Abstract—Software APIs exhibit rich diversity and complexity which not only renders them a common source of programming errors but also hinders program analysis tools for checking them. Such tools either expect a precise API specification, which requires program analysis expertise, or presume that correct API usages follow simple idioms that can be automatically mined from code, which suffers from poor accuracy. We propose a new approach that allows regular programmers to find API misuses. Our approach interacts with the user to classify valid and invalid usages of each target API method. It minimizes user burden by employing an active learning algorithm that ranks API usages by their likelihood of being invalid. We implemented our approach in a tool called Arbitrar for C/C++ programs, and applied it to check the uses of 18 API methods in 21 large real-world programs, including OpenSSL and Linux Kernel. Within just 3 rounds of user interaction on average per API method, Arbitrar found 40 new bugs, with patches accepted for 18 of them. Moreover, Arbitrar finds all known bugs reported by a state-of-the-art tool Apisan in a benchmark suite comprising 92 bugs with a false positive rate of only 51.5% compared to Apisan’s 87.9%.

I. INTRODUCTION

Modern software is composed of APIs. They provide a modular interface encapsulating rich semantic information, rendering them challenging to use in practice. According to a recent study [25], 17% of bugs stem from API misuses. These misuses can have serious security impact [18], [27], [46].

Various program analysis techniques have been proposed to check API misuses. However, API misuse errors still remain widespread [35], [70]. Existing tools for checking API misuses can be broadly classified into two categories. The first category comprises tools that check for violations of given API specifications, such as IMChecker [24], Semmle [19], and Sys [8]. The effectiveness of these tools depends on the quality of the specifications. However, writing precise specifications requires program analysis expertise, making it challenging even for experienced users. Moreover, these specifications must be written in Domain Specific Languages (DSLs) that vary with tools, e.g., Yaml for IMChecker, CodeQL for Semmle, and LLVM IR and Lisp for Sys, which further burden users.

The second category of tools, such as Apisan [69] and JADET [63], presume that correct API usages follow simple idioms which can be automatically mined from code. Specifically, given a large corpus of code using the API, the majority usage pattern is considered as the valid usage, and all deviations from it are regarded as misuses. These tools presume the availability of a large corpus of code using the API and that the majority of its uses are valid. Unfortunately, these two assumptions may not always hold, especially for less commonly used but critical APIs. Furthermore, as shown by recent work [25], these approaches fail to capture common API usage patterns. Additionally, state-of-the-art tools such as Apisan result in many false alarms when there are multiple valid usage patterns of an API.

In general, automatically detecting valid API usage patterns is hard [51], especially without sufficient usage examples. On the other hand, it is unlikely that developers are able and willing to write precise API specifications. However, given a program path representing an API usage, developers can easily determine the validity of the usage. In other words, distinguishing between a real and a false API misuse requires far less effort. Consider the snippet of code in Listing 1 with two calls to the target API png_destroy_write_struct, on Lines 6 and 13. The corresponding program paths (i.e., sequences of instructions or traces), denoted by P

There are four challenges to realize this objective:

1) Efficient trace generation: Given a target API within a codebase, we need a mechanism to efficiently generate all paths or traces to all call-sites of the API. Furthermore, we need a way to reduce each trace, as not all instructions are relevant to an invocation of the API (e.g., those denoted by ellipses at Line 10 in Listing 1).

2) Generic trace representation: Unlike other approaches such as Apisan, which only works for APIs with simple usage patterns [25], we seek a generic method to represent a trace that is capable of handling all classes of APIs with varying complexities and with multiple valid usages.

3) Real-time user interaction: Irrespective of the size and complexity of the traces or the API’s semantics, we need to learn from user feedback quickly and be responsive, i.e., respond to user feedback in a short time span.

4) Accurate alarm identification: We must accurately identify API usages that are most likely to be buggy so that users can confirm an API misuse within a few rounds of interaction. The longer this process draws out, the less likely users are to stay engaged, resulting in a rather low utility of this approach in practice.

In this paper, we present Arbitrar, an interactive tool for finding API misuse bugs based on Maximum Discrepancy
Kernel Density Estimation (MD-KDE), a novel active learning methodology. Unlike existing approaches, ARBITRAR requires neither the API’s specification nor a large code corpus with majority valid uses. Given a target API method to be checked within a codebase, ARBITRAR uses under-constrained symbolic execution [51] to generate inter-procedural program traces for all call-sites to the API method. We minimize traces by backward slicing from the API call-site. These optimized traces are converted into a set of feature vectors. Finally, we use MD-KDE on such vectorized traces to detect those representing the invalid usages of the API. Specifically, in each round of MD-KDE, a trace of the API usage with the highest probability of being invalid is presented to the user.

We evaluate ARBITRAR by applying it to check the uses of 18 target API methods in 21 C/C++ programs, including security-critical codebases like OpenSSL and Linux Kernel. ARBITRAR discovered 40 new bugs, out of which 18 were reported, confirmed, and patched. In addition, we demonstrate that ARBITRAR is highly efficient, taking only 3 rounds of user interaction to discover an API misuse on average. We also conduct a head-to-head comparison between ARBITRAR and API SAN, a state-of-the-art API misuse detector. On a benchmark suite comprising 92 bugs, ARBITRAR finds all known bugs reported by API SAN with a significantly lower false positive rate of 51.5% than API SAN’s 87.9%.

We summarize the contributions of our work:

- We present ARBITRAR, a user-guided API misuse detection tool that is precise (i.e., yields low false positive rates), efficient (i.e., requires few rounds of user interaction), and scalable (i.e., finds bugs in large-scale codebases).
- We propose MD-KDE, a novel active learning algorithm based on kernel density estimation. In particular, MD-KDE picks an unlabeled trace of an API usage that achieves the maximum discrepancy in estimated probability from the correct usage traces of the API.
- We perform an extensive evaluation of ARBITRAR. When applied to check uses of 18 API methods within 21 C/C++ programs, ARBITRAR found 40 new bugs in 3 rounds of user interaction on average per API method.
- The source code of ARBITRAR is made publicly available at https://github.com/petablox/arbitrar

II. MOTIVATION

In this section, we motivate our approach with an erroneous usage of `png_destroy_write_struct` found by ARBITRAR. This API takes two arguments, each as a pointer to a pointer, first to a `png` structure (`png_ptr_ptr`) and second to an `info` structure (`info_ptr_ptr`) as shown below:

```c
void png_destroy_write_struct(
    png_structpp png_ptr_ptr,
    png_infoopp info_ptr_ptr);
```

This function is used to free the memory associated with the `png` structure (`png_ptr_ptr`), which holds information for writing a PNG file, and the associated `info` structure (`info_ptr_ptr`). The `info_ptr_ptr` can be `NULL` in which case only the `png` structure will be freed.

```c
1  png_ptr = png_create_write_struct(...);
2  if (png_ptr == NULL) return(2);
3  info_ptr = png_create_info_struct(png_ptr);
4  if (info_ptr == NULL) {
5    // Valid usage
6    png_destroy_write_struct(png_ptr, NULL);
7    return(2);
8  }
9  // Invalid usage resulting in memory leak.
10 // The second argument should be &info_ptr.
11 #png_destroy_write_struct(png_ptr, (png_infopp)NULL);
```

Listing 1: Example showing valid (✔️) and invalid usages (❌) of `png_destroy_write_struct` as found by ARBITRAR.

Consider the code in Listing 1 that creates `png_ptr` at Line 1 and the associated `info_ptr` at Line 3. If the creation of `info_ptr` fails (i.e., `info_ptr == NULL` evaluates to true at Line 4), the `png_ptr` is freed by calling `png_destroy_write_struct` at Line 6, where `NULL` is passed as the argument for `info_ptr_ptr`.

The Bug: If the creation of both `png_ptr` and `info_ptr` succeeds, i.e., both `png_ptr == NULL` at Line 2 and `info_ptr == NULL` at Line 4 evaluate to false, then it is expected that both pointers will be released in the end. However, at Line 12, the function `png_destroy_write_struct` is incorrectly called with `spng_ptr` and `NULL`, but supposed to be called with `spng_ptr` and `sinfo_ptr`. This causes leakage of the memory allocated to the `info_ptr` and could have security implications such as Denial-of-Service [15].

We examine what it takes for the two state-of-the-art tools — Semmle (adopting a manual approach based on API specifications) and API SAN (adopting an automated approach based on anomaly detection) — to find the bug in Listing 1.

A. Semmle

Semmle finds bugs using manually written semantic patterns. To isolate this bug, we need to write a pattern that at least captures the following:

a) The presence of `png_destroy_write_struct((X, NULL)`, denoted by C1.

b) The existence of a non-NULL `info_ptr`, the second argument of C1, meaning, there exists a call `Y = png_create_info_struct(X)`, denoted by C2, and Y == NULL is false.

c) The existence of a path from C2 to C1, specifically, Y, the return value of C2 is passed as the second argument to C1.

It took a graduate student adept at logic programming and experienced in analyzing security bugs two hours to write a pattern for this task in CodeQL. The query itself consists of 40 lines and involves Semmle library calls to data-flow and control-flow analysis. The majority of time was spent in the edit-run-debug loop where the user was constantly suppressing false positives by adding new predicates. In the end, this checker is able to isolate the bug among 18 usages across 4 projects, although it is worth noting that it still fails to capture the full specification of the API.\(^1\)

\(^1\)The source code of this checker is provided in Appendix 5.
in multiple ways, the task becomes even more challenging, requiring one to either craft a small bug-isolation checker for each individual bug pattern or write a larger and comprehensive specification checker.

This example highlights the difficulty of writing precise API specifications and illustrates why developers loathe writing them. In fact, GitHub awards a bounty for each valid specification of a security vulnerability written in CodeQL [20].

B. API SAN

API SAN adopts an automated approach to finding API misuse bugs based on majority usage patterns. At a technical level, API SAN employs individual checkers, and each checker is responsible for a specific type of usage pattern, such as return value check, argument relation, causally related APIs, and pre- or post-conditions. To find the bug in Listing 1, API SAN must combine the return value with the argument checker to infer the two correct usage patterns of png_destroy_write_struct. Since API SAN does not consider the composition of different types of usage patterns, it will be unable to differentiate our bad use-case from the good one, unless it finds another unrelated signal. In fact, with its condition checker, API SAN flags our good use case as the only alarm of png_destroy_write_struct, while other checkers do not report any.

As we show in Section III-C ARBITRAR finds the bug in Listing 1 within just 4 rounds of user interaction.

III. FRAMEWORK

In this section, we present our ARBITRAR framework, whose overall workflow is depicted in Figure 1. Given a set of C/C++ program(s) and a target API method to check, we first compile the programs to LLVM bitcode, and then generate symbolic traces of all the uses of the API (Section III-A). These symbolic traces are then encoded into feature vectors that capture information relevant to classifying valid and invalid uses of the API (Section III-B). Finally, using our active learning algorithm, we interact with the user by presenting a potentially buggy trace and learning from the feedback (Section III-C) to identify anomalies.

A. Trace Generation

The goal of trace generation is to generate all program paths with calls to an API method in order to precisely capture different usage scenarios of the API method. However, it is infeasible to enumerate all program paths on large real-world software such as the Linux Kernel because of path explosion and engineering limitations. We overcome this problem by performing under-constrained symbolic execution on the contexts around target API calls.

1) Finding Execution Contexts: We use execution context (s) to define the entry point and scope for under-constrained symbolic execution. This is controlled by a parameter called context-depth (d) which represents the maximum distance from a call site k of the target API method. Formally, we define execution context (s) to be a pair s = (g, ϕ), where g is the entry point function and ϕ = {g₁, ..., gₙ} is the scope—a set of functions that are allowed to be explored. Given (i) a call site k of the target API method in a function f; (ii) call graph G of the program; and (iii) context-depth d, we compute the set of execution contexts (S_k) by following the steps in Algorithm 1.

We first perform reverse breadth first search (ReverseBFS) on G from f to find all the functions (F) within distance d (Line 3). Then for each function g in F, we find all the reachable functions (ϕ_g) in G within depth 2 * d using BFS (Line 5). Note that on Line 6, we remove the target API method from each computed scope because we only wish to check how it is used, not how it is implemented. The set of all (g, ϕ_g) pairs is considered as the possible execution contexts for k.

Algorithm 1: Finding execution contexts.

```
Input: CallSite k, CallGraph G, ContextDepth d
1 S_k ← ∅
2 f ← Function(k)
3 F ← ReverseBFS(G, f, d)
4 for g ∈ F do
5   ϕ_g ← BFS(G, g, 2 * d)
6   Remove target of k from ϕ_g
7   S_k ← S_k ∪ { (g, ϕ_g) }
8 return S_k
```

2) Under-Constrained Symbolic Execution: Given an execution context (g, ϕ) for a call site k, we symbolically execute from the entry-point function, g, by using fresh symbols as the arguments. At function calls, we step into the corresponding function h if h ∈ ϕ; otherwise we ignore the call instruction and create a new symbol as the return value of the call.

a) Memory model: We use a simple hash map based memory model where the address (symbolic or concrete) is used as a key into the map. We initialize each byte of memory using a fresh and unconstrained symbolic value. Our memory model follows that in recent works [14], [52] which have shown its effectiveness at finding bugs.

b) Handling loops and recursion: We unroll each loop once and do not step into recursive function calls to avoid path explosion. This is based on the intuition that a single run of the loop body can capture an API’s usage. Nonetheless, we record loop entry (BEGIN_LOOP) and exit (END_LOOP) events as they can be helpful to identify a bug or suppress a false alarm.

Fig. 1: Overview of ARBITRAR.
TABLE I: Interaction Example. We show step-by-step how our active learning model guides the user to find the two invalid usages. In each iteration, the user inspects the trace, and provides binary feedback. The 2D TSNE Plot row shows the distribution of trace encodings projected onto a 2D plane. We represent the positive, negative, unlabeled, and selected datapoints using +, ✗, ×, and ▼ respectively. The Trace Acquisition row provides the intuition for why the trace marked in ▼ is picked.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>1st</th>
<th>2nd</th>
<th>3rd</th>
<th>4th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace</td>
<td>llibpng/png.c:4577</td>
<td>pngtools/pngread.c:43</td>
<td>apngasm/apngasm.c:187</td>
<td>apngasm/apngasm.c:195</td>
</tr>
<tr>
<td>Info</td>
<td>Correct Usage: As the png_ptr and info_ptr are passed in from arguments</td>
<td>Invalid Usage: Double free because of multiple calls to png_destroy_read_struct</td>
<td>Invalid Usage: Double free because of multiple calls to png_destroy_read_struct</td>
<td>Normal trace</td>
</tr>
<tr>
<td>User Feedback</td>
<td>Not a Bug</td>
<td>Bug</td>
<td>Bug</td>
<td>Not a Bug</td>
</tr>
<tr>
<td>2D TSNE Plot</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Trace Acquisition</td>
<td>Pick a trace at random.</td>
<td>Pick a trace that is furthest from the previous negative trace.</td>
<td>Pick a trace that is closest to the known positive trace.</td>
<td>Pick a trace that is furthest from the negative trace and closest to the positive trace.</td>
</tr>
</tbody>
</table>

The result of under-constrained symbolic execution on each execution context is a set of program paths or symbolic traces, i.e., \{ρ₁,...,ρₙ\}. Each symbolic trace (ρ) represents a sequence of program events operating on symbolic or concrete values and has the format as shown in Figure 2. Note that, we ignore infeasible paths (i.e., path constraint is unsatisfiable) and all paths that do not contain our call site k. Every trace contains the event representing target call site k and indicate it by tₖ. Hereafter, we will use k to indicate the index of the program event representing the target API call.

Figure 3 shows an example code snippet and two traces that are generated to capture usages of an API method kzalloc.

**B. Trace Encoding**

The goal of trace encoding is to convert the set of program traces into fixed-dimensional feature vectors that will be used later in our active learning algorithm. Based on an analysis of various API misuse bugs [25], we define a list of features used to encode each trace as shown in Table III. These features are extensive and cover most of the behaviors related to API usage. As such, users are not required to define any features. Nonetheless, ARBITRAR provides a uniform and extensible interface to define features using Datalog rules over relational representations of program traces. Datalog [2], a declarative logic programming language, is popularly used to specify a variety of program analyses (e.g., [55], [64]). Many variants exist in the literature including PQL [43], CodeQL [4], and LogiQL [23].

We illustrate feature definitions in ARBITRAR using the example from Figure 3 which contains an API misuse bug that is exhibited along Trace 1: the return value of the target API kzalloc is dereferenced after it is assumed to be zero. This bug motivates the need for two boolean features: one capturing whether the return value of the target API is assumed to be zero (called ret.assumed_zero), and another capturing whether it is dereferenced (called ret.derefed).
We can define the feature `ret.assumed_zero` using the following three Datalog rules:

\[
\begin{align*}
\text{assumed_zero}(i, e) & \leftarrow \text{ASSUME}(i, e, =, 0, \text{true}). \quad (R_1) \\
\text{assumed_zero}(i, e) & \leftarrow \text{ASSUME}(i, e, \neq, 0, \text{false}). \quad (R_2) \\
\text{ret.assumed_zero} & \leftarrow \text{assumed_zero}(-, e). \quad (R_3)
\end{align*}
\]

A Datalog rule is an “if-then” rule read from right to left. Rules \((R_1)\) and \((R_2)\) compute the binary relation `assumed_zero` as the set of all tuples \((i, e)\) such that the \(i\)-th event in the given trace assumes symbolic expression \(e\) to be zero. Likewise, Rule \((R_3)\) computes the nullary relation \((i.e., \text{a boolean feature})\) which is true if and only if there exists a tuple in relation `assumed_zero` wherein the symbolic expression is the return value of the target API call in the trace, denoted \(\hat{e}_r\).

The feature `ret.derefered` is defined similarly. The complete set of features used in ARBITRAR is presented in Table II. Commonly used relations, such as `assumed_zero` defined by Rules \((R_1)\) and \((R_2)\) above, are presented in Figure 4. The features deal with different aspects of an API: return value, arguments, causality relations, and control flow. We briefly describe each of these sets of features:

1) **Return value.** Return value features are related to how the return value is used. We generate this set of features if the target function has a non-void return type.

2) **Arguments.** Argument features correspond to the symbol type and pre- and post-condition of the arguments. If the target API method has \(m\) arguments, we generate \(m\) sets of argument features, one per argument.

3) **Causality relations.** Causal relations arise when the target API method belongs to a “group” of functions that should be invoked together. Examples include `lock/unlock` and `open/close`. Given a target API method \(g\), we infer such functions as those that are invoked most frequently, across the collected traces. Specifically, we find the top-\(K\) occurring functions before and after the target API call, and construct two causality dictionaries \(D_{\text{in}}(k)\) and \(D_{\text{out}}(k)\). The value of \(K\) is configurable and is set to 5 as default. We then generate one set of causality features for each function \(g\) in \(D_r\) where the scope \(r\) is either \([1, k)\) or \((k, n]\).

4) **Control flow.** This set of features is not specific to any API. We include these loop-related features because they are indicative of the trace structure, and can serve as valuable signals to isolate the bug or suppress false alarms.

**ARBITRAR**’s features are extensible and could conceivably even be automatically generated from examples of valid and invalid API usages, using program synthesis techniques [50, 34]. Datalog, **ARBITRAR**’s language for defining features, is expressive enough and even supports recursion. For instance, the binary relation `belong_to_ptr` is recursively defined (in Figure 4), and enables us to define boolean feature `ret.indirectly_returned` (in Table I) which captures whether the return value of the target API call is returned from the given trace’s execution context indirectly via an arbitrary chain of element pointers.

We encode each symbolic trace into a boolean vector of features. For instance, Figure 2 shows the vectors of return value features for the two traces in the example. Finally, we apply binary encoding to the boolean vectors to produce fixed-dimensional binary vectors.

### C. Active Learning and User Interaction

In this section, we propose a novel active learning solution for the API misuse problem. We first formulate the problem as an interactive anomaly detection problem in machine learning, where the goal is to identify anomalies by interacting with a human expert. Then we present our human-in-the-loop algorithm, MD-KDE, to learn from expert feedback and accurately identify a suspicious data point that will be evaluated by the human expert in each round. Finally, we show that MD-KDE is highly efficient as it can be updated in linear time per round.

#### 1) The Interactive Anomaly Detection (IAD) Problem:

Traditional Anomaly Detection (AD) [29, 10] methods operate in a batch mode, i.e., a machine learning model trained on a dataset with zero or only a few labeled data points is responsible for predicting a set of anomaly candidates. This setting is particularly challenging due to the sparsity of positive signals—confirmed misuses of the target API in our case—and therefore these methods usually rely on additional distributional...
TABLE II: Features Used in ARBITRAR and Their Definitions in the Datalog Language.

<table>
<thead>
<tr>
<th>Feature Feature</th>
<th>Role Role</th>
<th>Return value features</th>
<th>Description Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ret.checked</td>
<td>checked</td>
<td>zero[i], zero[i]</td>
<td>Return value is checked</td>
</tr>
<tr>
<td>ret.assumed_zero</td>
<td>assumed_zero(r, e)</td>
<td>Return value is assumed to be zero</td>
<td></td>
</tr>
<tr>
<td>ret.assumed_not_zero</td>
<td>assumed_not_zero(r, e)</td>
<td>Return value is assumed to be non-zero</td>
<td></td>
</tr>
<tr>
<td>ret.stored</td>
<td>STORE(r, e)</td>
<td>Return value is stored to an existing location</td>
<td></td>
</tr>
<tr>
<td>ret.dereferenced</td>
<td>deref(e)</td>
<td>Return value is dereferenced</td>
<td></td>
</tr>
<tr>
<td>ret.returned</td>
<td>RET(n, e)</td>
<td>Return value is returned to the outer context</td>
<td></td>
</tr>
<tr>
<td>ret.indirectly_returned</td>
<td>STORE(e, e), belongs_to_ptr(e, e), RET(n, e)</td>
<td>Return value is stored into another pointer and returned</td>
<td></td>
</tr>
<tr>
<td>ret.used_in_call</td>
<td>CALL(r, e), e, e</td>
<td>Return value has been used as an argument of a call</td>
<td></td>
</tr>
</tbody>
</table>

Argument features

- arg.is_constant: e[n][i] = c
- arg.is_arg: e[n][i] = a
- arg.is_local: e[n][i] = a
- arg.is_global: e[n][i] = a
- arg.pre.checked: i < k, checked(i, e[n][i])
- arg.pre.assumed_zero: i < k, assumed_zero(i, e[n][i])
- arg.pre.assumed_not_zero: i < k, assumed_not_zero(i, e[n][i])
- arg.pre.used_in_call: i < k, CALL(i, e[n][i])
- arg.post.checked: i > k, checked(i, e[n][i])
- arg.post.dereferenced: i > k, deref(e[n][i])
- arg.post.used_in_call: i > k, CALL(i, e[n][i])

Control flow features

- cf.has_loop: BEG_LOOP(i)
- cf.target_inside_loop: IConfiguration_before_e(k(a), num_end_loop_before_k(b), a - b > 0)

assumptions on anomalies and/or normal data points, such as the “one-class” assumption [53, 57]. However, such an assumption is unrealistic in our case, as it amounts to assuming a single valid usage pattern of every target API. To address this issue, we provide more signals to the machine learning algorithm by allowing it to communicate with a human expert in an iterative fashion. We define the interactive anomaly detection problem below.

**Definition 1 (Interactive Anomaly Detection):** Given a dataset $X = \{x_1, x_2, \cdots, x_n\}$ of $n$ data points, and an expert oracle $f : X \rightarrow \{0, 1\}$ that maps a data point from $X$ to 0 (normal) or 1 (anomaly), the learner can query $f$ using one data point $x^{(i)}$ in each round, for a total number of $T$ rounds. The objective of the Interactive Anomaly Detection (IAD) problem is to maximize the total number of identified anomalies, i.e.,

$$\max \sum_{i=1}^{T} \mathbb{I}[f(x^{(i)}) = 1],$$

where $\mathbb{I}[]$ is the indicator function.

In the setting of API misuse detection, the data points are the trace encodings generated by the method described in Section III-B. The goal is to identify as many anomalous traces as possible by querying a human expert for up to $T$ rounds. In each round, the learner selects one unlabeled data point from the dataset and presents it to the human expert, who provides binary (anomaly or not) feedback to the algorithm. The overall performance is measured by the percentage of true anomalies among the $T$ queries, i.e., the overall precision.

**Remark.** Our problem formulation is different from another existing formulation of AD under the interactive mode [22].
\[ g(x, \mathcal{L} = (\mathcal{P}, \mathcal{N})) = \mathbb{E}_{u \sim \mathcal{D}} [K(u, x)] - \mathbb{E}_{v \sim \mathcal{N}} [K(v, x)], \quad (2) \]

where

\[ K(u, v) = \frac{1}{(h \sqrt{2 \pi})^d} e^{- \frac{|u - v|^2}{2h^2}} \]

is a standard Gaussian kernel with bandwidth \( h \). Here \( d \) is the dimension of the feature space, i.e., \( u, v \in \mathbb{R}^d \). \( h \) is a scalar hyperparameter selected by the leave-one-out cross validation [59]. The expectation in Eq (2) can be estimated using labeled data points in the following form,

\[ g(x, (\mathcal{P}, \mathcal{N})) = \frac{1}{|\mathcal{P}|} \sum_{u \in \mathcal{P}} K(u, x) - \frac{1}{|\mathcal{N}|} \sum_{v \in \mathcal{N}} K(v, x), \quad (3) \]

which is used to derive an efficient update rule in Section III-C3.

We call the active learning algorithm using the above acquisition function as the Maximum Discrepancy Kernel Density Estimation (MD-KDE) algorithm. Intuitively, MD-KDE behaves as follows, in an explore-then-exploit fashion:

1) At the beginning, when there is no positive data point detected, the algorithm simply chooses one unlabeled data point which is the farthest away from existing labeled (negative) data points. This can be viewed as the exploration stage, or model variance minimization [13] in classical active learning. This strategy of diversifying the samples accelerates the process of finding the first anomalous case.

2) When there are both positive and negative data points, the algorithm favors a data point close to existing positive points but far from existing negative points. In the presence of multiple data points that satisfy these criteria, we pick a random point from these data points. This can be viewed as the exploitation stage with a KDE classifier. Since anomalous cases of the same kind tend to be close to each other in the feature space, we find that many similar anomalous cases can be detected using this strategy.

Figure 5 illustrates an iteration in MD-KDE, including the selection of the querying sample, and two possible evaluation outcomes. We show the changes in the scoring landscape (computed by the acquisition function in Eq (2)) after a querying sample is evaluated. If the queried point is positive, it will reinforce the current estimation of the acquisition scores. Otherwise, the landscape will change abruptly to reflect the new evidence.

MD-KDE is a novel algorithm designed for the IAD problem setting where existing active learning-based anomaly detection algorithms [22], [21] fail to perform well. These methods optimize an one-class SVM [53], [57] like objective to classify data points into positive and negative classes. In each round of active learning, they query a data point on the decision boundary to improve the classification model but do not aim for discovering anomalous cases during the interaction with an expert. We highlight another key difference next.

3) Efficient Update for MD-KDE: A major advantage of MD-KDE over existing active learning-based anomaly detection methods is its efficient update between rounds of queries. Instead of training a new machine learning model using the updated labeled dataset per round, which usually takes an hour to several days depending on the size of the dataset and the choice of the model, MD-KDE enjoys exact round-to-round update using linear time proportional to the number of current unlabeled samples. Even in large databases with more than 50,000 traces, the update takes only a few seconds. This allows an expert to stay engaged with our system throughout an interaction session.

Suppose we already computed the acquisition value for an
unlabeled \( x \) at time \( t \) using Eq (\ref{eq:acquisition_function}): \[ g(x, (P_t, N_t)) = \frac{1}{|P_t|} \sum_{u \in P_t} K(u, x) - \frac{1}{|N_t|} \sum_{v \in N_t} K(v, x) \]
\[ = \frac{1}{|P_t|} S_{P_t}(x) - \frac{1}{|N_t|} S_{N_t}(x), \]

where we use short-hand notations \( S_{P_t}(x) \) and \( S_{N_t}(x) \) to denote the sums. Then at the next iteration, we can compute \( g(x, (P_{t+1}, N_{t+1})) \) in \( O(1) \) using,

\[ g(x, (P_{t+1}, N_{t+1})) = \begin{cases} \frac{1}{|P_t|} (S_{P_t}(x) + K(x^{(i)}, x)) - \frac{1}{|N_t|} S_{N_t}(x), & \text{if } x^{(i)} \in P_{t+1} \\ \frac{1}{|N_t|} S_{N_t}(x) - \frac{1}{|N_t|} (S_{N_t}(x) + K(x^{(i)}, x)), & \text{if } x^{(i)} \in N_{t+1} \end{cases} \]

Notice that after each round, a data point is added into either \( P \) or \( N \). For each unlabeled \( x \), the mean of \( K(u, x) \) for \( u \in P \) (or the mean of \( K(v, x) \) for \( v \in N \)) can be updated in \( O(1) \). Therefore the total update time is \( O(|U|) \), where \( U \) is the set of the current unlabeled data points. The update operation could be processed in batch to further optimize the constant factor in the time complexity.

### IV. Implementation

We implemented ARBITRAR in 3.5K and 2K Lines of Code (LoC) of Rust and Python respectively. The trace generation (under-constrained symbolic execution) and trace encoder are implemented using Rust with 3.5K LoC. Our symbolic execution is multi-threaded in order to achieve the performance need for executing on large and complex code bases. Our active learning framework, user interaction system and analysis database system are implemented in Python.

#### A. Analysis setup

In our experience, making a program analysis ready, i.e., fetching the sources, configuring and converting it to LLVM bitcode—which is often assumed to be easy—hinders the usage of many tools by its end-users.

Inspired by the way in which package managers handle open-source packages, we designed our system to be completely automated. As a result, any program available as a Debian package(s) can be checked by just specifying the package name. Given the package name(s) for the program, ARBITRAR automatically fetches and builds the program into LLVM bitcode using Apt-tools [26].

```bash
% fetch and build the debian packages.
$ arbitrar collect --deb libpng --deb pngtools
% ---or---
% fetch all the packages in the provided json file.
$ arbitrar collect packages.json
```

The trace generation and feature encoding for an API method, say `binder_inner_proc_lock`, of the built programs is done by using the following command:

```bash
$ arbitrar analyze -i binder_inner_proc_lock
```

Finally, active learning through user interaction can be commenced by using the following command:

```bash
$ arbitrar learn active binder_inner_proc_lock
```

#### B. Active learning

Our active learning technique interacts with the user by presenting a trace at the source-code level, and the user provides feedback by entering \( Y \) or \( N \), which means whether the presented trace is buggy or not. Figure 6 shows an example of user interaction wherein the trace is presented by highlighting the corresponding source lines. As the trace is path sensitive, it is easy for the user to reason about the sequence of corresponding program states and provide feedback.

### V. Evaluation

In this section, we evaluate ARBITRAR in terms of the following aspects:

- **Effectiveness**: How effective is ARBITRAR compared to APISAN, a state-of-the-art API misuse detection tool?
- **Impact**: Can ARBITRAR find previously unknown API misuse bugs?
- **Scalability**: Does ARBITRAR scale to large real-world codebases and still quickly learn from and respond to user feedback?
- **Extensibility**: How extensible is ARBITRAR to other usage scenarios?

Our experiments are conducted on a 40-core machine with 768 GB RAM. We set the context depth to be zero for all our scenarios?
We excluded bugs related to APIs which have less than five non-buggy traces. Consequently, if the trace containing the bugs is not presented in the first 5 interactions then we consider the bug to be not found by ARBITRAR, and, therefore a false negative. If \( Ar_f = \infty \), then the user can check all traces and there will be no false negatives. Similarly, if \( Ar_f = 0 \), none of the bugs will be found (i.e., false negatives = 100%) as ARBITRAR gets no feedback and may fail to learn. Recall that \( Ar_f \) is not the total number of traces that should be analyzed by the user but rather the number of false positive traces the user is willing to tolerate. Specifically, if the user analyzed \( n \) traces with \( Ar_f = k \), this means there are at least \( n - k \) traces that are true bugs.

Table [I] shows the results of running API\(\text{SAN} \) and ARBITRAR (with \( Ar_f = 5 \)).

b) Detection rate: Overall ARBITRAR’s detection rate (89.1\%) is substantially higher than API\(\text{SAN} \) (76.1\%). Furthermore, the detection rate of ARBITRAR is relatively similar on all benchmarks but API\(\text{SAN} \) has a skewed detection rate and fails to detect misuses of complex APIs. In fact, API\(\text{SAN} \) found zero bugs in APIMU4C’s Curl dataset. Listing 2 shows one such instance.

c) False positives: The false positive rate of API\(\text{SAN} \) (\( fp_{ap} \)) at 87.9\% is significantly higher than that of ARBITRAR (\( fp_{ar} \)) at 51.5\%. For instance, in the case of the OpenSSL benchmark, API\(\text{SAN} \) produced 160 bug reports out of which only 19 were true positives. This is also in line with the observation of the API\(\text{SAN} \) authors (Fig 12 in [69]).

Although, \( fp_{ar} \) at 51.5\% still seems large, in practice this feels negligible because of active learning. Recall that, we consider any non-buggy traces shown to the user as false positives. The false positive rate of 51.5\% means that we showed two traces for every bug and one of them is non-buggy. This means that ARBITRAR was able to learn the API misuses from the feedback on just one trace.

Nonetheless, as indicated by the missed bugs (10.9\%), there are bugs that are not detected by ARBITRAR. As \( Ar_f = 5 \), this means there are 10.9\% of the bugs where the trace containing the bug was not presented in the first 5 interactions. These are the APIs that have complex specifications or multiple disjoint but valid usage patterns. Consider the example in Figure [7] which took 13 interactions to find. Here, whether the return of \texttt{BN_CTX_get} need to be checked depends on previous calls to the function using same \texttt{ctx}.

As mentioned before, false negatives are determined by feedback tolerance (\( Ar_f \)). Figure 8 shows the percentage of experiments, i.e., \( d = 0 \). Symbolic trace generation and feature encoding for a heavily-used API like \texttt{kzalloc} in Linux Kernel takes 35 minutes and generates a database of size 2GB.

### A. Dataset

We use two API misuse bug benchmarks and 21 real-world programs including Linux Kernel as our dataset.

a) Bug-benchmark: Our bug benchmark contains 92 API misuse bugs, including 45 bugs found by API\(\text{SAN} \) in Linux Kernel (denoted \( B_{ap} \)) and 47 bugs in APIMU4C [25], a recently proposed API misuse bug dataset (denoted \( B_{mu} \)). We excluded bugs related to APIs which have less than five occurrences (or call-sites) in the dataset to avoid unnecessarily penalize API\(\text{SAN} \) as it is known to require more API call-sites to be effective [25].

b) Real-world programs: We chose \texttt{libpng} (10 Debian packages), \texttt{libbluetooth} (7 Debian packages), OpenSSL (3 Debian packages) and Linux Kernel 5.7. These programs have APIs with varying complexity and constitute a suitable dataset to evaluate the generality of ARBITRAR.

### B. Effectiveness

A traditional automated bug detection technique is expected to give warnings or bug reports with high precision (i.e., few false positives and false negatives). However, ARBITRAR does not have any prior knowledge about a bug and actively learns from the user feedback. Consequently, there will be non-buggy traces (or false positives) presented to the user so that ARBITRAR can learn API misuses. However, an effective technique will learn quickly from user feedback and should present fewer non-buggy traces.

a) False positives and negatives: Based on this observation, we define false positives of ARBITRAR, denoted by \( fp_{ar} \), as the number of non-buggy traces presented to the user. We also define feedback tolerance (\( Ar_f \)), as the upper bound on the number of non-buggy traces or false positives tolerable for the user. This enables us to measure false negatives (\( fn_{ar} \)). For instance, if \( Ar_f = 2 \), then the user can tolerate only two non-buggy traces. Consequently, if the trace containing the bug is not presented within the first two interactions then we consider the bug to be not found by ARBITRAR, and, therefore a false negative. If \( Ar_f = \infty \), then the user can check all traces and there will be no false negatives. Similarly, if \( Ar_f = 0 \), none of the bugs will be found (i.e., false negatives = 100\%) as ARBITRAR gets no feedback and may fail to learn. Note that \( Ar_f \) is not the total number of traces that should be analyzed by the user but rather the number of false positive traces the user is willing to tolerate. Specifically, if the user analyzed \( n \) traces with \( Ar_f = k \), this means there are at least \( n - k \) traces that are true bugs.

```c
// bug curl3
Missing return value check
if(fp = BIO_new(BIO_s_file()));
// if(fp == NULL) {
    // failIf(data,
    // "BIO_new return NULL, " OSSL_PACKAGE
    // " error %s",
    // openssl_strerror(ERR_get_error()), error_buffer,
    // sizeof(error_buffer)) );
// return 0;
```

Listing 2: Bug found by ARBITRAR and missed by API\(\text{SAN} \) in Curl of \( B_{mu} \) bug-benchmark.
TABLE III: Comparative Evaluation of ARBITRAR and APISAN on Bug-benchmarks.

<table>
<thead>
<tr>
<th>Bug benchmark</th>
<th>Programs</th>
<th># bugs</th>
<th>APISAN</th>
<th>ARBITRAR (Ar_f = 5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Bugs Detection (% of bugs)</td>
<td>False positive rate (fp_ar)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Found</td>
<td>Missed</td>
</tr>
<tr>
<td>APIsan Bugs (Bap)</td>
<td>Linux Kernel</td>
<td>45</td>
<td>45 (100%)</td>
<td>N/A*</td>
</tr>
<tr>
<td>Apimu4c (Bmu)</td>
<td>OpenSSL</td>
<td>32</td>
<td>19 (59.3%)</td>
<td>13 (40.7%)</td>
</tr>
<tr>
<td></td>
<td>Curl</td>
<td>9</td>
<td>0 (0%)</td>
<td>9 (100%)</td>
</tr>
<tr>
<td></td>
<td>Httpd</td>
<td>6</td>
<td>6 (100%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Cumulative</td>
<td></td>
<td>92</td>
<td>70 (76.1%)</td>
<td>22 (23.9%)</td>
</tr>
</tbody>
</table>

Fig. 7: The top listing shows a valid usage of BN_CTX_get: In a sequence of consecutive calls to BN_CTX_get, only the return value of last call needs to be checked. The bottom listing shows a bug as the return value of the function BN_CTX_get is not checked even though there are no previous calls to BN_CTX_get.

Listing 3: Use of uninitialized memory found by ARBITRAR (in 3rd interaction) because of missing memset call that is present in all other call-sites of hci_send_req.

```c
1 // All fields are uninitialized
2 struct hci_rq;
3
4 memset(&rq, 0, sizeof(rq));
5 // Only few fields are initialized
6 rq.rxq = OGF_LINK_CTL;
7 rq.event = EVT_AUTH_COMPLETE;
8 rq.cparam = &cp;
9 rq.mp = &mp;
10 rq.mlen = EVT_AUTH_COMPLETE_SIZE;
11 // uninitialized fields of rq may be
12 // be used by the method
13 if (hci_send_req(dd, &rq, to) < 0)
14 return -1;
```

Fig. 8: The percentage of bugs in bug-benchmarks detected by ARBITRAR with varying feedback tolerance (Ar_f).

any small change in the usage of an API could potentially keep the traces far apart in the feature space and hence takes more feedback to find the invalid usage.

C. Impact

In this section, we evaluate the ability of ARBITRAR to find new bugs in large real world programs. Table IV shows the results of running ARBITRAR and APISAN on chosen programs. In total, we found misuses of 18 APIs which resulted in a total of 40 new bugs. Furthermore, our patches have also been accepted for 18 of these bugs. All of these bugs have a non-ignorable impact on the target program and a few of them have security implications. Most of the bugs required less than 5 interactions with average being ~3. This indicates that ARBITRAR is effective at finding new bugs with minimal (just 3 interactions) user feedback. Consider the bug found in ARBITRAR in libbluetooth as shown in Listing 5. Here, calls to hci_send_req use memset to initialize all the fields of the first argument (rq) to 0 to avoid using uninitialized stack data, i.e., the feature share_arg_with_target is true for the memset call. However, at the location shown in Listing 5 there is no explicit memset, and is flagged by share_arg_with_target being false. Consequently, the trace was presented in just 3rd interaction, and the bug was found.

Unfortunately, irrespective of the huge number of reports (2,084), APISAN did not find any of the above bugs. The main reasons for this are:

1 // OPENSSL: crypto/bn/bn_div.c:190
2 BN_CTX_start(ctx);
3 res = (dv == NULL) ? BN_CTX_get(ctx) : dv;
4 // memset is missing.
5 memset(&rq, 0, sizeof(rq));
6 // Only few fields are initialized
7 rq.rxq = OGF_LINK_CTL;
8 rq.event = EVT_AUTH_COMPLETE;
9 rq.cparam = &cp;
10 rq.mp = &mp;
11 rq.mlen = EVT_AUTH_COMPLETE_SIZE;
12 // uninitialized fields of rq may be
13 // be used by the method
14 if (hci_send_req(dd, &rq, to) < 0)
15 return -1;
TABLE IV: New Bugs Found By ARBITRAR but missed by APISAN. The column First Bug Warning indicate the number of false warnings analyzed before finding the first bug. To avoid bloating the bug numbers, code smell issues and non-triggerable bugs (highlighted rows) are counted as just one irrespective of the number of occurrences.

<table>
<thead>
<tr>
<th>Program</th>
<th>API Method</th>
<th>Bug Description</th>
<th>Impact</th>
<th>ARBITRAR</th>
<th>APISAN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Total Bugs</td>
<td>First Bug Warning</td>
</tr>
<tr>
<td>Linux</td>
<td>cdev_add</td>
<td>cdev_init not called</td>
<td>Uninitialized read</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>i2c_new_client_device</td>
<td>Return value not stored</td>
<td>Memory Leak</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>i2c_add_adapter</td>
<td>Return value not checked</td>
<td>Code Smell</td>
<td>1 (12*)</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>v4l2_ctl_handler_setup</td>
<td>Return value not checked</td>
<td>Code Smell</td>
<td>1 (58*)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>v4l2_m2m_get_vq</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>1 (41*)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>vb2_plane_vaddr</td>
<td>Return value not checked</td>
<td>Code Smell</td>
<td>1 (118*)</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>SSL</td>
<td>CRYPTO_zalloc</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>143</td>
<td></td>
</tr>
<tr>
<td>libpng</td>
<td>png_destroy_read_struct</td>
<td>Double free</td>
<td>Memory corruption</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>png_destroy_write_struct</td>
<td>Memory leak</td>
<td>Memory corruption</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>png_create_info_struct</td>
<td>Argument not checked</td>
<td>Memory corruption</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>png_crc_read</td>
<td>Argument not checked</td>
<td>Memory corruption</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>png_malloc</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>png_callloc</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>libbluetooth</td>
<td>adp_get_proto_port</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hci_get_route</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adp_data_get</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hci_devba</td>
<td>Return value not checked</td>
<td>System Crash</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>hci_send_req</td>
<td>Argument not memset-ted</td>
<td>Uninitialized read</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1,022</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2,084</td>
<td></td>
</tr>
</tbody>
</table>

Total

40 Avg: 2.6

Listing 4: Missing initialization call bug found by ARBITRAR (in 3rd interaction) in Linux Kernel: Here, there is a missing call to cdev_init before cdev_add.

```
1  cdev = cdev_alloc();
2  if (!cdev) {
3    pr_err("Could not allocate cdev for minor %d, %s\n", minor, name);
4    ret = -ENOMEM;
5  }
6  goto done;
7 }
8 cdev->owner = THIS_MODULE;
9 cdev->ops = fops;
10 kobject_set_name(&cdev->kobj, name);
11 missing call to cdev_init
12 ret = cdev_add(cdev, dev, 1);
```

- **Small number of call-sites of the API method**: APISAN requires a relatively large number of API call-sites to learn semantic beliefs and consequently find misuses. But the above APIs have a relatively small number (10-100) of occurrences. However, using active learning allows ARBITRAR to overcome this limitation.

- **Complex API Semantics**: The semantics of certain APIs functions is complex and cannot be handled by the different checkers of APISAN. However, ARBITRAR can capture these by using a large set of semantic features. Listing 4 shows a missing initialization bug found by ARBITRAR. This bug requires understanding the causal relationship between cdev_add and cdev_init, which APISAN failed to infer.

### D. Scalability

As mentioned in Section IV, execution of ARBITRAR occurs in two independent steps, i.e., Trace generation (or Analysis setup) and Active learning.

1) **Trace generation**: We use a multi-threaded implementation for our trace generation and feature extraction process to achieve scalability. It takes ~66 milliseconds to generate a trace and for a fairly used API in Linux Kernel there are usually 500-800 traces. Hence for an API, the trace generation and feature extraction usually finish within a couple of minutes. Even for a heavily used API such as kzalloc with 1,937 occurrences and 31,704 traces, the generation and feature extraction take only ~35 minutes.

All the generated traces are stored in a database to be inspected later using our active learning technique. Note that, for a given API method and program, trace generation is a one-time task.

2) **Active learning**: Irrespective of the number of traces, our active learning technique is responsive with a response time of milliseconds, i.e., after the user provides the feedback it takes only milliseconds to learn and present the next most probable bug trace This is because of our fast update mechanism using MD-KDE. The traditional update mechanisms such as KDE and KDE with caching have very high response time and are not scalable to be used in a user-driven active learning setting especially when we have a large amount of traces. Figure 9 shows the response time of our MD-KDE compared to the existing techniques. As shown, the response time of the naïve-ly using the KDE model increases drastically with the number of traces. However, our update mechanism, i.e., MD-KDE is only experiencing linear growth. In reality, it stays in a relatively
constant runtime, enabling the user to interact with the system in real-time.

Separating our trace generation and active learning enables ARBITRAR to be run in a two-phase mode. The trace generation (automated but time consuming task) can be run overnight with all the traces stored in a database. The user can analyze these traces later using our fast active-learning technique. This two-phase approach is indeed what developers expect to see from a program analysis tool [12].

E. Extensibility

To demonstrate the extensibility and flexibility of our MD-KDE learning framework, we conduct an experiment to answer the following question: With one known bug, are we able to utilize ARBITRAR to find the remaining bugs faster?

To handle this situation, we extend our tool to allow the user to provide the labeled known bug at the beginning. The other parts of our system remain the same, and the user will interact with our system as usual. To perform the experiment, we pick a few APIs from our previous experiments that satisfy the following criteria: (a) there are multiple bugs, and (b) there is room for improvement. With each API, we randomly select one bug to be the known labeled bug.

We want to measure the improvements in terms of (1). the number of iterations needed before we hit the first true bug \( (x_f) \), and (2). the number of iterations needed to cover all the remaining bugs \( (x_i) \). Note that the selected sample bug will not be counted anymore, therefore the number of bugs \( y' \) in the with-sample-bug setting will be off-by-1 from the original number of bugs \( y \).

Table \[\text{V}\] shows 4 APIs we choose, the number of bugs \( y \), and our measurement \( x_i \) and \( x_f \) in both the original setup and the with-sample-bug setup. We can see a clear improvement when one bug label is provided. Therefore, when a user already possesses one bug pattern, they can leverage this and use ARBITRAR to find bugs even faster and with higher precision.

F. Discussion on false positives

We want to emphasize that our notion of false positive rate is fundamentally different from traditional bug finding tools. First, ARBITRAR does not assume any pattern about API usage, and has to learn from the user feedback from the ground up. The false positives (or non-buggy traces) before the first bug serve as feedback for ARBITRAR to learn a model for the API misuse detection. Second, traditional bug finding tools cannot learn from user feedback. As such, the false positive rate is fixed. However, with ARBITRAR, as shown in Section \[\text{V-E}\] each false positive marked by the user contributes to improving the accuracy of the model and consequently decreases the false-positive rate.

VI. LIMITATIONS

Although ARBITRAR is an effective API misuse detection tool, it has the following limitations:

- **Requirement of API method:** Unlike APISAN, ARBITRAR is API method-specific and requires the method name to be provided. We provide a way \[\text{[28]}\] to enumerate the number of call sites of an API method, which could be used to select API methods with large occurrences. However, to be more precise, a simple statistical analysis \[\text{[22]}\] can be made on the codebase to determine the methods of interest, which can be then provided to ARBITRAR.

- **Incomplete trace generation:** Similar to APISAN, we only consider direct call sites to the API method. Consequently, our trace generation technique may miss traces of the API method invoked through function pointers. However, we can use a points-to analysis \[\text{[56]}\] to resolve function pointer targets and consider them during trace generation.

- **Discrete user feedback:** Our active learning algorithm allows only one bit of user feedback, i.e., Yes or No. However, in practice, the user may want to provide other kinds of feedback, e.g., Yes with 80% confidence. Our current design does not allow for such feedback. However, techniques like ALPF \[\text{[40]}\] could be employed to handle such feedback.

- **Sensitivity to user feedback:** Our active learning algorithm trusts all user feedback. However, a user might make a mistake. In our experience, when given wrong feedback (i.e., marked a correct usage to be incorrect) for \text{malloc}, our model was able to correct itself after a few correct answers. However, this may not generalize to all APIs as the model depends on the complexity of the API. An extensive study is required to precisely assess the sensitivity of our model to user feedback.

VII. FUTURE WORK

In our future work, we plan to extend ARBITRAR in the following directions:

- **Model interpretation and transfer learning:** We believe that models learned by our algorithm are interpretable and transferable, i.e., a model built for \text{malloc} could work for other allocator functions such as \text{realloc} and \text{calloc}. We plan to explore this by testing a pre-trained model of an
API method on other related API methods. We also plan to generate a description for each trace explaining which features contributed to selecting the trace.

- **Prioritizing high severity bugs:** It is important to prioritize high severity API misuse bugs. For instance, API misuse bugs that cause memory leaks and code smells are arguably less important than those that cause memory corruption. It is important to prioritize these high severity bugs. To handle this, we plan to allow the user to provide type feedback as well (e.g., memory leak, memory corruption, etc.). We can then use this feedback to train a bug type detection model that can be used to prioritize traces indicating critical bugs.

### VIII. Related Work

The goal of bug-finding techniques is to detect violations of a certain set of rules, e.g., memory corruption, information flow from an untrusted source to a sensitive sink, integer overflow, etc. Static tools for finding such violations are based on model checking [5], data-flow analysis [62], [41], type inference [54], [9], or symbolic execution [8]. The rules that can be checked by a particular tool can be either fixed [41], configured via annotations in the code [58], [62], or specified by the user through a Domain-Specific Language (DSL) [17], [8].

API misuse bugs are violations of rules imposed by the specification of API functions. IMChecker [24] uses a YAML [67] based DSL for specifying the behavior of API functions and checks for its violations in lightweight static traces computed over the program. Semmle [19] can find API misuses by semantic patterns of correct or erroneous behavior specified using CodeQL [4], a declarative logic programming language based on Datalog, over a relational representation of programs. Similarly, MOPS [11] uses a finite automata based specification whose violations are checked using a model checker.

However, crafting a formal API specification is hard, and it is unreasonable to expect developers to write precise specifications (Section II). Consequently, many API specification inference techniques have been proposed. These techniques compute lightweight program traces and use static features such as control-dependencies [45] or various mining techniques such as FRECPO [47], [3], factor graphs [39], [33], semantics-constrained mining [6], and frequent item sets [37] to infer API specifications. These techniques usually require a large corpus of programs covering all possible usages of the API. As we show in Section [V] however, this assumption does not always hold in practice.

Another class of techniques is based on the intuition that bugs are anomalous or deviant behavior [16]. JUXTA [44] identifies common patterns in file system implementations and detects semantic bugs in file systems as violations of these patterns using lightweight symbolic reasoning. Similarly, JIGSAW [61] targets resource access vulnerabilities. APISAN [69] is a generic approach to find API misuse bugs by encoding common patterns as semantic beliefs. However, APISAN fails to capture certain common usage patterns [25]. Furthermore, as we also show in Section [V] APISAN has a high false positive rate for complex APIs. Machine learning techniques such as Apriori algorithm [60], [40], clustering [64], [65], and kNN [4] are also used to find bugs as anomalous patterns. Similar to specification mining techniques, the success of these bug finding techniques depends on the availability of a large corpus of programs. Other bug pattern learning techniques such as Vecfinder [48] and Vuldeepecker [36] also require a large bug corpus and hence are infeasible for API misuse detection. Unlike existing approaches, ARBITRAR requires neither the specification nor a large code corpus with majority valid uses. Furthermore, our technique is general and can handle APIs of diverse use cases and complexity. We avoid the need for a large code corpus by actively learning from user feedback. Unlike other active learning techniques [68], ARBITRAR learns quickly and provides a responsive interface, even though we interact with the user at a much finer level, i.e., program traces.

Employing user feedback for bug detection has been explored in Eugene [42] and Ursa [71], but these systems need a client analysis to be specified in Datalog. Similarly, other user-guided techniques based on Bayesian inference [49], [28] also require a client analysis. In contrast, ARBITRAR does not require any analysis specification.

### IX. Conclusion

We presented ARBITRAR, a user-guided approach for finding API misuses. ARBITRAR interacts with the programmer to classify valid and invalid uses of a target API. It employs a novel active learning algorithm to minimize user burden. We demonstrated the effectiveness of ARBITRAR by using it to find new bugs in a rich set of target APIs in large real-world C/C++ programs within a few rounds of user interaction.

### ACKNOWLEDGMENTS

We want to thank our shepherd Nick Nikiforakis and anonymous reviewers for their valuable comments and inputs to improve our paper. We also thank Anthony Canino and Kihong Heo for contributing to the initial implementation of ARBITRAR’s symbolic execution engine. This research was supported by grants from AFRL (#FA8750-20-2-0501), ONR (#N00014-18-1-2021), and NSF (#1836822).


import cpp
import semmle.code.cpp.dataflow.DataFlow

class DestroyWriteCall extends FunctionCall {
    DestroyWriteCall() {
        this.getTarget().getName() = "png_destroy_write_struct"
    }
}

class CreateInfoCall extends FunctionCall {
    CreateInfoCall() {
        this.getTarget().getName() = "png_create_info_struct"
    }
}

predicate zeroComparison(EqualityOperation e, Variable v) {
    exists (Expr zero |
        zero.getValue() = "0" |
        zero = e.getLeftOperand() and
        v = e.getRightOperand().(VariableAccess).getTarget() |
        or
        zero = e.getRightOperand() and
        v = e.getLeftOperand().(VariableAccess).getTarget())
}

from
    EqualityOperation e,
    Variable info_ptr,
    IfStmt control,
    DestroyWriteCall destroy_write_call
where
    zeroComparison(e, info_ptr) and
    exists (AssignExpr assign |
        assign.getEnclosingFunction() =
        control.getEnclosingFunction() and
        assign.getValue().(VariableAccess).getTarget() =
        info_ptr and
        exists (CreateInfoCall info_call |
            assign.getRValue() = info_call) and
        control.getControllingExpr() = e and
        destroy_write_call.getArgument(1).getValue() = "0" and
        destroy_write_call.getEnclosingFunction() =
        control.getEnclosingFunction() and
        not control.getThen().getBasicBlock().
        contains(destroy_write_call.getBasicBlock())
select e, control, destroy_write_call

Listing 5: Semmle checker for png_destroy_write_struct

Memory Leak Bug

APPENDIX

A. Semmle Checker

Listing 5 shows the Semmle Checker to find the occurrences of bug similar to the bug in the Listing 1.