H. Yang, "Symbolic automata for representing big code," *Acta Inf.*, vol. 53, no. 4, pp. 327–356, 2016.

[38] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Sci. Comput. Program.*, vol. 79, pp. 241–259, 2014.

[39] R. Holmes, R. J. Walker, and G. C. Murphy, "Strathcona example recommendation tool," in *ECE-FSE*, 2005, pp. 237–240.

[40] I. Olmos, J. A. Gonzalez, and M. Osorio, "Reductions between the subgraph isomorphism problem and hamiltonian and sat problems," in *Electronics, Communications and Computers, 2007. CONIELECOMP'07. 17th International Conference on*. IEEE, 2007, pp. 20–20.

[41] B. Das, P. Scharpfenecker, and J. Torán, *Succinct Encodings of Graph Isomorphism*. Cham: Springer International Publishing, 2014, pp. 285–296. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-04921-2_23