Scaling Code Pattern Inference with Interactive What-If Analysis

Hong Jin Kang
hjkang@cs.ucla.edu
University of California, Los Angeles

Kevin Wang
kwang1083@g.ucla.edu
University of California, Los Angeles

Miryung Kim
miryung@cs.ucla.edu
University of California, Los Angeles

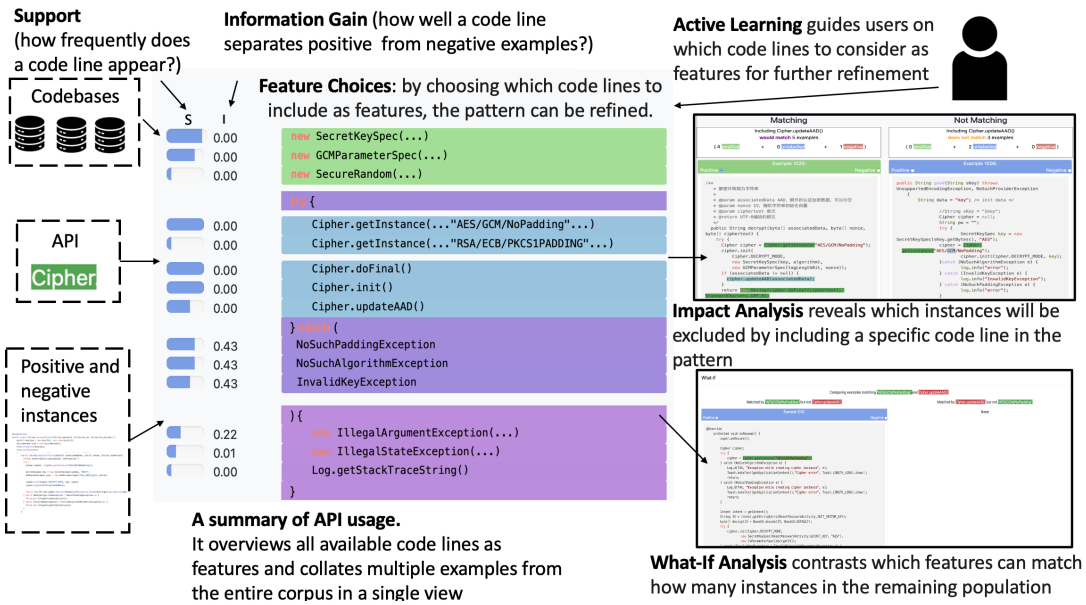

Figure 1: To guide incremental pattern construction and refinement, SURF **summarizes the global distribution of how individual features appear in the entire population in a single collated view. In** SURF**, users can directly provide code-line level feedback as features in addition to labeling positive and negative instances.** SURF **then re-generates a refined pattern. Users can contrast the impact of different feature choices using *impact analysis* and *what-if-analysis*. It visualizes how a specific feature choice is consistent with already labeled positive and negative instances and can match more instances in the unlabelled population.**

ABSTRACT

Programmers often have to search for similar code when detecting and fixing similar bugs. Prior active learning approaches take only instance-level feedback, i.e., positive and negative method instances. This limitation leads to increased labeling burden, when users try to control generality and specificity for a desired code pattern.

We present a novel feedback-guided pattern inference approach, called SURF. To reduce users' labelling effort, it actively guide users in assessing the implication of having a particular feature choice in the constructed pattern, and incorporates direct feature-level feedback. The key insight behind SURF is that users can effectively select appropriate features with the aid of *impact analysis*. SURF provides hints on the global distribution of how each feature is consistent with already labelled positive and negative instances, and how selection of a new feature can yield additional matching instances. Its *what-if-analysis* contrasts how different feature choices can include (or exclude) more instances in the rest of the population.

We performed a user study with 14 participants, designed with two-treatment factorial crossover. Participants were able to provide 30% more correct answers about different API usages in 20% less time. *All* participants found that *what-if-analysis* and *impact analysis* are useful for pattern refinement. 79% of the participants were able to produce the correct, expected pattern with SURF's feature-level guidance, as opposed to 43% of the participants when using the baseline with instance-level feedback only. SURF is the first approach to incorporate feature-level feedback with automated *what-if* analysis to empower users to control the generality (/ specificity) of a desired code pattern.

ICSE 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

active learning, code search patterns, API misuse, human feedback

ACM Reference Format:
Hong Jin Kang, Kevin Wang, and Miryung Kim. 2024. Scaling Code Pattern Inference with Interactive What-If Analysis. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Many software engineering tasks require searching for similar code, e.g., code search [19], code recommendation [26], applying similar fixes [21, 27], and detecting API misuses [17]. Completely automatic techniques face limitations, such as assumptions about frequent patterns [17, 24]. Therefore, the use of human feedback, such as *active learning* approaches, has emerged as a promising direction. Recent studies have shown the effectiveness of human-in-the-loop approaches to synthesize code patterns [14, 17, 24, 36].

Active learning approaches [14, 17, 24] usually require users to provide positive and negative function instances, e.g. the code snippets in Figure 3. However, to accurately label an instance, human users have to disambiguate similar but different instances, often from an unfamiliar codebase, demanding a high amount of human inspection and analysis effort. As human cognition is the bottleneck, scaling up active learning approaches poses several challenges. Firstly, there are not many labelled instances apriori. Second, the user has to iteratively assess more positive and negative instances, which can be challenging. Third, people tend to use specific code lines as a feature, such as the string constant `StandardCharsets.UTF_8` in the positive instance shown in Figure 3a) , but cannot easily provide such granular feedback. Moreover, they do not always know which code line is suitable for a desired feature (for refining the pattern in Figure 3 , a user may wonder "is the method call, `updateAAD` present in all positive instances?", or "if I include `updateAAD` instead of `Base64.decode`, how many positive instances would I no longer find?"). These limitations underscore the need to actively guide users in providing feedback.

We propose an interactive active learning approach, SURF (**S**caling **U**p inte**R**active **F**eedback), seen in Figure 1. Figure 2 shows the workflow of how a user interacts with SURF. Initially, the user provides some positive and negative instances, allowing SURF to bootstrap an initial pattern. SURF allows users to provide direct feedback on which code-line should be considered as a feature, which is used to iteratively re-infer another pattern. SURF guides the user in assessing which code lines to use as feedback. First, SURF provides an overview of the global distribution and overlays code lines from different instances as a summary skeleton. Second, it provides hints on the global distribution by computing importance metrics for each code line. Third, it visualizes the instances that are included or excluded for each choice of feedback on the pattern.

In this study, by "instances," we mean method-level locations in the population to be matched or unmatched by a given pattern. By "features," we mean individual code statements, i.e., nodes in a program dependence graph. Including a feature specializes the current pattern by adding a new requirement.

Suppose Alice uses SURF to construct a code pattern. After providing some positive and negative instances (Figure 3a, 3b, 3c), SURF infers a pattern. The inferred pattern, shown in Figure 3d, does not precisely match Alice's intent—the pattern does not capture the method call `Cipher.getInstance("AES/GCM/NoPadding")`, and

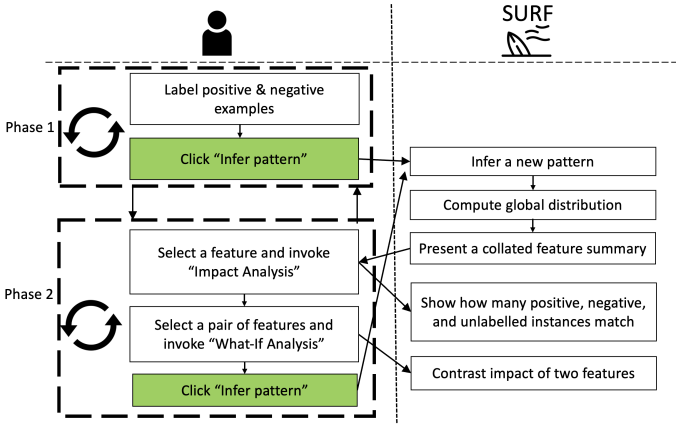

Figure 2: After a user labels positive and negative instances, SURF **infers an initial pattern, computes the global distribution, and presents a collated feature summary. By leveraging feature-level guidance from** SURF**, the user chooses a specific feature and** SURF **constructs a new pattern using Algorithm 1. Users can alternate Phase 1 and Phase 2 to iterate instance-level and feature-level feedback.**

would match programs such as Figure 3e that employ a different transformation, e.g., `Cipher.getInstance("DES")`). Without SURF, Alice provides a new negative instance, Figure 3e, that the pattern should exclude. This may not be enough information to identify the intended pattern due to ambiguity from the multiple possible code lines that can be included in the pattern (e.g., code highlighted in green in Figure 3a). Including any one of `Cipher.getInstance("AES/GCM/NoPadding")`, `Base64.decode`, and `updateAAD` would exclude Figure 3e.

With SURF, Alice provides direct code line-level feedback (e.g. selecting `"AES/GCM/NoPadding"`). To determine which code line should be selected, SURF helps Alice to understand the API usage and its distribution, summarized as a code skeleton that overlays code lines from different repositories. The collated view allows simultaneous inspection of multiple code examples, instead of analyzing each program one by one.

Guided by the importance metrics computed for each code line (Figure 4 Ⓐ), Alice notices code lines such as `new SecureRandom()` have a low support (indicated by the small blue bars under **S**upport). In other words, this `new SecureRandom()` feature occurs infrequently and is unlikely to be useful. Several code lines stand out to Alice due to their high **I**nformation gain. For example, `new SecretKeySpec(...)` has an information gain of 1.0, indicating improvement over the current pattern in separating positive and negative instances.

Due to the small number of positive and negative instances, information gain alone is not enough to identify a good code line. SURF provides *impact analysis*. For each choice, SURF immediately demonstrates which instances will be included or excluded if a particular code line is included in the pattern. This makes it easy for Alice to assess if including a code line, e.g., `updateAAD`, would overspecialize the pattern. Finally, SURF offers a *what-if analysis*, seen in Figure 6. SURF contrasts the impact of the two choices.

```java
try {
  Cipher cipher =
      Cipher.getInstance(
      "AES/GCM/NoPadding");
  ...
  cipher.init(Cipher.
      DECRYPT_MODE, key, spec);
  cipher.updateAAD(
      associatedData);
  return new String(
    cipher.doFinal(
      Base64.decode(cipherText)
    ),
    StandardCharsets.UTF_8);
}
catch (NoSuchAlgorithmException e)
{
  ...
}
```

(a) Positive instance 1: `decryptToString`

```java
Cipher cipher =
  Cipher.getInstance(
    "AES/GCM/NoPadding");

cipher.init(2, new
    SecretKeySpec(
  key.getBytes(), "AES"),
  new GCMParameterSpec(128,
      nonce.getBytes()));

if (StringUtils.isNotEmpty(
    associatedData)) {
  cipher.updateAAD(
    associatedData.getBytes());
}

ciphertext = parseObject(
  new String(
    cipher.doFinal(decode),
    StandardCharsets.UTF_8));
```

(b) Negative instance 1. `wxCallBack`

```java
try {
    Cipher cipher =
        Cipher.getInstance("AES/
        GCM/NoPadding");
    ...
    cipher.init(Cipher.
        DECRYPT_MODE, key,
        params);
    ...
    cipher.updateAAD(ciphertext
        , 0, ciphertextOffset);
    ...
    return cipher.doFinal(
        ciphertext, ...);
}
catch (NoSuchAlgorithmException e)
{
  ...
}
...
```

(c) Negative instance 2.`decrypt`

```java
try {
  // Guard
    Cipher.getInstance()
    new String(...StandardCharsets.UTF_8...)
    // Post-Method Call
} catch (
  NoSuchAlgorithmException
) {
    // Error Handling
}
```

(d) The initial pattern inferred by SURF matches (a) but not (b) and (c).

```java
try {
    Cipher cipher = Cipher.getInstance("DES");
    ...
    cipher.init(Cipher.DECRYPT_MODE, key);
    ...
    return new String(
       cipher.doFinal(encryptedBytes, ...),
       StandardCharsets.UTF_8);
} catch (NoSuchAlgorithmException e) {
  ...
}
...
```

(e) A new negative instance that the inferred pattern should not match. The initial pattern in Figure 3d can match (e), which is undesirable due to weak encryption using `"DES"`

Figure 3: Figure 3d shows an initial pattern matching Figures 3a, not 3b and 3c. This pattern (d) would incorrectly match Figure 3e. Instead of requiring a user to label more instances to exclude Figure 3e, SURF actively guides which feature among three green choices "`"AES/GCM/NoPadding"`", "`updateAAD`", and "`Base64.decode`" to select for pattern refinement.

We conducted a user study with 14 participants, including professional developers from industry. The participants provided feedback to construct a code search pattern in two tasks. We built a baseline by simulating the usage of a prior technique [17], in which users had to inspect individual programs and provide labels. Using SURF, participants demonstrated better understanding of the API usage distribution. They correctly answered 30% more usage comprehension questions and, when given the hints of what a target pattern should look like, they were able to guide the tool to infer 1.8X more correct patterns in 21.9% less time. Participants appreciated how SURF immediately provided feedback in contrasting their choices of code lines. The participants indicated that they were more confident using SURF, while they were overwhelmed and struggled to make sense of the API usages without SURF.

Without SURF, participants spent more time coming up with adhoc criteria for classifying positive vs. negative instances, and were unable to construct a code pattern matching the given API usage description. This confirms the hypothesis that instance-level feedback for active learning is inherently inadequate, as it burdens the user to manually classify unfamiliar code instances.

In summary, this paper makes the following contributions:

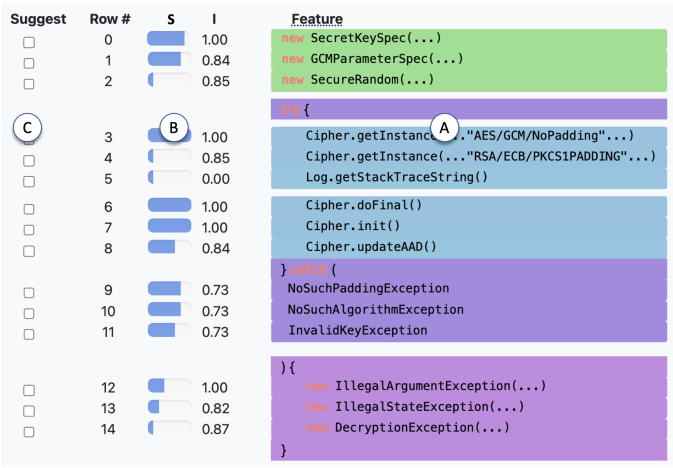

Figure 4: Ⓐ All features are overlaid with one another under a single code skeleton view. A method invocation such as `Cipher.getInstance()` is a feature. The features are structurally grouped, with each group distinguished from another by a different background color. Ⓑ Each code line feature f has *support (S)* how many instances include f and *information gain (I)* —how well f separates already labelled positive and negative instances. These scores guide a user to grasp the distribution of individual features in the entire population. Ⓒ A user can click on each feature's check box to include it in the pattern.

(1) SURF is an interactive active learning approach towards code pattern synthesis. SURF enables direct code line feedback, allowing 1.8X more participants to infer correct patterns.
(2) SURF helps users disambiguate similar API usages through three features: (a) a code skeleton summarizing the population, (b) importance metrics for individual code lines as features, and (c) impact analysis and what-if analysis.
(3) In a user study, participants using SURF demonstrated greater understanding of the global API usage distribution and required less time to construct the expected patterns.

The rest of this paper is organized as follows. Section 2 describes a usage scenario of SURF. Section 3 introduces SURF. Section 4 presents the design of our user study. Section 5 shows the evaluation results. Section 6 discusses the threats to validity of our study. Section 7 presents related work. Finally, we draw the conclusions of our work in Section 8.

2 USAGE SCENARIO

Suppose Alice discovers that, in her code, the use of a cryptographic API (the *Cipher* API) for performing decryption does not adhere to her organization's best practices. She suspects such API misuses are potential security vulnerabilities [32]. Ideally, the usages of Cipher have to correctly handle a range of exceptions [2], encode a decrypted string in UTF-8 [1], and use a strong transformation, e.g. passing "`AES/GCM/NoPadding`" to `Cipher.getInstance` [3]. As the uses of Cipher that do not adhere to best practices [4, 20] are widespread [12], Alice decides to check whether her API usage is consistent with her organization's practices. This requires her to write a rule for the static analyzer in detecting violations. Alice searches for code snippets using Cipher within the hundreds of repositories of her organization. Alice uses *grep*, but searching for "Cipher" results in a large number of matches. The sheer volume of required effort leads Alice to scale up her code search.

Suppose Alice manually writes a rule for matching code. She includes some code lines that she identified, including the function calls, such as `init` and `updateAAD`, that occur in the positive instances. However, Alice finds writing rules to be challenging. This is unsurprising because even expert-written rules are imperfect. In fact, CodeQL's rulesets of cryptographic APIs [5] rules were fixed several times over a period of two years, due to bugs in the rules. Comparing the buggy version of their rules against the newest version of the rule, the fixed rule leads to 217 more warnings on 64 open-source repositories (code snippets that call `Cipher.getInstance`) from the CodeQL databases on github.com [10]. This suggests that the initial version of the rule caught only 80% of the expected API usage locations.

State-of-the-art Active Learning. Alice turns to active learning approaches [17] for identifying code patterns. These approaches would allow Alice to construct a pattern that encodes the correct use of Cipher API, distinguishing correct use from incorrect uses. Using a state-of-the-art active learning approach, ALP [17], Alice provides the positive and negative instances. It infers a pattern (e.g. Figure 3d) that is too general (i.e., it incorrectly matches Figure 3e). Then, it requests for more positive and negative instances.

Because Alice cannot easily find positive and negative instances, she cannot easily refine a pattern and begins to grow frustrated. On inspecting the inferred pattern that now includes `Base64.decode` when constructing a string, Alice thinks that the tool has gone off track as `Base64.decode` just happens to be in the same positive instances, where data are decrypted.

Code line-level feedback. Alice gives SURF a try. SURF organizes a summary of the API usages, displaying and soliciting feedback on code lines from the population instead of considering each instance independently. SURF requires Alice to understand the code lines and to provide direct feedback on individual lines.

Distribution-level analysis. The code lines are organized by their semantic role [46, 48] (e.g., initialization, error handling) in the use of the API, enabling Alice to reason about the code lines simultaneously. By **overlaying** code lines from different instances, Alice can understand common usages comprising dependent code segments, such as interaction with other API methods, at a glance. SURF provides distribution-level statistics regarding each code line's *support* and *information gain*. *Support* measures how frequently a code line appears. Alice determines the most common uses of Cipher at a glance (e.g., the transformation "`RSA/ECB/PKCS1PADDING`" was not frequently used, indicated by the shorter blue bars). *Information gain* measures the improvement from including a code line in the pattern. Given that the current pattern matches 1 positive and 1 negative instance, it has an entropy of 1. By including `new SecretKeySpec(...)`, with a high information gain of 1.0, indicating that it can separate the positive and negative instances, entropy can be decreased to 0. These metrics show Alice the distribution of the code lines among the entire population of programs. A good feature is likely to have both high support and information gain.

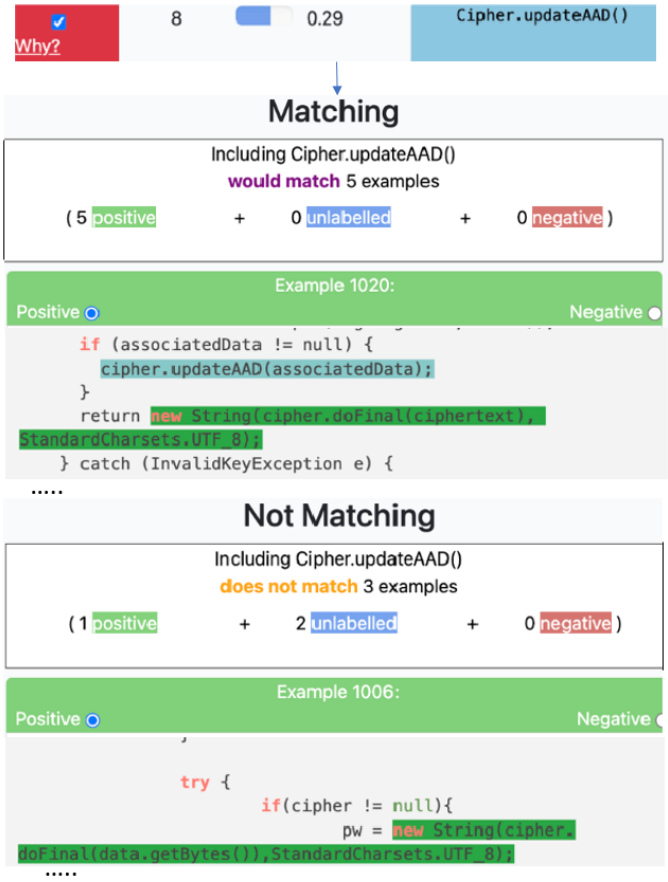

Figure 5: Impact Analysis. Clicking on each feature shows how the feature is distributed among already labelled positive and negative instances, and would match additional instances in the population. Including feature `Cipher.updateAAD` causes over-specialization, and will no longer match a positive instance, i.e., Example 1006.

Impact and What-If analysis. As Alice inspects the code lines with high information gain and support, SURF visualizes their impact on matching the instances (Figure 5). This allows her to identify instances affected by the inclusion of the code line, e.g., instances that do not match `updateAAD`, if it were included. Using the Impact Analysis, Alice discovers that including `updateAAD` would cause a positive instance and two unlabelled instances to not be matched. Alice inspects the instances and believes that they should be matched. Alice suspects that calling `updateAAD` is optional.

Alice clicks on two code lines, `Cipher.updateAAD` and `Cipher.getInstance("AES/GCM/NoPadding")`. SURF immediately provides hints indicating that `updateAAD` is redundant with respect to `Cipher.getInstance("AES/GCM/NoPadding")`. Alice performs a what-if analysis. Alice opens up the What-if view (Figure 6) for a side-by-side comparison of the matching programs if the pattern was updated to include one code line over the other. Contrasting the instances matched by the two possible patterns, Alice realizes that all instances containing the method call `Cipher.updateAAD` also

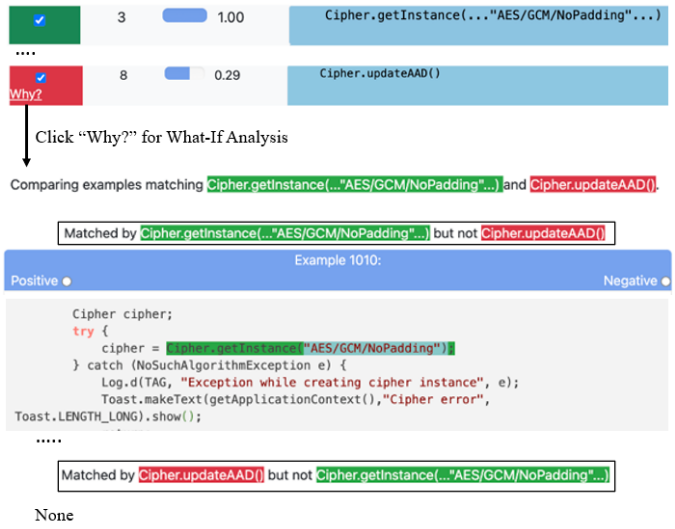

Figure 6: What-If Analysis. Users can contrast the impact of two feature choices $f1$ `updateAAD` and $f2$ `Cipher.getInstance("AES/GCM/NoPadding."` SURF explains no additional match will be found by choosing $f1$ over $f2$, while one additional match can be found with $f2$.

invoke `Cipher.getInstance("AES/GCM/NoPadding")`. Alice realizes that `updateAAD` should only be called if `Cipher.getInstance("AES/GCM/NoPadding")` is used, and is indeed optional.

With this information, Alice settles on `Cipher.getInstance("AES/GCM/NoPadding")` as a suggestion to SURF. Accounting for the feedback on the pattern features, SURF infers a new pattern.

3 SURF

3.1 Inferring a code pattern

SURF aims to infer a code pattern incorporating code line-feedback. SURF represents programs as a graph following prior studies [7, 17], and mines subgraph patterns. SURF mines discriminative subgraphs [37], which can separate positive and negative instances.

3.1.1 Problem formulation. The task of inferring a pattern while considering the user provided code line-level feedback is as follows:

Input: A set of positive instances, $P = \{P_1, P_2, P_3, \dots\}$, and negative instances, $N = \{N_1, N_2, N_3, \dots\}$, identified by the human user, and a set of code lines, C, suggested by the user.

Output: A pattern, which is a set of subgraphs, $S = \{s_1, s_2, s_3, \dots\}$.

At each iteration: SURF requests feedback from the human user, who either labels more instances, i.e., providing more positive and negative instances, or suggest code lines. Ideally, the output pattern should maximize their separation of P and N (further elaborated in Section 3.1.2), and maximize the number of provided code line features that match a vertex from the subgraphs: $|C_{\text{matched}} = \{c \in C : \exists s \in S, c \in V(s)\}|$, where $V(s)$ are the vertices in the subgraph, s.

In the prior study [17], human users only provided positive and negative instances. Thus, there are always zero suggested code

lines, $C = \emptyset$. Using SURF, the size of C increases after each iteration if the user provides code line feedback.

3.1.2 Mining Discriminative Subgraphs. We modify the pattern miner from a previous study [17] to support code line-level feedback. Algorithm 1 shows the algorithm to infer a pattern. First, from a set of positive and negative instances, SURF mines subgraphs that can separate the positive instances from the negative instances (lines 4–6). The CORK [37] criterion, an efficient feature selection mechanism that compares the overlap in features of the positive and negative instances, is applied to discard subgraphs that do not contribute to separating the positive and negative instances.

We reduce the need for a large number of labelled instances and use the code line-level feedback to select subgraphs. Similar to prior work, SURF enumerates subgraphs. In prior work [17], a test of statistical significance was performed to select subgraphs that match more positive instances than negative instances. In SURF, we remove the tests of statistical significance to enable subgraph mining from just a few positive and negative instances. In practice, a human user is unlikely to provide enough instance-level feedback for identifying statistically significant, discriminative subgraphs.

Finally, SURF applies CORK to filter subgraphs that do not separate positive instances from the negative instances(line 11). Filtering is sensitive to the order of subgraphs. Of two subgraphs that separate the same positive and negative instance (say `Base64.code` versus `new String()` in Figure 3), the subgraph first presented to the pattern miner will be selected, and the later subgraph removed (since it no longer contributes to a better positive-negative instance separation). This choice of subgraph depends on how the subgraphs were sorted (lines 8–10). SURF favors the code line-level feedback (line 8) before ordering subgraphs by their discriminativeness (line 9), i.e., if Alice selects only `updateAAD`, subgraphs that contain it precede other subgraphs. The subgraphs are enumerated in decreasing order of the number of matches in the entire population, favoring more general subgraphs over subgraphs specific to a few instances (line 10). For example, `new SecretKeySpec(...)` occurs frequently in the population, and is thus favored over `Log.getStackTraceString`, an uncommon function.

When initially inferring a pattern, we limit the size of the inferred pattern (line 11). This prevents a pattern that overfits the few positive and negative instances.

Algorithm 1 A pattern is mined to separate positives $\mathcal{P}$ from negatives $\mathcal{N}$. Patterns containing the user's suggested code lines are favored.

Require:
- $\mathcal{P} \leftarrow$ positive instances
- $\mathcal{N} \leftarrow$ negative instances
- $\mathcal{A} \leftarrow$ all instances
- $C \leftarrow$ code lines suggested by the user
- $\mathcal{S} \leftarrow$ maximum pattern size to be considered

```
1:  function INFER_PATTERN
2:      D ← {}
3:      for s ∈ enumerateSubgraphs(P , S) do
4:          if match(s, P) > match(s, N) then
5:              D ← D ∪ s
6:          end if
7:      end for
8:      sort(D, compareBy(containsCodeLines(C))
9:          .thenCompareBy(discriminativeness)
10:         .thenCompareBy(matchPopulation(A)))
11:     return filterSubgraphsThatSeparatesPosAndNeg(D, S)
12: end function
```

3.2 Importance metrics

Inspired by active learning techniques [35], we guide users toward *informative* and *representative* code lines.

Support For each code line, SURF counts the support of the pattern if the code line were included, e.g., a code line with a reported support of 10 in SURF means that the code line appears 10 times in the population. Support is computed over the entire population, ignoring their labels. While not all frequent patterns are useful [17, 24], infrequent code lines are not useful.

Information Gain We use *information gain* to measure how well a pattern separates the positive and negative instances after including the code line. Including a code line is analogous to splitting the data at a decision node in a decision tree. The instances matched by the original pattern are partitioned into two sets, one set of instances that match the new pattern, and one set of instances that do not. First, we compute the entropy of the three sets:

- G_P: positive and negative instances matched by the pattern,
- G_m: positive and negative instances matched after the pattern is updated,
- G_e: positive and negative instances excluded after the pattern is updated

Then, for each group G, entropy is computed using the proportion of positive instances (p_+) and negative instances (p_-):

$$\text{Entropy}(G) = -p_+ \log_2(p_+) - p_- \log_2(p_-)$$

The information gain of including a code line is as follows:

$$\text{Entropy}(G_P) - \left(\frac{|G_m|}{|G_P|} \times \text{Entropy}(G_m) + \frac{|G_e|}{|G_P|} \times \text{Entropy}(G_e) \right)$$

If a pattern initially matches five positive and three negative instances, then Entropy(G_P) is 0.95. Say the pattern is modified to include a code line, so it excludes two negative instance, then G_m contains five positive instance and one negative instances, and G_e contains zero positive instances and two negative instance. Entropy(G_m) is 0.65 and Entropy(G_e) is 0. The information gain associated with the code line is 0.95 - (6/8 × 0.65 + 2/8 × 0) = 0.46.

4 EVALUATION DESIGN

We ran a user study with a two-treatment factorial crossover design to evaluate SURF and answer the following research questions:

(1) RQ1. Does SURF improve the participant's ability to comprehend the API usage distribution?
(2) RQ2. How much effort reduction does SURF provide in inferring code patterns?
(3) RQ3. What features in SURF do the participants perceive to be useful?

Table 1: Each task is motivated by a known common weakness enumeration (CWE). Task A is to find code snippets with proper `Cipher` usage and error handling. In CWE-311, NIST determined that Cipher transformations using `"AES/GCM/NoPadding"` can avoid CVE-2016-2183. Task B is to find bugs similar to CWE-330, where the Spring framework had a vulnerability due to incorrect initialization of random values (CVE-2019-3795). For each task, a few positive and negative instances are provided to bootstrap an initial pattern. A user is not shown the target pattern and their task is to refine the initial pattern to fit the given natural language description (Table 2).

Description	CWE	Initial labelled instances	Target pattern
A. `Cipher` usage with a strong transformation and appropriate error-handling	CWE-311 e.g., CVE-2016-2183	4 positive, 4 negative instances that enable inference of `Cipher.getInstance( "AES/GCM/NoPadding")`	`Cipher.getInstance( "AES/GCM/NoPadding")` with `catch(NoSuchPaddingException)`
B. Seeding `SecureRandom` with the current time and using it as a source of randomness for generating keys	CWE-330 e.g., CVE-2019-3795	2 positive, 3 negative instances that enable inference of `SecureRandom.setSeed( System.currentTimeMillis())`	`SecureRandom.setSeed( System.currentTimeMillis())` used with `KeyPairGenerator.generateKeyPair()`

To answer the RQs, we curated two case studies of programs using cryptographic APIs [47]. We analyze cryptographic APIs as many prior studies have demonstrated that fully automatic API usage mining methods do not succeed in inferring the desired patterns [13]. Most usages of the APIs are incorrect [13, 29], limiting the effectiveness of frequency-based pattern mining techniques and thus users must examine and refine the inferred patterns directly to adjust their generality and specificity. Human feedback is, therefore, essential for refining the cryptographic APIs usage patterns.

In the case studies, we pre-defined a target code pattern that distinguishes correct from incorrect usage. When preparing code snippets for a user study, we inlined all relevant fields of classes and static variables into a single method, so that participants could read each instance at the granularity of a single method. We ensured that pattern mining technique used in a prior study [17] could infer the target API usage pattern, if given enough correctly labelled positive and negative instances. For each case study, we also asked code comprehension questions about the global distribution of individual API method invocation. We used counterbalancing to control the order effect. Each participant carried out two different tasks, Task A and Task B, once using SURF and once the baseline.

Baseline. We constructed a baseline tool that mimics an active learning approach for inferring API usage patterns based on instance-level feedback, described in a prior study [17]. This baseline was constructed by downgrading SURF to allow for only labeling positive and negative instances without enabling feature-level feedback, importance metrics, impact analysis, and what-if analysis. As the tool was accessed through a web browser, participants were able to perform text search using Ctrl-F to look for instances containing specific keywords.

After an initial pattern was inferred from positive and negative instances, the tool presented a list of additional matched and unmatched instances to the user. Users could provide additional instance-level feedback by labeling each instance as either positive or negative, or skip the instance if they were unable to decide on a label.

4.1 Participants

As the study has a crossover design, we require fewer subjects as the variability among subjects is controlled for [39]. We recruited 14 participants by reaching out to students in the Computer Science department as well as our contacts working in industry. In total, we recruited 8 Ph.D. students and 6 professional developers. 1 participant had 3 years of programming experience, 5 had 3-8 years of experience, and 8 had more than 8 years of experience. The participants self-reported their familiarity with cryptographic APIs, which is the focus of our case studies, on a 7-point Likert scale. The mean familiarity was 1.5, where 1 is "Most unfamiliar" and 7 is "Most familiar", simulating a scenario where a developer works with an unfamiliar API.

4.2 Study Protocol

We conducted a 1 hour long user study with each participant. The study involved using SURF and the baseline tool described above and two pattern refinement tasks, described in Table 1. The order of the assigned tool (either SURF or the baseline) and the assigned tasks (Task A `Cipher` or Task B `SecureRandom`) were counterbalanced across the participants through random assignment. Each participant was required to complete both case studies, with each case study requiring 12 minutes.

Pre-study survey (2 mins). We asked for the participants' experience and background through a short survey.

Tutorial and warm-up questions (15 mins). We walked the participants through several warm-up questions designed for the participants to discover each user interface feature in SURF. The warm-up questions involved inspecting a small number of usages of the `MessageDigest` API from GitHub.

Each task required 12 minutes to complete (2 minutes for each of the 5 comprehension questions. 2 minutes for providing feedback).

Usage comprehension question. For each task, the participants had to answer five questions related to the API usage and its distribution in the entire population of about 30 instances. We designed the questions with varying difficulties; the simplest questions required a simple lookup of a code line (e.g., what is an example input argument to method `setSeed`?), while the later questions

Table 2: The code comprehension questions are designed to test the participant's ability to understand the global distribution of individual features and the resulting matched and unmatched instances in the rest of population. They test, in order of increasing difficulty, the participants ability to (a) understand a single instance, (b) understand the impact of including a specific line as a feature, (c) understand the distribution of API uses, (d) contrast positive and negative instances, or *matched* and *unmatched* instance, and (e) assess which unlabelled instances could match (or unmatch) by including a specific feature.

Task A (`Cipher`)	Task B (`SecureRandom` and `KeyPairGenerator`)
Questions	
Q1. What is one exception constructed in a catch block (after catching another exception)? (a)	Q1. What is one argument of SecureRandom.setSeed? (a)
Q2. Which positive instances invoke Cipher.init() but not Cipher.updateAAD()? (b)	Q2. Which instances invoke KeyPairGenerator.getInstance() but not KeyPairGenerator.generateKeyPair()? (b)
Q3. How many negative instances constructed a new SecretKeySpec()? (c)	Q3. How many instances construct a new SecureRandom through its constructor, i.e., new SecureRandom(..)? (c)
Q4. What is one class of exception caught by the positive instances but not a negative instance? (d)	Q4. Which instances do not call KeyPairGenerator.getInstance()? (d)
Q5. Which unlabelled instances catch NoSuchPaddingException? (e)	Q5. How many unlabelled instances call KeyPairGenerator.getInstance()? (e)
Pattern Refinement Task	
Detect more instances with exception handling similar to the positive instances but not the negative instances	Detect more instances that generate a key using the same function as the positive instances but not the negative instances.

required contrasting positive and negative instances (e.g., what is one exception caught by the positive instances but not negative instances?), and different code lines (e.g., which instances invoke `getInstance` but not `generateKeyPair`?). These questions, given in Table 2, assess the participants' comprehension of the API usages, and guide them to develop insights about individual code features that they can consider incorporating to refine the current pattern.

Providing feedback for improving the pattern. After answering the questions and building up their understanding of the population of API usages, we instructed the participants to provide feedback. For the baseline tool, the participants provide feedback by labelling instances as either positive or negative. We instructed the participants to skip an instance if they were unsure of its label. For SURF, the participants provide feedback by suggesting code lines. For both tools, we instructed the participants to skip to the last question in the last 2 minutes if they were still answering the usage comprehension questions. This gives each participant at least 2 minutes to provide feedback to the tool for improving the pattern. This task mimics a single iteration of an active learning algorithm requesting feedback. The tasks are shown in Table 2.

Post-study survey (12 mins) At the end of the session, participants answered questions about their experience using each tool, described how they tried to answer the usage comprehension questions and provided feedback on each user interface feature.

Tool setup. For each task, we started with a set of positive and negative instances provided to the participant. For these tasks, we disabled the instance-level feedback on SURF to encourage the participants to provide only code line-level feedback.

Data. For both case studies, we curated about 30 code instances from different repositories on GitHub. We selected about only 30 instances to prevent the human users from getting too overwhelmed from inspecting the instances. The instances were picked to display a range of different uses of the API, including the target pattern.

We make SURF and our data available [6].

Table 3: Answering usage-comprehension questions. SURF **leads to 30% more correct answers, while requiring 22% less time. On average, participants spent about 8 mins using** SURF **as opposed to 10 mins using the baseline.**

	Task A		Task B	
	SURF	**Baseline**	SURF	**Baseline**
Q1	5	6	7	7
Q2	4	5	6	3
Q3	6	2	5	0
Q4	5	2	4	1
Q5	3	2	5	2
# correct answers	3.3	2.4	3.9	2.0
Time taken (mins)	9.6	10.2	8.2	11.5

5 EVALUATION RESULTS

In this section, we report and analyze the results of our user study. We denote each participant as P#.

5.1 Improving usage comprehension

To evaluate user performance using SURF on usage comprehension, we assessed the participants' answers. The detailed results are shown in Table 3. Every participant provided an answer to all five questions in both tasks, regardless of the tool. Participants using SURF provided 1.5 more correct answers (30%). Using a linear mixed-effect model, where the number of correct answers depends on the order, tool, and task. We found that SURF significantly improved over the baseline (p-value < 0.005).

The use of SURF led to greater improvements on Task B, which may be more complex as the pattern includes the use of two APIs (SecureRandom, KeyPairGenerator). On average, participants using SURF on Task B provided 3.8 correct answers while using the

Table 4: 79% of participants using SURF managed to construct the correct pattern, whereas only 43% of those using the baseline achieved the same result. Each participant's pattern was compared against a ground-truth pattern after each study.

	Task A		Task B	
	SURF	**Baseline**	SURF	**Baseline**
# participants with correct patterns	6/7 (86%)	4/7 (57%)	5/7 (71%)	2/7 (29%)

baseline provided just 2 correct answers. Four participants provided correct answers to every question in Task B when using SURF, while no participant got every question correct when using the baseline. Using the baseline, participants would get stuck on a question, giving themselves less time for the subsequent questions.

In the post-study questionnaire, 11 participants (79%) indicated that SURF made understanding the instances easier. To understand the distribution of code lines, P1 indicated that having a tool that "automatically did the comparisons for me was very helpful".

> Overall, SURF helped the participants better understand the API usage distribution. Participants using SURF correctly answered 30% more usage distribution questions.

5.2 Reducing human effort

Participants using SURF completed their tasks in an average of 8 mins and 40 seconds. Using the baseline, participants required 10 mins 48s on average, which is 22% slower than SURF. Every participant was able to utilize the Impact and What-if Analysis in SURF. Overall, SURF helped the participants provide useful feedback for code pattern inference in less time.

We evaluated the patterns inferred. As shown in Table 4, participants were 1.8X more likely to construct the expected pattern when using SURF. Using the baseline, participants only succeeded in producing the expected pattern 43% of the time. Participants using SURF produced the expected pattern 79% of the time.

Using the baseline, we observed that the participants were overwhelmed. They spent more time coming up with ad-hoc crtieria for distinguishing positive instances from negative instances. However, these criteria did not always lead to correct labels.

The post-study questionnaire's responses showed that the participants were more confident in their answers and ability to provide feedback to the tools using SURF. On a 7-point Likert scale shown in Figure 7, participants rated SURF an average of 5.6 of out 7, while rating the baseline an average of 2.9 out of 7. Their feedback suggests that the participants perceived the summary and analysis of SURF to be helpful in finding facts about the API usage distribution.

> Participants using SURF required 22% less time for completing each task, but were 1.8X more likely to construct the expected pattern.

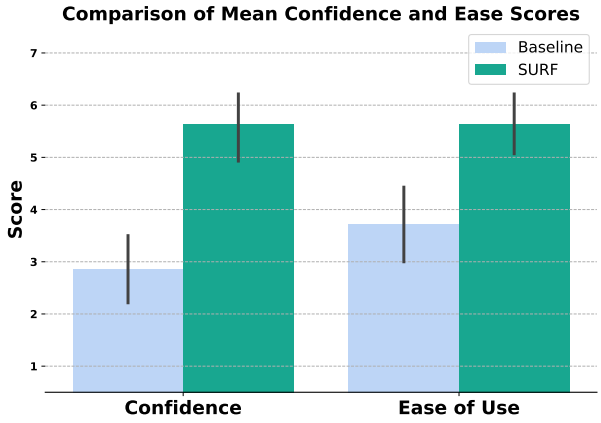

Figure 7: The participants reported a higher confidence score when using SURF (with a median of 5.6) than the baseline (with a median of 2.9). The participants rated SURF (median of 6) to be easier to use than the baseline (median of 4).

5.3 Users' perception of SURF

Our post-study survey solicited the participants' feedback. Participants found SURF to be easier to use (5.6 vs 3.7 on a 7-point Likert scale) compared to the baseline. The distribution of responses is given in Figure 7.

Summary code skeleton. 13 of the 14 participants found the summary code skeleton to be useful, with a median rating of 7 out of 7. P3 mentioned that SURF made it easier to understand the code distribution as it "gives a clear pattern template and functionalities to list concrete instances related to a specific pattern".

Importance Metrics. 13 of the 14 participants found the importance metrics to be useful, with a median rating of 6 out of 7. P6 found them "convenient to determine the line of code to select".

Impact and What-if Analysis. 13 of the 14 participants found the impact and What-if analysis to be useful, with a median rating of 6.5 and 7, respectively. P10 mentioned that he "found myself using the Impact Analysis often to answer the questions". P2 was confident in his answers as SURF "gave me immediate feedback and flagged suggestions " as he used the tool, indicating that dynamically providing feedback was helpful for building confidence.

Limitations and Suggestions. The participants pointed out features that they wished SURF had. Their responses suggested that developers may need support in understanding the code beyond their use of the APIs. P10 wrote that she wanted "tooltips that automatically show the docs for a method when hovering over it.". 5 participants indicated that they wished to have more concrete information, such as their purpose, about the code using the API.

We found that participants cared about double-checking their answers, requiring closer inspection of the instances. P9 wrote "I wished I could see some concrete examples to confirm my understanding of what I was doing". Double-checking their work slowed the participants down. However, if the participants were given more time, they may trust SURF more and double-check their work less. P1 wrote that he "manually confirmed that the patterns were matching (probably a beginner's thing as I build trust in the tool)".

> The participants found SURF easier to use. Participants found every user interface feature useful, with the code skeleton and What-if Analysis perceived as the most useful.

6 DISCUSSION

6.1 Qualitative Analysis

Usage Comprehension. Using the baseline tool, participants found understanding the instances difficult. Participants may get stuck on the earlier usage comprehension questions, lacking enough time for the later questions. On providing feedback to the baseline, P13 wrote "I can't begin to do.". Participants tried to develop a criteria for matching positive and negative instances. P9 wrote that he "tried to remember the patterns which had appeared".

Participants wanted to contrast the positive and negative instances when using the baseline. P10's strategy was to "search for a method name or exception to see if it was used in the positive/negative examples. If it was in all of the positive examples and not in the negative examples, I would use that to check against the unlabelled examples." P5 wrote "I will see the pattern in positive example, and then see what it share with the negative examples". Identifying code lines with high information gain is directly supported by SURF and appears to be a good match for the participants' mental model of how to perform the task.

Text search. Prior work [22] has shown that `grep` returns many irrelevant matches. We observed the same limitation of simple text search. Though participants could locate instances with specific method calls or constants, they were confused by irrelevant results.

Active Learning. Prior studies [17] on active learning assumed that human users can effectively provide instance-level feedback. However, our user study suggests that human feedback would be a bottleneck. Without the interactive support from our tool, the participants in our study were not confident in using the baseline, with a self-reported rating of 2.9 out of 7. The participants developed ad-hoc critiria for classifying positive and negative instances, which led to inaccurate instance-level labels. The participants' poor performance confirms our hypothesis that the challenge lies in soliciting instance-level feedback from users.

Using SURF allowed more participants to successfully guide SURF to infer the expected pattern in 21.9% less time. This suggests that the participants benefited from both providing feedback at the granularity of code lines guided by the distribution of their usages and in the remaining population.

6.2 Threats to Validity

A threat to validity is the lack of familiarity of the study's participants with cryptographic APIs. However, this reflects the reality of developers' understanding of these APIs [29]. The participants' performance may be a lower bound since they may have better results from reduced cognitive overhead analyzing a familiar API.

While our study included only 14 participants, studies designed with a crossover minimize variability, thus requiring fewer participants [39]. As few as 10 participants are often sufficient to gain valuable insights [9]. Our study also showed statistically significant improvements.

7 RELATED WORK

API Learning. Programmers often inspect code examples to understand how to use an API [33, 34]. In particular, they often desire multiple instances of an API use to understand how it is used [15, 33, 44].

Inspired by Examplore [15], SURF overlays code lines from the population of API usages into a code skeleton to simultaneously visualize multiple code instances when soliciting human feedback [11]. Unlike SURF, Examplore only visualizes the API usages, does not infer patterns or offer Impact and What-if Analysis.

API usage patterns. To detect API misuses, researchers have proposed using API specifications, e.g., JavaMOP [16], to detect violations. These specifications produce many false positive [23], showing how human-written rules are error-prone and may benefit from automated techniques.

Dynamic analysis-based API misuse detectors [18, 42] are limited to misuses that throw exceptions. They are less effective for APIs whose failures may not result in exception, e.g., cryptographic APIs.

Many approaches for mining API usage patterns have been proposed [7, 25, 28, 30, 31, 38, 40, 41, 46, 49]. The majority of these approaches do not use any human feedback. Fully automatic approaches may be ineffective on frequently misused APIs [13, 17, 24].

Comparing choices The What-if analysis in SURF supports users in understanding the impact of possible feature choices. SURF provides side-by-side comparison of the instances, with the highlighting of salient differences, to allow users to understand their detailed differences. Similarly, Yan et al. [43] contrast usages of different choices of similar libraries. While their work contrasts code that use different libraries with same usage, SURF contrasts instances with different usages of the same API. The Impact Analyses resembles speculative analysis [8], which anticipates and executes developers' actions in the background to inform them and avoid possible merge conflicts. SURF has a different goal of empowering users to comprehend the matching capability of individual features.

Active Learning using instance-level feedback. Approaches using active learning include ALP [17], ALICE [36], RhoSynth [14] and Arbitrar [24]. Both SURF and ALP shares the same graph representation [7] of programs and mines subgraphs patterns to detect API misuses. ALICE express programs as logic programs. RhoSynth synthesizes code quality rules from user-provided positive and negative instances. Arbitrar detects API misuses by having human users analyze and provide feedback on program traces. These tools do not have interactive intuitive interface to provide fine-grained feedback nor reason about differences among similar API usages.

Manual pattern refinement. CRITICS [45] allows for manual refinement of code patterns, but does not have a pattern inference algorithm. In contrast, users of SURF benefit from both inference and human feedback while considering the guidance provided.

8 CONCLUSION

SURF is an approach for active learning for code pattern inference. Users interactively provide direct code line-feedback to SURF, which guides them through a code skeleton summarizing a population of API usages, importance metrics, impact and what-if analysis. Participants in our user study inspected case studies of cryptographic API usages. Using SURF, participants correctly answered

30% more comprehension questions and 1.8X more likely to construct the expected pattern. Our study has ramifications for active learning; studies should consider the challenge of human users understanding unfamiliar code when providing feedback, and that a different granularity of feedback may reduce human effort.

REFERENCES

[1] 2018. MUBench's misuse in Adempiere related to string encoding. https://github.com/stg-tud/MUBench/blob/4f22263ac364e8861d4e368ab33180607ec4f14e/data/adempiere/misuses/1/misuse.yml#L9.

[2] 2018. MUBench's misuse in IText related to exception handling. https://github.com/stg-tud/MUBench/blob/4f22263ac364e8861d4e368ab33180607ec4f14e/data/itext/misuses/1/misuse.yml#L7.

[3] 2018. MUBench's misuse in YApps related to string encoding. https://github.com/stg-tud/MUBench/blob/4f22263ac364e8861d4e368ab33180607ec4f14e/data/yapps/misuses/1/misuse.yml#L6.

[4] 2018. Rules of using Cipher according to CogniCrypt. https://github.com/CROSSINGTUD/Crypto-API-Rules/blob/master/BouncyCastle-JCA/src/Cipher.crysl.

[5] 2023. CodeQL rule for "Use of a broken or risky cryptographic algorithm". https://codeql.github.com/codeql-query-help/java/java-weak-cryptographic-algorithm/.

[6] 2023. SURF's tool and data. https://github.com/UCLA-SEAL/SURF.

[7] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. 2019. Investigating next steps in static API-misuse detection. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 265–275.

[8] Yuriy Brun, Reid Holmes, Michael D Ernst, and David Notkin. 2011. Proactive detection of collaboration conflicts. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 168–178.

[9] Sarah E Chasins, Elena L Glassman, and Joshua Sunshine. 2021. PL and HCI: Better together. *Commun. ACM* 64, 8 (2021), 98–106.

[10] Github Docs. 2023. Preparing your code for CodeQL analysis (Downloading databases from GitHub.com). https://docs.github.com/en/code-security/codeql-cli/getting-started-with-the-codeql-cli/preparing-your-code-for-codeql-analysis#downloading-databases-from-githubcom.

[11] Ekwa Duala-Ekoko and Martin P Robillard. 2012. Asking and answering questions about unfamiliar APIs: An exploratory study. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 266–276.

[12] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. 2013. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 73–84.

[13] Jun Gao, Pingfan Kong, Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Negative results on mining crypto-api usage rules in android apps. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 388–398.

[14] Pranav Garg and Srinivasan H Sengamedu. 2022. Synthesizing code quality rules from examples. *Proceedings of the ACM on Programming Languages* 6, OOPSLA2 (2022), 1757–1787.

[15] Elena L Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. 2018. Visualizing API usage examples at scale. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. 1–12.

[16] Dongyun Jin, Patrick O'Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient parametric runtime monitoring framework. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 1427–1430.

[17] Hong Jin Kang and David Lo. 2021. Active learning of discriminative subgraph patterns for api misuse detection. *IEEE Transactions on Software Engineering* 48, 8 (2021), 2761–2783.

[18] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. 2019. Effective and efficient API misuse detection via exception propagation and search-based testing. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 192–203.

[19] Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. FaCoY: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*. 946–957.

[20] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. 2017. Cognicrypt: Supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 931–936.

[21] Julia Lawall and Gilles Muller. 2018. Coccinelle: 10 years of automated evolution in the Linux kernel. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 601–614.

[22] Julia Lawall and Gilles Muller. 2022. Automating Program Transformation with Coccinelle. In *NASA Formal Methods Symposium*. Springer, 71–87.

[23] Owolabi Legunsen, Wajih Ul Hassan, Xinyue Xu, Grigore Roşu, and Darko Marinov. 2016. How good are the specs? A study of the bug-finding effectiveness of existing Java API specifications. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 602–613.

[24] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. 2021. Arbitrar: User-guided api misuse detection. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1400–1415.

[25] Zhenmin Li and Yuanyuan Zhou. 2005. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 306–315.

[26] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–28.

[27] Na Meng, Miryung Kim, and Kathryn S McKinley. 2011. Systematic editing: generating program transformations from an example. *ACM SIGPLAN Notices* 46, 6 (2011), 329–342.

[28] Martin Monperrus, Marcel Bruch, and Mira Mezini. 2010. Detecting missing method calls in object-oriented software. In *European Conference on Object-Oriented Programming*. Springer, 2–25.

[29] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. 2016. Jumping through hoops: Why do Java developers struggle with cryptography APIs?. In *Proceedings of the 38th International Conference on Software Engineering*. 935–946.

[30] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. 2009. Graph-based mining of multiple object usage patterns. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering (ESEC/FSE)*. 383–392.

[31] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. 2012. Statically checking API protocol conformance with mined multi-object specifications. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 925–935.

[32] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 2019. Cryptoguard: High precision detection of cryptographic vulnerabilities in massive-sized java projects. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2455–2472.

[33] Martin P Robillard. 2009. What makes APIs hard to learn? Answers from developers. *IEEE software* 26, 6 (2009), 27–34.

[34] Martin P Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16 (2011), 703–732.

[35] Burr Settles. 2009. Active learning literature survey. (2009).

[36] Aishwarya Sivaraman, Tianyi Zhang, Guy Van den Broeck, and Miryung Kim. 2019. Active inductive logic programming for code search. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 292–303.

[37] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S Yu, Xifeng Yan, and Karsten M Borgwardt. 2010. Discriminative frequent subgraph mining with optimality guarantees. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 3, 5 (2010), 302–318.

[38] Suresh Thummalapenta and Tao Xie. 2011. Alattin: mining alternative patterns for defect detection. *Automated Software Engineering* 18, 3-4 (2011), 293–323.

[39] Sira Vegas, Cecilia Apa, and Natalia Juristo. 2015. Crossover designs in software engineering experiments: Benefits and perils. *IEEE Transactions on Software Engineering* 42, 2 (2015), 120–135.

[40] Andrzej Wasylkowski and Andreas Zeller. 2011. Mining temporal specifications from object usage. *Automated Software Engineering* 18, 3-4 (2011), 263–292.

[41] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE*. 35–44.

[42] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exposing library API misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 866–877.

[43] Litao Yan, Miryung Kim, Björn Hartmann, Tianyi Zhang, and Elena L Glassman. 2022. Concept-annotated examples for library comparison. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*. 1–16.

[44] Tianyi Zhang, Björn Hartmann, Miryung Kim, and Elena L Glassman. 2020. Enabling data-driven api design with community usage data: A need-finding study. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[45] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. 2015. Interactive code review for systematic changes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 111–122.

[46] Tianyi Zhang, Ganesha Upadhyaya, Anastasia Reinhardt, Hridesh Rajan, and Miryung Kim. 2018. Are code examples on an online Q&A forum reliable?: a study of API misuse on stack overflow. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 886–896.

[47] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao, and Na Meng. 2022. Automatic Detection of Java Cryptographic API Misuses: Are We There Yet? *IEEE Transactions on Software Engineering* 49, 1 (2022), 288–303.

[48] Hao Zhong and Hong Mei. 2017. An empirical study on API usages. *IEEE Transactions on Software Engineering* 45, 4 (2017), 319–334.

[49] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and recommending API usage patterns. In *European Conference on Object-Oriented Programming*. Springer, 318–343.