AFRL-RI-RS-TR-2021-188

DRAKE: SIGNATURE-GUIDED DETECTION OF BUGS AND VULNERABILITIES

UNIVERSITY OF PENNSYLVANIA

NOVEMBER 2021

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nations. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2021-188 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE CHIEF ENGINEER:

/ S /
WILLIAM E. MCKEEVER
Work Unit Manager

/ S /
GREGORY J. HADYNSKI
Assistant Technical Advisor
Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
NOVEMBER 2021	FINAL TECHNICAL REPORT	JAN 2020	MAY 2021

4. TITLE AND SUBTITLE

DRAKE: SIGNATURE-GUIDED DETECTION OF BUGS AND VULNERABILITIES

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
FA8750-20-2-0501	N/A	62788F

5d. PROJECT NUMBER	5e. TASK NUMBER	5f. WORK UNIT NUMBER
NOVA	PE	NN

6. AUTHOR(S)

Mayur Naik

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8.PERFORMING ORGANIZATIONREPORT NUMBER
University of Pennsylvania 3330 Walnut Street Philadelphia PA 19104	

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10.SPONSOR/MONITOR'S ACRONYM(S)	11.SPONSOR/MONITOR'S REPORT NUMBER(S)
Air Force Research Laboratory/RITA 525 Brooks Road Rome NY 13441-4505		AFRL-RI-RS-TR-2021-188

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. **SUPPLEMENTARY NOTES**

14. ABSTRACT

Existing static analysis approaches and tools have limited accuracy in finding bugs and vulnerabilities in large, complex programs. The DRAKE project develops a machine learning based methodology to help developers interactively craft custom static checkers. The user is not expected to know the internals of program analysis; at the same time DRAKE is extensible by program analysis experts, to target new classes of bugs and vulnerabilities. We demonstrate the effectiveness of DRAKE at finding hundreds of new Application Programming Interface (API) misuse bugs in widely used C++ programs such as the Linux kernel and the OpenSSL cryptographic library, with few rounds of user interaction and high accuracy.

15. SUBJECT TERMS

Static analysis, security vulnerabilities, machine learning, anomaly detection, API misuses

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
U	U	U	SAR	52

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
WILLIAM E. MCKEEVER	**N/A**

PREVIOUS EDITION IS OBSOLETE.

STANDARD FORM 298 (REV. 5/2020)

Prescribed by ANSI Std. Z39.18

Contents

LIST OF FIGURES

LIST OF TABLES

1 Summary

Software Application Programing Interfaces (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 Drake 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, Drake found 40 new bugs, with patches accepted for 18 of them. Moreover, Drake 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%.

2 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 [31], 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@6 and P@12, consist of Line 1, 2, 3, 4, 6; and 1, 2, 3, 4, 12 respectively. When presented with the two traces, developers

with knowledge of `png_destroy_write_struct` can recognize that P@6 is valid and P@12 is invalid. Can we use this feedback to create effective API misuse checkers?

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 report, we present Drake, 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, Drake 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, Drake 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 Drake 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. DRAKE discovered 40 new bugs, out of which 18 were reported, confirmed, and patched. In addition, we demonstrate that DRAKE 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 DRAKE and APISAN, a state-of-the-art API misuse detector. On a benchmark suite comprising 92 bugs, DRAKE finds all known bugs reported by APISAN with a significantly lower false positive rate of 51.5% than APISAN's 87.9%.

We summarize the main contributions:

- We present DRAKE, 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 DRAKE. When applied to check uses of 18 API methods within 21 C/C++ programs, DRAKE found 40 new bugs in 3 rounds of user interaction on average per API method.

3 Methods, Assumptions, and Procedures

3.1 Motivation

We motivate our approach with an erroneous usage of `png_destroy_write_struct` found by DRAKE. 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:

```
void png_destroy_write_struct(
  png_structpp png_ptr_ptr, png_infopp 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.

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 `&png_ptr` and `NULL` but *supposed* to be called with `&png_ptr` and `&info_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 APISAN (adopting an automated approach based on anomaly detection) — to find the bug in Listing 1.

3.1.1 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 $C1$.

```
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  ...
10 // Invalid usage resulting in memory leak.
11 // The second argument should be &info_ptr.
12 🐞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 DRAKE.

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]. When misuses of an API manifest 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].

[1]The source code of this checker is provided in Appendix.

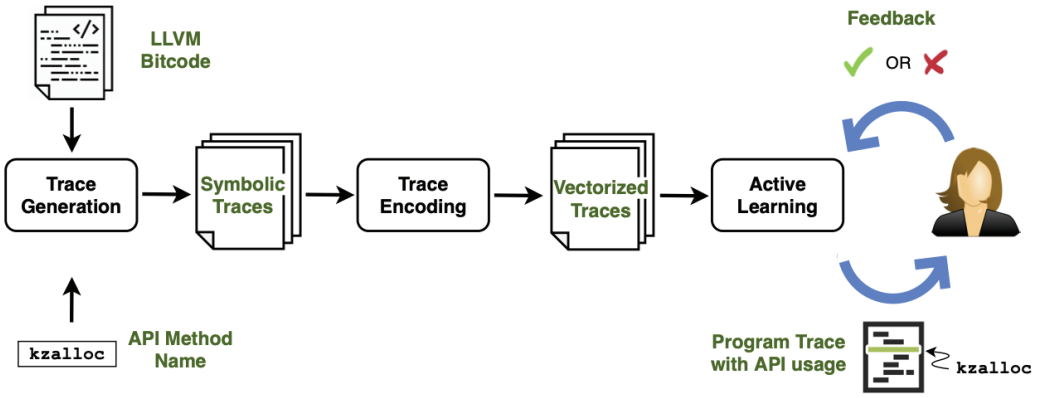


Figure 1: Overview of DRAKE.

3.1.2 APISAN

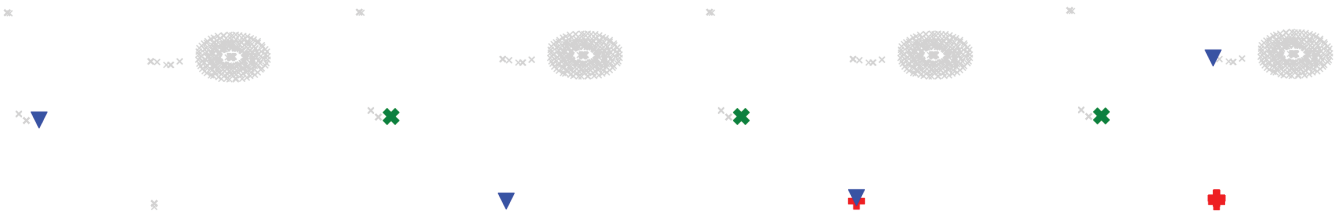
APISAN adopts an automated approach to finding API misuse bugs based on majority usage patterns. At a technical level, APISAN employs individual checkers, and each checker is responsible for a specific type of usage pattern, such as *return value check*, *argument relation*, *causally related APIs*, and *pre- or post-conditions*. To find the bug in Listing 1, APISAN must combine the *return value* with the *argument* checker to infer the two correct usage patterns of `png_destroy_write_struct`. Since APISAN 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, APISAN 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 3.2.3, DRAKE finds the bug in Listing 1 within just 4 rounds of user interaction.

3.2 The DRAKE Framework

In this section, we present our DRAKE 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 3.2.1). These symbolic traces are then encoded into feature vectors that capture information relevant to classifying valid and invalid uses of the API (Section 3.2.2). Finally,

Table 1: 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.

Iteration	1^{st}	2^{nd}	3^{rd}	4^{th}
Trace	libpng/png.c:4577 →png_destroy_read_struct(&c.png_ptr, &c.info_ptr, NULL); fclose(fp); return sng_error;	pngtools/pngread.c:43 →png_destroy_read_struct(&png_ptr, &info_ptr, ... png_destroy_read_struct(&png_ptr, &info_ptr,	apngasm/apngasm.c:187 →png_destroy_read_struct(&png_ptr, &info_ptr, ... png_destroy_read_struct(&png_ptr, &info_ptr,	apngasm/apngasm.c:195 →png_destroy_read_struct(&png_ptr, &info_ptr,
Info	Correct Usage: As the png_ptr and info_ptr are passed in from arguments	Invalid Usage : Double free because of multiple calls to png_destroy_read_struct	Invalid Usage : Double free because of multiple calls to png_destroy_read_struct	Normal trace
User Feedback	Not a Bug	Bug	Bug	Not a Bug
2D TSNE Plot				
Trace Acquisition	Pick a trace at random.	Pick a trace that is furthest from the previous negative trace.	Pick a trace that is closest to the known positive trace.	Pick a trace that is furthest from the negative trace and closest to the positive trace.

using our active learning algorithm, we interact with the user by presenting a potentially buggy trace and learning from the feedback (Section 3.2.3) to identify anomalies.

3.2.1 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* [51] on the contexts around target API calls.

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 $s = (g, \varphi)$, where g is the entry point function and $\varphi = \{g_1, ..., g_n\}$ 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.

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 $h \in \varphi$; otherwise we ignore the call instruction and create a new symbol as the return value of the call.

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

Algorithm 1: Finding execution contexts.

Input: `CallSite` k, `CallGraph` G, `ContextDepth` d

$S_k \leftarrow \emptyset$
$f \leftarrow \texttt{Function}(k)$
$F \leftarrow \texttt{ReverseBFS}(G, f, d)$
for $g \in F$ **do**
    $\varphi_g \leftarrow \texttt{BFS}(G, g, 2 * d)$
    Remove target of k from φ_g
    $S_k \leftarrow S_k \cup \{ (g, \varphi_g) \}$
return S_k

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.

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 (`BEG_LOOP`) and exit (`END_LOOP`) events as they can be helpful to identify a bug or suppress a false alarm.

The result of under-constrained symbolic execution on each execution context is a set of program paths or symbolic traces, i.e., $\{\rho_1, ..., \rho_n\}$. 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 t_k. 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`.

3.2.2 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

(symbolic variable)	α		
(function name)	f		
(integer)	c		
(boolean)	b	::=	`true` | `false`
(arithmetic operation)	$\oplus$	::=	$+ \mid - \mid \times \mid \div \mid \%$
(symbol type)	τ	::=	`arg` | `local` | `global` | `symbol`
(symbolic expression)	e	::=	$c \mid \alpha_\tau$
(relational operation)	p	::=	$= \mid \neq \mid \geq \mid > \mid \leq \mid <$
(i-th program event)	t_i	::=	`CALL` $(i, e_r, f, \bar{e}_a)$
		|	`ASSUME` (i, e_1, p, e_2, b)
		|	`STORE` (i, e_l, e_r)
		|	`LOAD` (i, e_r, e_l)
		|	`GEP` (i, e_r, e_l)
		|	`BINARY` $(i, e_r, \oplus, e_1, e_2)$
		|	`RET` (i, e)
		|	`BEG_LOOP`(i)
		|	`END_LOOP`(i)
(target event)	t_k	$\equiv$	`CALL`$(k, \hat{e}_r, \hat{f}, \bar{\hat{e}}_a)$
(symbolic trace)	ρ	::=	$[t_1, \ldots, t_k, \ldots, t_n]$

Figure 2: Symbolic trace format.

list of features used to encode each trace as shown in Table 2. 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, DRAKE 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 DRAKE 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

Source		Trace 1 (ρ_1)	Trace 2 (ρ_2)
	`void *make_copy(void *b, size_t s){`	$\texttt{CALL}(1, \alpha_{\texttt{symbol}}, \texttt{kzalloc}, [\gamma_{\texttt{arg}}, \texttt{GFP\_KERNEL}])$.	$\texttt{CALL}(1, \alpha_{\texttt{symbol}}, \texttt{kzalloc}, [\gamma_{\texttt{arg}}, \texttt{GFP\_KERNEL}])$.
(ρ_1, ρ_2)	`  void *p;`	$\texttt{ASSUME}(2, \alpha_{\texttt{symbol}}, =, 0, \texttt{true})$.	$\texttt{ASSUME}(2, \alpha_{\texttt{symbol}}, =, 0, \texttt{false})$.
(ρ_1, ρ_2)	`  p = kzalloc(s, GFP_KERNEL);`	$\texttt{STORE}(3, \alpha_{\texttt{symbol}}, 0)$.	$\texttt{CALL}(3, \_, \texttt{memset}, [\alpha_{\texttt{symbol}}, \beta_{\texttt{arg}}, \gamma_{\texttt{arg}}])$.
(ρ_1, ρ_2)	`  if(!p){`	$\texttt{RET}(4, \alpha_{\texttt{symbol}})$.	$\texttt{RET}(4, \alpha_{\texttt{symbol}})$.
(ρ_1)	`  *p = 0;`	**Feature vector:**	**Feature vector:**
(ρ_1)	`  goto err;`	`ret.checked = true`	`ret.checked = true`
	`  }`	`ret.assumed_zero = true`	`ret.assumed_is_zero = false`
(ρ_2)	`  memcpy(p, b, s);`	`ret.assumed_not_zero = false`	`ret.assumed_not_zero = true`
(ρ_1, ρ_2)	`  err: return p;`	`ret.used_in_call = false`	`ret.used_in_call = true`
	`}`	`ret.derefed = true`	`ret.derefed = false`
		`ret.returned = true`	`ret.returned = true`

Figure 3: Example traces and features generated to check usages of `kzalloc`. In both traces, `p` is checked and returned: Trace 1 assumes `p` to be zero whereas Trace 2 assumes `p` to be non-zero. Furthermore, `p` is dereferenced in Trace 1, whereas it is used as an argument to `memset` in Trace 2. This information is captured in the return value features of their respective feature vectors. Note that the two arguments `void *b` and `size_t s` are assigned symbolic variables $\beta_{\texttt{arg}}$ and $\gamma_{\texttt{arg}}$ in our symbolic traces.

Datalog rules:

$$\begin{array}{lll} \texttt{assumed\_zero}(i,e) & :- & \texttt{ASSUME}(i,e,=,0,\texttt{true}\ ). \quad (R_1) \\ \texttt{assumed\_zero}(i,e) & :- & \texttt{ASSUME}(i,e,\neq,0,\texttt{false}). \quad (R_2) \\ \texttt{ret.assumed\_zero} & :- & \texttt{assumed\_zero}(\_,\hat{e}_r). \quad (R_3) \end{array}$$

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., 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.derefed` is defined similarly. The complete set of features used in DRAKE is presented in Table 2. 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 `fopen`/`fclose`. 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 $D_{[1,k)}$ and $D_{(k,n]}$. 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.

$$
\begin{aligned}
\texttt{checked}(i, e) &:\!- \texttt{ASSUME}(i, e_1, \_, e_2, \_), e_1 = e \texttt{ or } e_2 = e.\\
\texttt{assumed\_zero}(i, e) &:\!- \texttt{ASSUME}(i, e, =, 0, \texttt{true}) \texttt{ or}\\
&\quad\ \texttt{ASSUME}(i, e, \neq, 0, \texttt{false}).\\
\texttt{assumed\_not\_zero}(i, e) &:\!- \texttt{ASSUME}(i, e, =, 0, \texttt{false}) \texttt{ or}\\
&\quad\ \texttt{ASSUME}(i, e, \neq, 0, \texttt{true}) \texttt{ or}\\
&\quad\ \texttt{ASSUME}(i, e, >, 0, \texttt{true}) \texttt{ or}\\
&\quad\ \texttt{ASSUME}(i, e, <, 0, \texttt{true}).\\
\texttt{derefed}(i, e) &:\!- \texttt{STORE}(i, e, \_) \texttt{ or } \texttt{LOAD}(i, \_, e) \texttt{ or } \texttt{GEP}(i, \_, e).\\
\texttt{used\_in\_binary}(i, e) &:\!- \texttt{BINARY}(i, \_, \_, e_1, e_2), e_1 = e \texttt{ or } e_2 = e.\\
\texttt{belongs\_to\_ptr}(e_1, e_2) &:\!- e_1 = e_2.\\
\texttt{belongs\_to\_ptr}(e_1, e_2) &:\!- \texttt{GEP}(i, e_1, e), \texttt{belongs\_to\_ptr}(e, e_2).\\
\texttt{num\_beg\_loop\_in\_range}(i, j, 0) &:\!- i > j.\\
\texttt{num\_beg\_loop\_in\_range}(i, j, c) &:\!- i \leq j, \texttt{ENTER\_LOOP}(i), c' = c + 1,\\
&\quad\ \texttt{num\_beg\_loop\_in\_range}(i + 1, j, c').\\
\texttt{num\_beg\_loop\_in\_range}(i, j, c) &:\!- i \leq j,\\
&\quad\ \texttt{num\_beg\_loop\_in\_range}(i + 1, j).\\
\texttt{num\_end\_loop\_in\_range}(i, j, 0) &:\!- i > j.\\
\texttt{num\_end\_loop\_in\_range}(i, j, c) &:\!- i \leq j, \texttt{EXIT\_LOOP}(i), c' = c + 1,\\
&\quad\ \texttt{num\_end\_loop\_in\_range}(i + 1, j, c').\\
\texttt{num\_end\_loop\_in\_range}(i, j, c) &:\!- i \leq j,\\
&\quad\ \texttt{num\_end\_loop\_in\_range}(i + 1, j).\\
\texttt{num\_beg\_loop\_before\_k}(c) &:\!- \texttt{num\_beg\_loop\_in\_range}(0, k - 1, c).\\
\texttt{num\_end\_loop\_before\_k}(c) &:\!- \texttt{num\_end\_loop\_in\_range}(0, k - 1, c).
\end{aligned}
$$

Figure 4: Datalog rules defining relations commonly used in defining features (Table 2). The ':—' operator is implication (from right to left), '_' is projection, and comma is conjunction.

DRAKE'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, DRAKE's language for defining features, is expressive enough and even supports recursion. For instance, the binary relation `belongs_to_ptr` is recursively defined (in Figure 4), and enables us to define boolean feature `ret.indirectly_returned` (in Table 2) 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 3 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.

3.2.3 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

Table 2: Features Used in DRAKE and Their Definitions in Datalog.

Feature	Rule	Description
	Return value features	
ret.checked	$\texttt{checked}(\_, \hat{e}_r)$	Return value is checked
ret.assumed_zero	$\texttt{assumed\_zero}(\_, \hat{e}_r)$	Return value is assumed to 0
ret.assumed_not_zero	$\texttt{assumed\_not\_zero}(\_, \hat{e}_r)$	Return value is assumed to be non-0
ret.stored	$\texttt{STORE}(\_, \_, \hat{e}_r)$	Return value is stored to an existing location
ret.derefed	$\texttt{derefed}(\_, \hat{e}_r)$	Return value is dereferenced
ret.returned	$\texttt{RET}(n, \hat{e}_r)$	Return value is returned to the outer context
ret.indirectly_returned	$\texttt{STORE}(\_, \hat{e}_r, e_1), \texttt{belongs\_to\_ptr}(e_1, e_2), \texttt{RET}(n, e_2)$	Return value is stored into another pointer and returned
ret.used_in_binary	$\texttt{used\_in\_binary}(\_, \hat{e}_r)$	Return value has been used in a binary operation
ret.used_in_call	$\texttt{CALL}(\_, \_, \_, \bar{e}_a), \hat{e}_r \in \bar{e}_a$	Return value has been used as an argument of a call
	Argument features	
$\texttt{arg.is\_constant}_x$	$\bar{\hat{e}}_a[x] = c$	x-th argument is a constant
$\texttt{arg.is\_arg}_x$	$\bar{\hat{e}}_a[x] = \alpha_{\texttt{arg}}$	x-th argument is a trace argument
$\texttt{arg.is\_local}_x$	$\bar{\hat{e}}_a[x] = \alpha_{\texttt{local}}$	x-th argument is a local variable
$\texttt{arg.is\_global}_x$	$\bar{\hat{e}}_a[x] = \alpha_{\texttt{global}}$	x-th argument is a global variable
$\texttt{arg.pre.checked}_x$	$i < k, \texttt{checked}(i, \bar{\hat{e}}_a[x])$	x-th argument is checked before target call
$\texttt{arg.pre.assumed\_zero}_x$	$i < k, \texttt{assumed\_zero}(i, \bar{\hat{e}}_a[x])$	x-th argument is assumed to be zero before target call
$\texttt{arg.pre.assumed\_not\_zero}_x$	$i < k, \texttt{assumed\_not\_zero}(i, \bar{\hat{e}}_a[x])$	x-th argument is assumed to be non-zero before target call
$\texttt{arg.pre.used\_in\_call}_x$	$i < k, \texttt{CALL}(i, \_, \_, \bar{e}_a), \bar{\hat{e}}_a[x] \in \bar{e}_a$	x-th argument is used in a call before target call
$\texttt{arg.post.checked}_x$	$i > k, \texttt{checked}(i, \bar{\hat{e}}_a[x])$	x-th argument is checked after target call
$\texttt{arg.post.derefed}_x$	$i > k, \texttt{derefed}(i, \bar{\hat{e}}_a[x])$	x-th argument is dereferenced after target call
$\texttt{arg.post.returned}_x$	$i > k, \texttt{RET}(n, \bar{\hat{e}}_a[x])$	x-th argument is returned to outer context
$\texttt{arg.post.used\_in\_call}_x$	$i > k, \texttt{CALL}(i, \_, \_, \bar{e}_a), \bar{\hat{e}}_a[x] \in \bar{e}_a$	x-th argument is used in a call after target call
	Causality relation features	
$\texttt{invoked}_{(g,r)}$	$i \in r, \texttt{CALL}(i, \_, g, \_)$	Function g is called during scope r
$\texttt{invoked\_multi}_{(g,r)}$	$i \in r, j \in r, i \neq j, \texttt{CALL}(i, \_, g, \_), \texttt{CALL}(j, \_, g, \_)$	Function g is invoked multiple times during scope r
$\texttt{share\_arg\_with\_target}_{(g,r)}$	$i \in r, \texttt{CALL}(i, \_, g, \bar{e}_a), \bar{e}_a \cap \bar{\hat{e}}_a \neq \emptyset$	Function g is called with at least one common argument as target call
$\texttt{target\_uses\_g\_ret}_{(g,r)}$	$i \in r, \texttt{CALL}(i, e_r, g, \_), e_r \in \bar{\hat{e}}_a$	Result of call to g is used as an argument in the target call
$\texttt{g\_uses\_target\_ret}_{(g,r)}$	$i \in r, \texttt{CALL}(i, \_, g, \bar{e}_a), \hat{e}_r \in \bar{e}_a$	Result of target call is used as an argument in a call to g
$\texttt{g\_ret\_checked}_{(g,r)}$	$i \in r, \texttt{CALL}(i, e_r, g, \_), \texttt{checked}(i, e_r)$	Result of call to g is checked
$\texttt{g\_ret\_assumed\_zero}_{(g,r)}$	$i \in r, \texttt{CALL}(i, e_r, g, \_), \texttt{assumed\_zero}(i, e_r)$	Result of call to g is assumed to be 0
$\texttt{g\_ret\_assumed\_not\_zero}_{(g,r)}$	$i \in r, \texttt{CALL}(i, e_r, g, \_), \texttt{assumed\_not\_zero}(i, e_r)$	Result of call to g is assumed to be non-0
	Control flow features	
cf.has_loop	$\texttt{BEG\_LOOP}(\_)$	The trace contains a loop
cf.target_inside_loop	$\texttt{num\_beg\_loop\_before\_k}(a)$, $\texttt{num\_end\_loop\_before\_k}(b), a - b > 0$	The target call is inside a loop

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.

Interactive Anomaly Detection (IAD) 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 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* $\mathcal{X} = \{x_1, x_2, \cdots, x_n\}$ *of* n *data points, and an expert oracle* $f : \mathcal{X} \mapsto \{0, 1\}$ *that maps a data point from* $\mathcal{X}$ *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.,*

$$\max_{\{x^{(i)}\}_{i=1}^{T},\ s.t.\ x^{(i)} \neq x^{(j)}\ \forall i \neq j} \sum_{i=1}^{T} \mathbb{I}[f(x^{(i)}) = 1], \tag{1}$$

where $\mathbb{I}[\cdot]$ *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 3.2.2. 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, 1, 21]. Instead of optimizing the precision within the T queries, their setting focuses on obtaining a better classifier, which is measured by the final prediction AUC scores on the entire dataset. This objective is a reasonable choice for some AD applications where false positives are tolerable. However in the setting of API misuse detection, false positives are not tolerable and every anomalous case needs to be confirmed by a developer. Therefore these extra queries should be counted as part of the querying budget in our IAD formulation.

A Kernel Density Estimation based Active Learning Algorithm We present a novel kernel density estimation (KDE) based active learning algorithm for the IAD problem. The proposed algorithm follows the active learning framework in Algorithm 2. In each round, given the current labeled dataset, the learner first computes the ranking score for each unlabeled data point using the acquisition function, and then asks the expert to evaluate the unlabeled data point with the maximum score. Intuitively, such a data point corresponds to the trace that the algorithm believes is the most likely to be a misuse of the target API, based on the labels provided by the developer thus far.

Algorithm 2: The active learning algorithm framework.

Input: Data points $\mathcal{X} = \{x_1, x_2, \cdots, x_n\}$, expert $f : \mathcal{X} \mapsto \mathcal{Y}$, acquisition function g.

Initialize unlabeled data points $\mathcal{U} := \mathcal{X}$, labeled data points $\mathcal{L} := \emptyset$
for $t \leftarrow 1$ **to** T **do**
  Select $x^{(i)} \leftarrow \text{argmax}_{x \in \mathcal{U}}\, g(x, \mathcal{L})$
  Evaluate $y^{(i)} \leftarrow f(x^{(i)})$
  Update $\mathcal{U} \leftarrow \mathcal{U} \setminus x^{(i)}$
  Update $\mathcal{L} \leftarrow \mathcal{L} \cup \{(x^{(i)}, y^{(i)})\}$
return $\mathcal{L}$

The acquisition function g is a scoring function for every unlabeled data point in $\mathcal{U}$ using the labeled dataset $\mathcal{L}$. Our acquisition function is designed

to choose the unlabeled data point which achieves the maximum discrepancy between the kernel density estimator of the positive (anomalous) and negative (normal) data. More formally, with the labeled dataset $\mathcal{L}$ represented using the positive and negative data points $\mathcal{P}$ and $\mathcal{N}$, the acquisition function is defined as follows,

$$g(x, \mathcal{L} = (\mathcal{P}, \mathcal{N})) = \mathbb{E}_{u \sim \mathcal{P}}[K(u, x)] - \mathbb{E}_{v \sim \mathcal{N}}[K(v, x)], \tag{2}$$

where

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

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,

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

which is used to derive an efficient update rule in Section 7.

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.

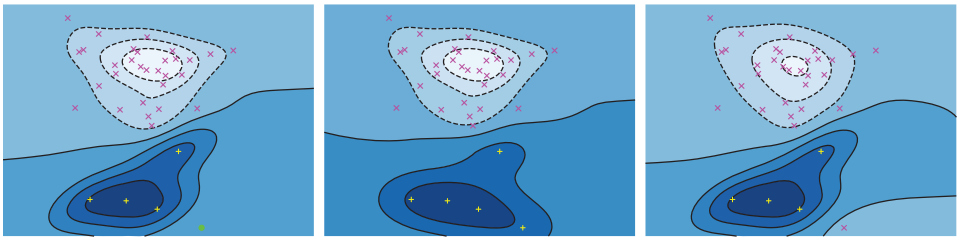


Figure 5: The change of the (acquisition function) scoring landscape in one round of MD-KDE. We represent the positive, negative, and the selected unlabeled data point(s) using yellow + signs, magenta × signs, and a green dot, respectively. Left: an unlabeled data point is selected; Middle: the updated landscape when the point is evaluated to be positive; Right: the updated landscape when the point is evaluated to be negative.

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.

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 (3):

$$\begin{aligned} g(x, (\mathcal{P}_t, \mathcal{N}_t)) &= \frac{1}{|\mathcal{P}_t|} \sum_{u \in \mathcal{P}_t} K(u, x) - \frac{1}{|\mathcal{N}_t|} \sum_{v \in \mathcal{N}_t} K(v, x) \\ &= \frac{1}{|\mathcal{P}_t|} S_{\mathcal{P}_t}(x) - \frac{1}{|\mathcal{N}_t|} S_{\mathcal{N}_t}(x), \end{aligned}$$

where we use short-hand notations $S_{\mathcal{P}_t}(x)$ and $S_{\mathcal{N}_t}(x)$ to denote the sums. Then at the next iteration, we can compute $g(x, (\mathcal{P}_{t+1}, \mathcal{N}_{t+1}))$ in $\mathcal{O}(1)$ using,

$$g(x, (\mathcal{P}_{t+1}, \mathcal{N}_{t+1})) =$$
$$\begin{cases} \frac{1}{|\mathcal{P}_t|+1}(S_{\mathcal{P}_t}(x) + K(x^{(i)}, x)) - \frac{1}{|\mathcal{N}_t|} S_{\mathcal{N}_t}(x), & \text{if } x^{(i)} \in \mathcal{P}_{t+1} \\ \frac{1}{|\mathcal{P}_t|} S_{\mathcal{P}_t}(x) - \frac{1}{|\mathcal{N}_t|+1}(S_{\mathcal{N}_t}(x) + K(x^{(i)}, x)), & \text{if } x^{(i)} \in \mathcal{N}_{t+1} \end{cases}$$

Notice that after each round, a data point is added into either $\mathcal{P}$ or $\mathcal{N}$. For each unlabeled x, the mean of $K(u, x)$ for $u \in \mathcal{P}$ (or the mean of $K(v, x)$ for $v \in \mathcal{N}$) can be updated in $\mathcal{O}(1)$. Therefore the total update time is $\mathcal{O}(|\mathcal{U}|)$, where $\mathcal{U}$ 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.

3.3 Implementation

We implemented DRAKE 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.

3.3.1 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, DRAKE *automatically fetches and builds the program into LLVM bitcode* using Apt-tools [26].

```
% 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:

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

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

```
$ arbitrar learn active binder_inner_proc_lock
```

3.3.2 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.

4 Results and Discussion

4.1 Evaluation

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

- **Effectiveness:** How effective is DRAKE compared to APISAN, a state-of-the-art API misuse detection tool?
- **Impact:** Can DRAKE find previously unknown API misuse bugs?

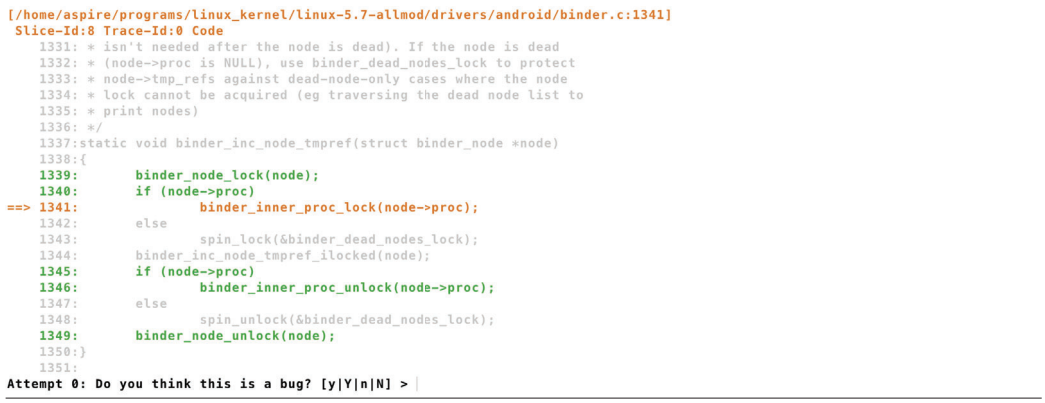


Figure 6: The user interface of DRAKE showing a path sensitive trace and requesting for user feedback.

- **Scalability:** Does DRAKE scale to large real-world codebases and still quickly learn from and respond to user feedback?
- **Extensibility:** How extensible is DRAKE 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 experiments, i.e., $d = 0$. Symbolic trace generation and feature encoding for a heavily-used API like `kzalloc` in Linux Kernel takes 35 minutes and generates a database of size 2GB.

4.1.1 Dataset

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

Bug-benchmark Our bug benchmark contains 92 API misuse bugs, including 45 bugs found by APISAN 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 APISAN as it is known to require more API call-sites to be effective [25].

Real-world programs We chose `libpng` (10 Debian packages), `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 DRAKE.

4.1.2 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, DRAKE 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 DRAKE can learn API misuses. However, an effective technique will learn quickly from user feedback and should present *fewer* non-buggy traces.

False positives and negatives Based on this observation, we define false positives of DRAKE, 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 DRAKE, 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 DRAKE 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.

Table 3 shows the results of running APISAN and DRAKE (with $Ar_f = 5$).

Detection rate Overall DRAKE's detection rate (89.1%) is substantially higher than APISAN (76.1%). Furthermore, the detection rate of DRAKE is relatively similar on all benchmarks but APISAN has a skewed detection rate and fails to detect misuses of complex APIs. In fact, APISAN found zero bugs in APIMU4C's Curl dataset. Listing 2 shows one such instance.

Table 3: Comparative Evaluation of DRAKE and APISAN on Bug-benchmarks.

Bug benchmark	**Programs**	**# bugs**	APISAN			DRAKE ($Ar_f = 5$)		
			Bugs Detection (% of bugs)		**False positive rate (fp_{ap})**	**Bugs Detection (% of bugs)**		**False positive rate (fp_{ar})**
			Found	**Missed**		**Found**	**Missed**	
APISan Bugs (B_{ap})	Linux Kernel	45	45 (100%)	N/A*	71.9%	40 (88.9%)	5 (11.1%)	50.6%
APIMU4C (B_{mu})	OpenSSL	32	19 (59.3%)	13 (40.7%)	88.1%	28 (87.5%)	4 (12.5%)	50.9%
	Curl	9	**0**	9 (100%)	100%	8 (88.9%)	1 (11.1%)	57.9%
	Httpd	6	6 (100%)	0	97.6%	6 (100%)	0	50%
Cumulative		92	**70 (76.1%)**	**22 (23.9%)**	**87.9%**	**82 (89.1%)** ↑	**10 (10.9%)** ↓	**51.5%** ↓

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

Listing 2: Bug found by DRAKE and missed by APISAN in Curl of B_{mu} bug-benchmark.

False positives The false positive rate of APISAN (fp_{ap}) at 87.9% is significantly higher than that of DRAKE (fp_{ar}) at 51.5%. For instance, in the case of the OpenSSL benchmark, APISAN *produced 160 bug reports out of which only 19 were true positives*. This is also in line with the observation of the APISAN 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 DRAKE *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 DRAKE. 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 `BN_CTX_get` need to be checked depends on previous calls to the function using same `ctx`.

As mentioned before, false negatives are determined by feedback tolerance (Ar_f). Figure 8 shows the percentage of bugs detected (Y-axis) as we increase the feedback tolerance (X-axis). The line ($Ar_f = 5$) shows the result used in Table 3. There are several interesting observations: First, most (80%) of the bugs can be found within the first two interactions ($Ar_f = 2$). Second, almost all (99%) of the bugs c an be f ound within the first twelve i nteractions

```
1  // OPENSSL:crypto/bn/bn_div.c:190
2  BN_CTX_start(ctx);
3  res = (dv == NULL) ? BN_CTX_get(ctx) : dv;
4  ✔tmp = BN_CTX_get(ctx);
5  ✔snum = BN_CTX_get(ctx);
6  ✔sdiv = BN_CTX_get(ctx);
7  if (sdiv == NULL)
8      goto err;
```

```
1  ctx = BN_CTX_new();
2  if (ctx == NULL)
3      goto err;
4  BN_CTX_start(ctx);
5  Missing null check.
6  tmp = BN_CTX_get(ctx);
7  // if (tmp == NULL)
8  //     goto err; // bug 1ff7425d61
```

Figure 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`.

($Ar_f = 12$). Finally, there were only two bugs that require more than 12 interactions. These bugs reveal a *drawback of our extensive feature vector*: 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.

4.1.3 Impact

In this section, we evaluate the ability of DRAKE to find new bugs in large real world programs. Table 4 shows the results of running DRAKE 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 DRAKE is effective at finding new bugs with minimal (just 3 interactions) user feedback. Consider the bug found in DRAKE in `libbluetooth` as shown in Listing 3. Here, calls to `hci_send_req` use `memset` to initialize all the fields of the first argument (`rq`) to 0 to avoid using uninitialized s tack data, i .e.,

Table 4: New Bugs Found By DRAKE 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.

Program	**API Method**	**Bug Description**	**Impact**	DRAKE		APISAN
				Total Bugs	**First Bug Warning**	**Warnings**
Linux	`cdev_add`	`cdev_init` not called	Uninitialized read	1	3	445
	`i2c_new_client_device`	Return value not stored	Memory Leak	3	5	
	`i2c_add_adapter`	Return value not checked	Code Smell	1 (12*)	10	
	`v4l2_ctrl_handler_setup`	Return value not checked	Code Smell	1 (58*)	1	
	`v4l2_m2m_get_vq`	Return value not checked	System Crash	1 (41*)	2	
	`vb2_plane_vaddr`	Return value not checked	Code Smell	1 (118*)	2	
SSL	`CRYPTO_zalloc`	Return value not checked	System Crash	5	3	143
libpng	`png_destroy_read_struct`	Double free	Memory corruption	2	2	474
	`png_destroy_write_struct`	Memory leak	Memory corruption	1	4	
	`png_create_info_struct`	Argument not checked	Memory corruption	3	2	
	`png_crc_read`	Argument not checked	Memory corruption	1	1	
	`png_malloc`	Return value not checked	System Crash	7	1	
	`png_calloc`	Return value not checked	System Crash	1	1	
libbluetooth	`sdp_get_proto_port`	Return value not checked	System Crash	1	1	1,022
	`hci_get_route`	Return value not checked	System Crash	2	3	
	`sdp_data_get`	Return value not checked	System Crash	4	1	
	`hci_devba`	Return value not checked	System Crash	4	2	
	`hci_send_req`	Argument not memset-ted	Uninitialized read	1	3	
Total				40	Avg: 2.6	2,084

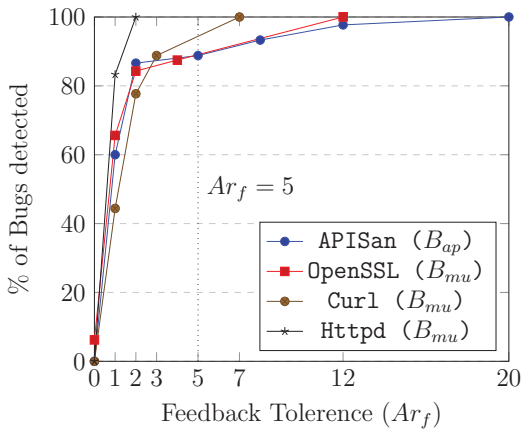


Figure 8: The percentage of bugs in bug-benchmarks detected by DRAKE with varying feedback tolerance (Ar_f).

the feature `share_arg_with_target` is `true` for the `memset` call. However, at the location shown in Listing 3 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:

- *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 DRAKE 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, DRAKE can capture these by using a large set of semantic features. Listing 4 shows a missing initialization bug found by DRAKE. This bug requires understanding the causal relationship between `cdev_add` and `cdev_init`, which APISAN failed to infer.

4.1.4 Scalability

As mentioned in Section 3.3, execution of DRAKE occurs in two independent steps, i.e., Trace generation (or Analysis setup) and Active learning.

```
// All fields are uninitialized
struct hci_request rq;

memset is missing.
memset(&rq, 0, sizeof(rq));
// Only few fields are initialized
rq.ogf    = OGF_LINK_CTL;
rq.event  = EVT_AUTH_COMPLETE;
rq.cparam = &cp;
rq.rparam = &rp;
rq.rlen   = EVT_AUTH_COMPLETE_SIZE;
// uninitialized fields of rq may be
// be used by the method
if (hci_send_req(dd, &rq, to) < 0)
  return -1;
```

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

```
cdev = cdev_alloc();
if (!cdev) {
  pr_err("Could not allocate cdev for minor %d, %s\n",
         minor, name);
  ret = -ENOMEM;
  goto done;
}

cdev->owner = THIS_MODULE;
cdev->ops = fops;
kobject_set_name(&cdev->kobj, name);

missing call to cdev_init
ret = cdev_add(cdev, dev, 1);
```

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

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.

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 DRAKE 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].

4.1.5 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 DRAKE to find the remaining bugs faster?

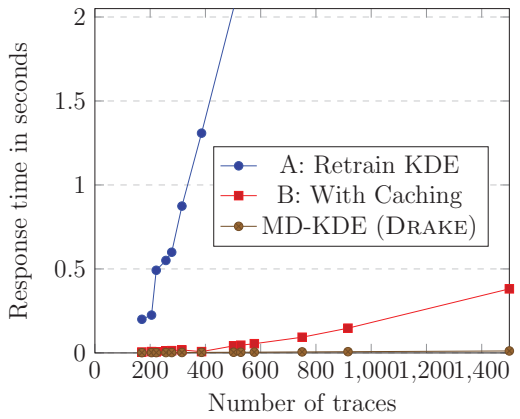


Figure 9: The response time of the update used by DRAKE compared with Naïve approaches. A: Train a brand new KDE model in each iteration; B: Caches intermediate result but recalculate score for all points in each iteration; MD-KDE: Performs only $\mathcal{O}(|\mathcal{U}|)$ updates in each iteration.

Table 5: Compare of our original setup vs. with example bug setup.

API	Dataset	Original			With Sample Bug		
		y	x_f	x_l	y'	x'_f	x'_l
`kmalloc`	APISAN	2	12	14	1	6	6
`kthread_run`	APISAN	2	9	10	1	1	1
`BN_CTX_get`	APIMU4C OpenSSL	3	13	15	2	1	2
`i2c_add_adapter`	Ours (Kernel 5.7)	11	10	24	10	3	15

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_l). Note that the selected sample bug will not be counted anymore, therefore the number of bugs y' in the with-sample-bug s etting will be off-by-1 f rom the original number of bugs

y.

Table 5 shows 4 APIs we choose, the number of bugs y, and our measurement x_l 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 DRAKE to find bugs even faster and with higher precision.

4.1.6 Discussion on false positives

We want to emphasize that our notion of false positive rate is fundamentally different from traditional bug finding tools. First, DRAKE 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 DRAKE 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 DRAKE, as shown in Section 4.1.5, each false positive marked by the user contributes to improving the accuracy of the model and consequently decreases the false-positive rate.

4.2 Limitations

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

- *Requirement of API method:* Unlike APISAN, DRAKE is API method-specific and requires the method name to be provided. We provide a way [38] 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 [32] can be made on the codebase to determine the methods of interest, which can be then provided to DRAKE.

- *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 [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 [30] 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 `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.

4.3 Future Work

In our future work, we plan to extend DRAKE 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 `malloc` could work for other allocator functions such as `realloc` and `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 high severity bugs. To handle this, we plan to allow the user to provide bug 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.

4.4 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 3.1). 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 4.1, 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 4.1, ApiSan has a high false positive rate for complex APIs. Machine learning techniques such as Apriori algorithm [60, 40], clustering [66, 65], and kNN [7] 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 Vccfinder [48] and Vuldeepecker [36] also require a large bug corpus and hence are infeasible for API misuse detection. Unlike existing approaches, Drake 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], Drake

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, DRAKE does not require any analysis specification.

5 Conclusions

We presented DRAKE, a user-guided approach for finding API misuses. DRAKE 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 DRAKE 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.

References

[1] N. Abe, B. Zadrozny, and J. Langford, "Outlier detection by active learning," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006, pp. 504–509.

[2] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases: The Logical Level*, 1st ed. Pearson, 1994.

[3] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining api patterns as partial orders from source code: from usage scenarios to specifications," 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*, 2007, pp. 25–34.

[4] P. Avgustinov, O. de Moor, M. P. Jones, and M. Schäfer, "QL: Object-oriented queries on relational data," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2016, pp. 2:1–2:25.

[5] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 73–85, 2006.

[6] P. Bian, B. Liang, W. Shi, J. Huang, and Y. Cai, "Nar-miner: discovering negative association rules from code for bug detection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 411–422.

[7] P. Bian, B. Liang, Y. Zhang, C. Yang, W. Shi, and Y. Cai, "Detecting bugs by discovering expectations and their violations," *IEEE Transactions on Software Engineering*, vol. 45, no. 10, pp. 984–1001, 2018.

[8] F. Brown, D. Stefan, and D. Engler, "Sys: a static/symbolic tool for finding good bugs in good (browser) code," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 199–216.

[9] E. N. Ceesay, J. Zhou, M. Gertz, K. Levitt, and M. Bishop, "Using type qualifiers to analyze untrusted integers and detecting security flaws in c

programs," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2006, pp. 1–16.

[10] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, pp. 1–58, 2009.

[11] H. Chen and D. Wagner, "Mops: an infrastructure for examining security properties of software," in *Proceedings of the 9th ACM conference on Computer and communications security*, 2002, pp. 235–244.

[12] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016. [Online]. Available: https://doi.org/10.1145/2970276.2970347

[13] D. A. Cohn, Z. Ghahramani, and M. I. Jordan, "Active learning with statistical models," *Journal of artificial intelligence research*, vol. 4, pp. 129–145, 1996.

[14] E. Coppa, D. C. D'Elia, and C. Demetrescu, "Rethinking pointer reasoning in symbolic execution," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 613–618.

[15] CWE, "Cwe-401: Missing release of memory after effective lifetime." 2006, https://cwe.mitre.org/data/definitions/401.html.

[16] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5, pp. 57–72, 2001.

[17] D. Evans, J. Guttag, J. Horning, and Y. M. Tan, "Lclint: A tool for using specifications to check code," *ACM SIGSOFT Software Engineering Notes*, vol. 19, no. 5, pp. 87–96, 1994.

[18] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The most dangerous code in the world: validating ssl certificates in non-browser software," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 38–49.

[19] GitHub, "Semmle - a code analysis platform for finding zero-days and automating variant analysis." 2017, https://semmle.com/.

[20] ——, "The bug slayer." 2020, https://securitylab.github.com/bounties.

[21] N. Görnitz, M. Kloft, K. Rieck, and U. Brefeld, "Active learning for network intrusion detection," in *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*, 2009, pp. 47–54.

[22] ——, "Toward supervised anomaly detection," *Journal of Artificial Intelligence Research*, vol. 46, pp. 235–262, 2013.

[23] T. J. Green, "LogiQL: A declarative language for enterprise applications," in *Proceedings of the Symposium on Principles of Database Systems (PODS)*, 2015, pp. 59–64.

[24] Z. Gu, J. Wu, C. Li, M. Zhou, Y. Jiang, M. Gu, and J. Sun, "Vetting api usages in c programs with imchecker," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 2019, pp. 91–94.

[25] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu, "An empirical study on api-misuse bugs in open-source c programs," in *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1. IEEE, 2019, pp. 11–20.

[26] R. Hat, "Advanced package tool." 2010, http://apt-rpm.org/scripting.shtml.

[27] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting ssl usage in applications with sslint," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 519–534.

[28] K. Heo, M. Raghothaman, X. Si, and M. Naik, "Continuously reasoning about programs using differential bayesian inference," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 561–575.

[29] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial intelligence review*, vol. 22, no. 2, pp. 85–126, 2004.

[30] P. Hu, Z. C. Lipton, A. Anandkumar, and D. Ramanan, "Active learning with partial feedback," *arXiv preprint arXiv:1802.07427*, 2018.

[31] K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "Fuzzgen: Automatic fuzzer generation," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020.

[32] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 2006, pp. 81–90.

[33] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler, "From uncertainty to belief: Inferring the specification within," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 161–176.

[34] M. Law, A. Russo, and K. Broda, "Inductive learning of answer set programs," in *Proceedings of the European Conference on Logics in Artificial Intelligence (JELIA)*, 2014, pp. 311–325.

[35] O. Legunsen, W. U. Hassan, X. Xu, G. Roşu, and D. Marinov, "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, 2016, pp. 602–613.

[36] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.

[37] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.

[38] Z. Li, "Count number of occurrences of an api." 2021, https://github.com/petablox/arbitrar/blob/master/doc/how_to_use.md#20-get-occurrences.

[39] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 75–86, 2009.

[40] B. Livshits and T. Zimmermann, "Dynamine: finding common error patterns by mining software revision histories," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 296–305, 2005.

[41] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "{DR}.{CHECKER}: A soundy analysis for linux kernel drivers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 1007–1024.

[42] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 462–473.

[43] M. Martin, B. Livshits, and M. Lam, "Finding application errors and security flaws using PQL: a program query language," in *Proceedings of the ACM International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005, pp. 365–383.

[44] C. Min, S. Kashyap, B. Lee, C. Song, and T. Kim, "Cross-checking semantic correctness: The case of finding file system bugs," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 361–377.

[45] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 166–177.

[46] NIST, "National vulnerability database." 2020, https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=incorrect+usage&search_type=all.

[47] J. Pei, H. Wang, J. Liu, K. Wang, J. Wang, and P. S. Yu, "Discovering frequent closed partial orders from strings," *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 11, pp. 1467–1481, 2006.

[48] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[49] M. Raghothaman, S. Kulkarni, K. Heo, and M. Naik, "User-guided program reasoning using bayesian inference," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2018, pp. 722–735.

[50] M. Raghothaman, J. Mendelson, D. Zhao, M. Naik, and B. Scholz, "Provenance-guided synthesis of datalog programs," in *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, 2020, pp. 62:1–62:27.

[51] D. A. Ramos and D. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 49–64.

[52] N. Redini, A. Machiry, D. Das, Y. Fratantonio, A. Bianchi, E. Gustafson, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Bootstomp: on the security of bootloaders in mobile devices," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 781–798.

[53] B. Schölkopf, R. C. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt, "Support vector method for novelty detection," *Advances in neural information processing systems*, vol. 12, pp. 582–588, 1999.

[54] U. Shankar, K. Talwar, J. S. Foster, and D. A. Wagner, "Detecting format string vulnerabilities with type qualifiers." in *USENIX Security Symposium*, 2001, pp. 201–220.

[55] Y. Smaragdakis and M. Bravenboer, "Using Datalog for fast and easy program analysis," in *Proceedings of the Datalog 2.0 Workshop*, 2010.

[56] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*. ACM, 2016, pp. 265–266.

[57] D. M. Tax and R. P. Duin, "Support vector data description," *Machine learning*, vol. 54, no. 1, pp. 45–66, 2004.

[58] L. Torvalds, "Sparse - a semantic parser for c." 2013, https://sparse.wiki.kernel.org/index.php/Main_Page.

[59] B. A. Turlach, "Bandwidth selection in kernel density estimation: A review," in *CORE and Institut de Statistique*. Citeseer, 1993.

[60] A. I. Verkamo, "Efficient algorithms for discovering association rules," in *KDD Workshop*, 1994.

[61] H. Vijayakumar, X. Ge, M. Payer, and T. Jaeger, "{JIGSAW}: Protecting resource access by inferring programmer expectations," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 973–988.

[62] X. Wang, H. Chen, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Improving integer security for systems with {KINT}," in *Presented as part of the 10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, 2012, pp. 163–177.

[63] A. Wasylkowski, A. Zeller, and C. Lindig, "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*, 2007, pp. 35–44.

[64] J. Whaley and M. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2004, pp. 131–144.

[65] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 797–812.

[66] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 499–510.

[67] YAML, "Yaml ain't markup language." 2001, https://yaml.org/.

[68] Z. Yu, C. Theisen, L. Williams, and T. Menzies, "Improving vulnerability inspection efficiency using active learning," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[69] I. Yun, C. Min, X. Si, Y. Jang, T. Kim, and M. Naik, "Apisan: Sanitizing {API} usages through semantic cross-checking," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 363–378.

[70] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "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, 2018, pp. 886–896.

[71] X. Zhang, R. Grigore, X. Si, and M. Naik, "Effective interactive resolution of static analysis alarms," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–30, 2017.

A Appendix

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

```
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.getLValue().(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

Acronyms

AD	Anomaly Detection
ALPF	Active learning with partial feedback
API	Application Programming Interface
APIMU4C	API Mis-Use for C
APISAN	Not an Acronym (API Sanitizing Tool)
CodeQL	Not an Acronym
Datalog	Not an Acronym
DRAKE	Not an Acronym
DSL	Domain Specific Language
FRECPO	Frequent Closed Partial Order
GB	Gigabyte
IAD	Interactive Anomaly Detection
IMChecker	Not an Acronym
IR	Intermediate Representation
JADET	Not an Acronym
JIGSAW	Not an Acronym
JUXTA	Not an Acronym
KDE	Kernel Density Estimation
kNN	K-Nearest Neighbor
LLVM	Lower Level Virtual Machine
LOC	Lines of Code
LogiQL	Not an Acronym
MD-KDE	Multi-Dimensional Kernel Density Estimation
OpenSSL	Open Secure Socket Layer
PNG	Portable Network Graphics
PQL	Process Query Language
RAM	Random Access Memory
ReverseBFS	Reverse Breadth First Search
Semmle	Not an Acronym
SVM	Support Vector Machine
TSNE	Not an Acronym
YAML	Yet Another Markup Language