An Empirical Study on API-Misuse Bugs in Open-Source C Programs

Zuxing Gu*, Jiecheng Wu*, Jiaxiang Liu†, Min Zhou*, Ming Gu*
*School of Software, Tsinghua University, China
†College of Computer Science & Software Engineering, Shenzhen University, China

***Abstract*—Today, large and complex software is developed with integrated components using application programming interfaces (APIs). Correct usage of APIs in practice presents a challenge due to implicit constraints, such as call conditions or call orders. API misuse, i.e., violation of these constraints, is a well-known source of bugs, some of which can cause serious security vulnerabilities. Although researchers have developed many API-misuse detectors over the last two decades, recent studies show that API misuses are still prevalent.**

In this paper, we provide a comprehensive empirical study on API-misuse bugs in open-source C programs. To understand the nature of API misuses in practice, we analyze 830 API-misuse bugs from six popular programs across different domains. For all the studied bugs, we summarize their root causes, fix patterns and usage statistics. Furthermore, to understand the capabilities and limitations of state-of-the-art static analysis detectors for API-misuse detection, we develop APIMU4C, a dataset of API-misuse bugs in C code based on our empirical study results, and evaluate three widely-used detectors on it qualitatively and quantitatively. We share all the findings and present possible directions towards more powerful API-misuse detectors.

***Index Terms*—API misuse, empirical study, benchmark, bug detection**

I. Introduction

Large-scale software is often achieved by the use of frameworks and libraries, which provide application programming interfaces (APIs). To correctly use these APIs, programmers must conform to their *usage constraints*, such as call conditions on parameters or call orders. While high-quality documentation of an API's usage constraints could help to guide correct usages, it is often insufficient, at least in its current form to solve the problem [1]. Violation of such usage constraints, called *API misuse* [2], is a prevalent cause of software bugs, crashes, and vulnerabilities [3], [4]. For example, missing parameter validation of `PKCS7_dataInit()` of OpenSSL in CVE-2015-0289[1], allows remote attackers to cause a denial of service by maliciously crafted data. It affected more than 33 released versions of OpenSSL and threatened almost all Linux distributions.[2]

Over the last two decades, numerous tools, techniques, and methodologies have been proposed to address the problem of API misuse. For example, some methods have been developed to recommend correct usage of API parameters [5] and calling locations [6] to prevent misuses. And others aim at finding API misuses through code review [7], runtime verification [8] and static analysis [9]. However, recent studies show that API misuses remain widespread [10], [11]. Particularly, a recent study on 11748 Android applications on Google Play marketplace shows that 10327, overall 88%, misused at least one cryptography-related API [12]. Moreover, these bugs are even in source code written by experienced developers [13] and bug fix patches. For example, a developer of OpenSSL created a patch (sha:1c4221) to "*fix memory leak in crl2pkcs7 app*", but he forgot to consider all branches, resulting in a double free problem in one branch. Later, a commit (sha:d285b5) was patched to "*avoid a double-free in crl2pl7*".

To better combat API-misuse bugs in practice, our community urgently needs a thorough understanding of the characteristics of API-misuse bugs as well as current limitations of existing approaches to API-misuse detection. Particularly, we focus on API misuses in open-source C programs and free static analysis detectors. Static analysis detectors have become an essential pillar of modern software quality assurance approaches, since they only require access to source code [14], typically available early in the development process. It leads to cost savings as the earlier a bug is detected, the cheaper it is to fix [15]. Therefore, this paper aims to address the following research questions:

- RQ1 (API-Misuse Characteristic): What are the characteristics of API-misuse bugs? While existing studies have proposed several software defect classifications [9], [16], [17], they give no insights into the root causes and fix patterns specific to API-misuse bugs in real-world C programs. Understanding the nature of API-misuse bugs is essential for deriving guidance towards better API design as well as more powerful bug detectors.
- RQ2 (Detection Capability): What kind of API-misuse bugs can(not) the state-of-the-art static analysis detectors find? To advance the state-of-the-art in API-misuse detection, we need to understand how existing approaches compare to each other, and what their limitations are. The results would help researchers to improve API-misuse detectors by enhancing current strengths and proposing new approaches to overcome weakness.

In this work[3], we conduct an empirical study on API-misuse bugs to answer these research questions. For RQ1, we

Corresponding author: Jiaxiang Liu, jiaxiang.liu@hotmail.com

[1]https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0289

[2]https://www.cvedetails.com/cve/CVE-2015-0289/

[3]We publish all the original data and artifacts in a temporary repository for double-blind review process at: https://github.com/imchecker/compsac19

analyze 830 API-misuse bugs from six popular open-source C programs across different domains (including operating systems, libraries, and applications) to understand the true nature of API-misuse bugs in real-world C programs. We thoroughly study each of these bugs to summarize the root causes and fix patterns. Furthermore, to assess API-misuse detection capabilities of static analysis detectors, we develop APIMU4C, a dataset of API-misuse bugs in C code based on the summarized root causes. APIMU4C consists of 2172 test cases modified from two widely-used benchmarks (i.e., Juliet Test Suite [18] and ITC [19]) and 100 cases from three real-world programs (OpenSSL [20], Curl [21] and Httpd [22]). Using APIMU4C, we compare three widely-used static analysis detectors which support multiple types of API-misuse bugs, qualitatively and quantitatively.

Through our in-depth analysis of the above research questions, we obtain many interesting findings. We summarize our main findings as follows:

- API-misuse bugs are not corner-cases (17.05% of bug-fix related commits). Even though the distribution of root causes is different among programs, there are mainly three generic types of patterns, i.e., improper parameter using (IPU, 16.27%), improper error handling (IEH, 30.96%) and improper causal function calling (ICC, 36.02%). These bug patterns can motivate new specific-purpose approaches to detect API-misuse bugs.
- Almost all (96.15%) API-misuse bugs can be fixed within 10 lines, such as adding sanity checks of parameters, correctly validating error code status, and adding missing function calls. However, it may require control-flow, data-flow, and complex project-specific semantics to fix these bugs correctly. It indicates that new approaches are in great need to help developers understand the context of API-misuse bugs, and to provide automatic repair suggestions.
- For the misused APIs, 21.45% of them have been incorrectly used more than once in its corresponding program, i.e., at least two commits to fix misuses of an API with the same pattern in different program locations. It indicates that an API misuse tracking system may be required to recommend and remind developers, especially for the APIs of widely-used third-party libraries.
- Static analysis detectors are capable of detecting API-misuse bugs, but they perform lower (missing 17-54% test cases respectively) than the expectations from their tool descriptions or publications. To improve precision, detectors need to consider more semantics (e.g., path-sensitive and inter-procedural), instead of simply syntactic checking. To improve recall, static analysis detectors based on encoded bug patterns could provide a user-friendly interface to specify usage constraints for project-specific APIs. For detectors employing mining techniques, they need to go beyond the naive assumption that a deviation from the most frequent usage corresponds to a misuse, because it may lack sufficient data for an individual program in a realistic setting to infer correct models. Building probabilistic models or providing interfaces to analyze multiple programs simultaneously could be a way to eliminate or reduce insufficient threats.

In summary, we make the following main contributions:

- We present the first comprehensive study of API-misuse bugs in open-source C programs. Our in-depth analysis of 830 API-misuse bugs from six large-scale programs across different domains reveals common root causes of API misuses.
- We provide APIMU4C, a benchmark for API-misuse detectors in C programs, which could facilitate systematic and reproducible evaluation of misuse detectors.
- We perform a qualitative and quantitative comparison of three existing open-source static detectors with different analysis strategies on APIMU4C. We manually analyze the merit and demerit of these detectors and hope our study can shed new light on combating API-misuse bugs.

The rest of the paper is organized as follows. Section II introduces the background and terminology of API misuse. We illustrate the methodology in Section III. All of the results are presented in Section IV. We discuss the related work in Section V and conclude in Section VI.

II. Background and Terminology

An API *usage* is a piece of code which aims at invoking a specific API to accomplish a task. It usually combines basic program elements to ensure the correct usage, such as checking parameter properties and error status code encapsulated in return values, and calling function calls. The combination of these elements is subject to *usage constraints* or *specifications*, which depend on the nature of the API. When a usage violates one or more such constraints, we call it a *misuse*.

The automatic detection of API misuses can be approached by *static analysis* on source code or *dynamic analysis* on runtime logs and traces. In this paper, we focus on static analysis detectors and call such detectors as *API-misuse detectors* (detectors for short). To find API misuses, detectors require either specifications of correct usage to detect violations or specifications of incorrect usage to match instances. Such specifications can be *crafted manually* by domain experts and hard-coded into detectors by developers. During the analysis of the source code, detectors employ different representations (e.g., abstract syntax tree, control flow graph, self-defined intermediate representation, etc.) and detection techniques (e.g., abstract interpretation, string pattern matching, etc.) to find violations. Since manually maintaining specifications is costly, there have been many attempts to *automatically infer* such specifications. The key idea of these detectors is that a deviation from the most frequent usage corresponds to a misuse. In this paper, we will evaluate both of these two kinds of detectors.

TABLE I: Empirical Study Subjects

Project	Loc	Studied Period	Total	Bug Fix	API Misuse
Linux [23]	12.96M	20170901-20171231	24651	6401	868
OpenSSL [24]	454K	20150701-20171231	7564	2391	529
FFmpeg [25]	915K	20160701-20171231	8162	2783	610
Curl [21]	113K	20130101-20170630	7082	2043	499
FreeRDP [26]	259K	20130701-20171231	7565	3535	495
Httpd [22]	203K	20130701-20171231	6072	1323	149
Total	14.90M	-	61096	18476	3150

III. Methodology

This section introduces our major steps to collect and analyze API-misuse bugs, and experimental setup to evaluate existing detectors on API-misuse bugs.

A. Bug Collecting

1) Target Programs: To better understand what kind of API-misuse bugs occur in real-world C programs and how developers fix them in practice, we investigate these bugs from six popular open-source programs as listed in Table I.

These programs represent different domains, including operating systems, widely-used libraries, and applications using studied libraries. Also, they are under ongoing development, receiving considerable attention on GitHub and frequently mentioned in bug detection literature. All six programs have well-organized and publicly available commits tracing repositories on Github.

2) Collecting API-Misuse Bugs: To collect API-misuse bugs from these six programs, we extract and analyze version histories. First, we begin with retrieving all commits with commit messages, patch files and other meta information within the studied period for each program. We remove the commits without modification to *.c files. We randomly chose 600 commits (100 for each program) to summarize bug-fix related keywords (e.g., "*bug*", "*error*", "*fix*", and "*check*", etc.) and API-misuse related keywords (e.g., "*fix API*", "*missing check*", "*null pointer dereference*", "*memory leak*", "*return value*", etc.) in commit messages. Then, we collect bug-fix related commits and further search for API-misuse related ones by approximate string matching with keywords. For example, we identified a Linux kernel commit (sha: 059c98599) with the commit message "*add checks on wl18xx_top_reg_write() return value*" as a target commit.

We randomly chose 180 commits (30 from each program) that were marked as API-misuse related commits to evaluate the precision of the above approach. By manually verifying the commit messages and their corresponding patches, 166 (92.22%) of 180 are API-misuse related. In all wrongly extracted commits, although the relevant keywords are present, they were not related to API-misuse bugs. For example, one commit of OpenSSL (sha: 4af389) with commit message "*Fix compilation with OPENSSL API COMPAT=XXX*" was patched to fix a compilation bug.

Totally, we extract 3150 API-misuse related patches from 18476 bug-fix related commits as shown in Table I.

B. Analyzing API-Misuse Bugs

To answer our research questions, we perform an in-depth analysis of these API-misuse bugs according to commit messages, patches, source code, and documentation. Note that manually understanding a bug without context semantics and project-specific domain knowledge requires a mass of time and efforts. Therefore, we randomly investigated 830 true API-misuse bugs (see Table III in Section IV-A1). For each bug, we endeavour to understand the misuse context, assign them into different categories according to the root causes, and summarize fix patterns. In addition, we calculate the statistics of these misused APIs. Through analyzing these 830 bugs, we obtain many interesting findings (see Section IV-A). We further employ these findings to guide the evaluation of existing detectors.

TABLE II: Combination of APIMU4C

Location	# of Cases	Type	Loc
Single file	2172	manual craft	100∼200
OpenSSL	50	real-world patch	454K
Curl	30	real-world patch	113K
Httpd	20	real-world patch	203K
Summary	2272	Covering all the summarized root causes	

C. Detector Evaluation Setup

To advance state-of-the-art API-misuse detection, we need to understand the capabilities and shortcomings of existing detectors.

1) Dataset and Metrics: To the best of our knowledge, there is no existing benchmark towards API misuses for C code. In this paper, we create APIMU4C, a dataset of API misuses that can be used to benchmark and compare API-misuse detectors according to the root causes of our empirical study on API-misuse bugs. APIMU4C consists of two parts as listed in Table II: (1) 2172 manually crafted cases modified from two widely-used benchmarks (i.e., Juliet Test Suite [18] and Toyota ITC [19]). Each of these cases includes a main function invoking one bad caller with one misuse and several good callers with correct usages within small pieces of code. They are designed to capture the essence of bugs occurring in real-world code, and covering syntax structures of C language as many as possible. It will be useful to understand the analysis strategy and supported language features of detectors. (2) 100 real-world cases injected into three open-source projects (i.e., OpenSSL, Curl, and Httpd) according to their bug-fix patches. It will be useful to evaluate the detectors' capabilities on real-world programs.

We employ three metrics to compare the existing static analysis detectors on APIMU4C:

- Conceptual-Recall(CR): CR is the upper bound of recall for a detector, i.e., the conceptual capability of a subject detector calculated offline, where we assume that the detector always can find the bugs.
- $Recall = \frac{\text{\# of API misuses in reports}}{\text{\# of all API misuses}}$: It is important for developers to know the detection capabilities of existing tools to choose adequate tools, as well as for researchers

to direct future work. An ideal detector should have an empirical recall upper bound equal to its conceptual recall upper bound.
- $Precision(P) = \frac{\text{\# of API misuses in reports}}{\text{\# of reports}}$: Past studies show that developers rarely use analysis tools that produce many false positives [27]. Therefore, for a detector to be adopted in practice, it needs a high precision.

2) Subject Detectors: In this study, we focus on static detectors supporting API-misuse detection of C code. We contact authors of related publications and search free tools online. Currently, we select three widely-used tools with different detection techniques and strategies. All of the three tools provide a user-friendly client with well-format bug reports and support multiple types of API misuses. Also, they achieve a stable performance in the preliminary experiment and are capable of detecting bugs in real-world programs.

- APISan Master Branch at 2018-06-01 [28]: is an academic tool designed for API-misuse bug detection, which combines code mining with static analysis techniques. It extracts likely correct usage patterns intra-procedurally by considering semantic constraints, including implications of function return values, relations between function arguments, causal relationships between functions, and implicit pre- and post-conditions of functions.
- Cppcheck Version 1.83 [29]: is an open-source tool to find undefined behavior and dangerous coding constructs. It works by splitting source code into tokens and finding suspicious patterns in the tokens with flow-, context-sensitive analysis inter-procedurally. Beyond the core of static analysis engine, Cppcheck also provides interfaces to write rules for project-specific APIs. Therefore, we manually create these rules for APIMU4C.[4]
- Clang-SA Version RELEASE_600: is an open-source tool of LLVM[5]. It employs symbolic execution to reason about the semantics with flow-, context-sensitive analysis inter-procedurally. It provides a number of checkers to support different kinds of API-misuse bugs.[6]

D. Threats to Validity

We select API-misuse bugs from six open-source programs for a given period. So it does not cover all types of programs, and the bugs and patches beyond the studied period are not included. We employ keywords to extract target bug patches. However, developers may not put the keywords that we look for in the commit messages. Nonetheless, these six programs are widely-used, covering a diverse set of domains, and totally 830 commits within a five-year period have been studied. We conduct a preliminary experiment to find keywords on 600 bug-fix related patches across all six programs. This makes it unlikely that we miss a prevalent category of API misuse. Our manual understanding may be affected by authors' biases. For each bug, we carefully study its description, patches and discussions among developers, and read source code to have a deep understanding. All studied bugs have been discussed and confirmed by at least two authors in this paper. Thus, we believe that all studied bugs are true positives and have been thoroughly studied.

Our study focuses on static detectors for C code. Approaches based on other techniques and another language may perform differently and have unique strengths and weaknesses. Performance of a detector depends on its configuration. Due to the high effort of reviewing findings, we could not try each combination. To give a fair chance for each detector, we selected the optimal configurations reported in the respective publications and employed all the diverse checkers supported by the tools. We evaluate on APIMU4C, which may not have enough examples to cover all kinds of API misuses with full syntax structures of C language. We publish all the original data, APIMU4C, tools' results to encourage researchers and developers to extend our experiments as well as to design more powerful approaches to combat API misuses.

IV. Results

A. API Misuse Characteristics

We totally extracted 3150 (17.05% on average) API-misuse related commits from 18476 bug-fix related commits in six open-source programs as shown in Table I. As discussed in Section III-A2, our approach achieves an approximate precision of 92.22% by manually verifying a group of sample instances. Therefore, we believe that API misuses are not corner-case, but prevalent (considering that many misuses are still hiding in source code).

> ***Finding 1:*** *API-misuse bugs are not corner-case in source code. Approximately, 17.05% of bug-fix related commits are patched for API misuses.*

To understand the true nature of API-misuse bugs, we randomly investigated 830 instances as listed in Table III. We summarize characteristics of these misuses, including the root causes, fix pattern and usage statistics.

1) Root Causes: Even though the distribution of root causes is different across programs, we identify three generic categories of API-misuse, improper parameter using (IPU), improper error handling (IEH) and improper causal function calling (ICC), as listed in Table III. We used the code snippets from bug-fix patches to discuss the details of these categories below.

Improper Parameter Using (IPU): APIs simplify programming by abstracting underlying implementation details for a specific target. Certain conditions must hold whenever an API is invoked, which are called *preconditions* [30]. For example, when an argument in method calls is a pointer type, it usually has to be ensured that the pointer is not NULL. However, we find that developers often forget to guarantee this type of constraints. For example, function `strchr`(*str*, *c*) searches for the first occurrence of the character *c* (an unsigned char) in the string pointed to by the argument *str*. Without validating

[4] http://cppcheck.sourceforge.net/manual.pdf
[5] https://llvm.org/
[6] http://clang-analyzer.llvm.org/available_checks.html

TABLE III: Investigated API-misuse Bugs and Patches

Project	# of Bug Fix	API Misuse		Investigated		IPU		IEH		ICC		Other	
		# of	$Rate_1$	# of	$Rate_2$	# of	$Rate_3$	# of	$Rate_3$	# of	$Rate_3$	# of	$Rate_3$
Linux	6401	868	13.56%	283	32.60%	43	15.19%	96	33.92%	77	27.21%	67	23.67%
OpenSSL	2391	529	22.12%	127	24.00%	21	16.54%	42	33.07%	49	38.58%	15	11.81%
FFmpeg	2783	610	21.92%	126	20.66%	18	14.29%	43	34.13%	52	41.27%	13	10.32%
Curl	2043	499	24.42%	134	26.85%	23	17.16%	38	28.36%	57	42.54%	16	11.94%
FreeRDP	3535	495	14.00%	119	24.04%	22	18.49%	30	25.21%	48	40.34%	19	15.97%
Httpd	1323	149	11.26%	41	27.52%	8	19.51%	8	19.51%	16	39.02%	9	21.95%
Total	18476	3150	17.05%	830	26.35%	135 (16.27%)		257 (30.96%)		299 (36.02%)		139 (16.75%)	

$Rate_1$= # of API-misuse commits / # of bug-fix commits, $Rate_2$= # of investigated commits / # of API-misuse commits
$Rate_3$= # of each category / # of investigated commits for each project

```
1  FreeRDP: libfreerdp/core/gateway/http.c
2  CommitID: 9e5be6f7e8*33d68012f6
3  Log: Fixed API nonnull warning.
4  void OPENSSL_config(const char *config_name){
5     ...
6     // missing parameter validation
7  -  colon_pos = strchr(line, ':');
8  +  if (line)
9  +      colon_pos = strchr(line, ':');
10 +  else
11 +      colon_pos = NULL;
12    ...
```

(a) Single parameter validation.

```
1  FFmpeg: libavcodec/libxvid_rc.c
2  CommitID: f7d183f084*a12670ed0e
3  Log: Check return value of write() call
4     ...
5  // missing valiation between parameter and return value
6  -  write(fd, tmp, strlen(tmp));
7  +  if (strlen(tmp) > write(fd, tmp, strlen(tmp))) {
8  +      av_log(s, AV_LOG_ERROR, "MSG");
9  +      return AVERROR(EIO);}
10    ...
```

(b) Parameter validation with inter-relation of return value.

Fig. 1: Examples of IPU bugs and fix patches.

the argument *line* at Line 7 in Figure 1a, it may result in a null pointer dereference bug.

Relations among arguments should be taken into consideration in the meantime. Typical examples are memory operation APIs, such as `memcpy`(*d, s, n*), where the size of destination buffer *d* should be equal to or larger than the copy length *n*. In addition, a parameter may also be semantically inter-related to the return value. For example, function *size_t* `write`(*int fd, void* buf, size_t cnt*) writes *cnt* bytes from *buf* to the file or socket associated with *fd*. The return number ret of `write()` records how many bytes are actually written, indicating that ret has to be checked with *cnt* to ensure the success of the written operation as shown in Figure 1b.

> ***Finding 2:*** *In the investigated API-misuse bugs, 14.29-19.51% are caused by improper parameter using (IPU), including missing validation of a single parameter, inter-relations among parameters as well as the return value.*

Improper Error Handling (IEH): Secure and reliable software should handle all possible failure conditions correctly. Unfortunately, C does not provide any error handling primitives. Developers usually employ certain values conventionally to represent the execution status, especially for large-scale systems [31]. Thus, after calling the target function f, the caller c should check its return value properly before proceeding. However, we find that developers often forget to check the API return values. Figure 2a shows such an example from Curl. Function `Curl_client_write()` sends data to the write callbacks, where the return value records a predefined error status code. The caller function did not check whether the API

```
1  Curl: ib/ldap.c
2  CommitID: 086ad79970*6acdc9d2da
3  Log: check Curl_client_write() return codes
4     ...
5  -  Curl_client_write(p1, p2, p3, 4);
6     ...
7  +  result = Curl_client_write(p1, p2, p3, 4);
8  +  if(result)
9  +     goto quit;
10    ...
```

(a) Missing error status code checking.

```
1  Curl: lib/ssluse.c
2  CommitID: 520833cbe1*c894e7ee63e
3  Log: SSL_read() returning 0 is an error too
4     ...
5     nread = (ssize_t)SSL_read(p1, buf, buffsize);
6     // incorrect checking error code status
7  -  if(nread < 0) {
8  +  if(nread <= 0) {
9     ...
```

(b) Incorrect checking of error status code.

```
1  Curl: lib/hostip.c
2  CommitID: eb5199317e*0ca7f7aa1e
3  Log: add error message when resolving using SIGALRM
4     ...
5  -  if(timeout < 1000)
6  +  if(timeout < 1000) {
7  // output error message and propagate error status
8     +   failf(data, ERROR_MSG, timeout);
9     return CURLRESOLV_TIMEDOUT;
10    ...
11 Openssl: ssl/t1_lib.c
12 CommitID: 884a790e17*8c1fe18902
13 Log: Fix missing NULL checks in key_share processing
14    ...
15    skey = ssl_generate_pkey(ckey);
16 +   if (skey == NULL) {
17 +      SSLerr(SSL_F_ADD_CLIENT_KEY_SHARE_EXT,
      ERR_R_MALLOC_FAILURE);
18 +      return 0;
19 +   }
```

(c) Error handling actions when an API fails.

Fig. 2: Examples of IEH bugs and fix patches.

```
1   Openssl: crypto/objects/o_names.c
2   CommitID: 0a618df059*8bc1eeff07
3   Log: Fix a mem leak on an error path
4       ...
5       onp = OPENSSL_malloc(sizeof(*onp));
6       if (onp == NULL) {
7           /* ERROR */ return 0;
8       }
9       ... // processing
10      if (lh_OBJ_NAME_error(names_lh)) {
11  +       OPENSSL_free(onp);
12          return 0;
13      ....
```

(a) Missing resource release.

```
1   Openssl: apps/crl2p7.c
2   CommitID: d285b5418e*e61bdd1a50
3   Log: Avoid a double-free in crl2pl7
4       ...
5       if ((certflst == NULL) && (certflst =
6               sk_OPENSSL_STRING_new_null()) == NULL)
7           goto end;
8   -   if (!sk_OPENSSL_STRING_push(certflst, opt_arg())) {
9   -       sk_OPENSSL_STRING_free(certflst);
10  +   if (!sk_OPENSSL_STRING_push(certflst, opt_arg()))
11          goto end;
12      ...
13  end:
14      sk_OPENSSL_STRING_free(certflst);
15      ...
```

(b) A double free bug.

Fig. 3: Examples of ICC bugs and fix patches.

returns an error code, which caused an improper error handling bug. However, properly checking the return values is not trivial in reality, since each API uses return values differently. As shown in Figure 2b, the caller of `SSL_read()` checked the return value against a negative error code. However, returning zero is also an error according to the OpenSSL specification.[7]

To correctly handle the API failures, developers usually employ two common actions in our empirical study as shown in Figure 2c: (1) Propagate error status upstream using an appropriate return value to inform the rest of the system about this failure. For instance, Curl returns CURLRESOLV_TIMEDOUT to indicate a previous alarm expired. (2) Log/output an appropriate error message so that the users become aware of the failure, such as calling `failf()` to output the messages stating why Curl fails, and calling `SSLerr()` in OpenSSL.

> ***Finding 3:*** *In the investigated API-misuse bugs, 19.51-34.13% are caused by improper error handling (IEH), including missing/incorrect checking error code status and missing/incorrect error handling actions.*

Improper Causal Function Calling (ICC): Causal relationships, also known as the a-b pattern, are common in API usage, such as lock/unlock and malloc/free. Missing the second function call will result in a resource leak error. For example, function `OPENSSL_malloc()` of OpenSSL behaves similarly to `malloc()` to allocate system resources. In Figure 3a, it forgot to `OPENSSL_free()` the *onp* along the error handling path at Line-11 when function `lh_OBJ_NAME_error()` fails, resulting in a memory leak bug.

Currently, many tools focus on finding these "direct" causal relationships, that is, no context constraint between two API calls. However, there are many constrained causal relationships in practice. For example, the non-NULL return value of `malloc()` requires a call to function `free()` to release the memory. Moreover, resource release action should semantically match the allocation. Specifically, releasing one resource multiple times without reallocation will result in a double free bug as shown in Figure 3b.

> ***Finding 4:*** *In the investigated API-misuse bugs, 27.21-42.54% are caused by improper causal function calling (ICC), including missing resources release and redundant calling.*

Others: A total of 139 patches occur due to project-specific issues which cannot be directly applied to generic API usages. For example, some patches are created to update function definitions (e.g., Curl-82232bbbaf and FreeRDP-1bca1e7820). Another motivation is to refine error handling mechanism (e.g., Httpd-b5eae6a3e2), such as changing logging functions (e.g., Linux-5b60fc0980) and error messages (e.g., OpenSSL-0cb8c9d85e). Others arise to fix typos (e.g., Curl-27ac643455), or other functionalities (e.g., FFmpeg-2dafbae994 to deprecate old APIs and Curl-4dae049157 to remove compilation warnings).

2) Bug Fixing: For each investigated API-misuse bug, we study how developers fix it and how difficult to fix it. We discuss the findings in the following part.

Fix Strategies. Fixes of API-misuse bugs are highly related to their root causes and project-specific specifications as illustrated in Figure 1-3. For misuses caused by IPU, a simple fix is to add parameter validation before the function call. For IEH bugs, developers perform sanity check against the error status code. If the error is detected, error handling actions should be performed according to project-specific specifications. For ICC bugs, developers should correctly release the resource after its lifecycle, especially along the error handling paths as shown in Figure 3a.

Fix Complexity. Table IV shows the Loc of patches for the studied API-misuse bugs. More than half (79.40%) of the misuses can be fixed within 5 lines and a majority

TABLE IV: Patch lines of studied API-misuse bugs

Project	bugs	Loc of patchs$^{\alpha}$					
		1-5	6-10	10+	Avg	Med	Max
Linux	283	225	47	11	3.88	4	14
OpenSSL	127	108	17	2	3.07	2	13
FFmpeg	126	95	26	5	4.01	3	21
Curl	134	98	26	10	4.65	3	21
FreeRDP	119	103	14	2	2.63	2	11
Httpd	41	30	9	2	4.04	3	11
Summary	830	659 (79.40%)	139 (16.75%)	32 (3.85%)	3.73	3	21

α :We count the lines starting with +/- (i.e., modifying a single line will be counted twice). For the patch fixing several calls, we use the average Loc.

[7]https://www.openssl.org/docs/manmaster/man3/SSL_read.html

```
1   Curl: lib/vtls/curl_darwinssl.c
2   CommitID: 0426670f0a*ed522feb7f
3   Log: Check CA certificate in curl_darwinssl.c
4      ...
5      SecCertificateRef cacert =
6      SecCertificateCreateWithData(kCFAllocatorDefault,
           certdata);
7      ...
8   +  // Check if cacert is valid.
9   +  SecKeyRef key;
10  +  OSStatus ret = SecCertificateCopyPublicKey(cacert,
         &key);
11  +  if(ret != noErr) {
12  +     CFRelease(cacert);
13  +     failf(data, "SSL: invalid CA certificate");
14  +     return CURLE_SSL_CACERT;
15  +  }
16  +  CFRelease(key);
17
18     CFArrayAppendValue(array, cacert);
19     CFRelease(cacert);
20     ....
```

(a) Add project-specific parameter validation.

```
1   FreeRDP: client/common/cmdline.c
2   CommitID: 1845c0b590*d7bcfb90cf
3   Log: Fixed possible memory leak.
4      ...
5      layouts = freerdp_keyboard_get_layouts(PR1);
6      ... // processing
7   +  free(layouts);
8      layouts = freerdp_keyboard_get_layouts(PR2);
9      ... // processing
10  +  free(layouts);
11     layouts = freerdp_keyboard_get_layouts(PR3);
12     ... // processing
13     free(layouts);
14     ...
```

(b) Fix memory leak with data-flow semantics.

Fig. 4: Examples of patches requiring complex and project-specific semantics.

(96.15%) of them can be fixed within 10 lines. However, it may demand great efforts to create correct patches. For example, instead of simply adding a parameter validation against a constant integer or NULL, it may require complicated semantic checkings as presented in Figure 4a. In Curl, `SecCertificateCreateWithData()` returns a non-NULL *SecCertificateRef* even if the buffer holds an invalid or corrupt certificate. Therefore, it requires to call `SecCertificateCopyPublicKey()` to make sure the return value is a valid certificate. Moreover, data-flow and context semantics should be taken into considerations during the bug fixing phase. As shown in Figure 4b, without explicitly releasing *layouts* at Line-7 and Line-10, it will cause a memory leak.

> ***Finding 5:*** *A majority (92.89%) of API-misuse bugs can be fixed within 10 lines, but bug fixing is complicated and requires amounts of efforts to understand the context semantics.*

3) Repeatability: During the investigation of API-misuse bugs, we find that one API may be misused several times, though designers of the library provide well-format documentation. For example, `BN_CTX_get()`[8] is used to obtain temporary BIGNUMs and returns NULL on error. However, three commits were patched to fix misuses in different files by two developers. In the investigated API misuses, we find that 55 different APIs (10 for Linux, 10 for OpenSSL, 14 for FFmpeg, 14 for Curl, 5 for FreeRDP and 2 for Httpd) have been misused more than once, consisting 178 (21.45%) of all investigated commits. Particularly, `calloc()` caused 21 (15.67% of misuses) memory leak bugs in FreeRDP. Besides, misusing APIs of third-party libraries are common in applications(e.g., 13 in Curl, 6 in FreeRDP and 3 in Httpd were misuses of OpenSSL APIs). For example, a commit (sha: 0b5efa57ad) of Curl was created to "Fix certificate load check" of misuse of `SSL_CTX_load_verify_locations()` in OpenSSL.

> ***Finding 6:*** *Even though developers provide well-format documentation, APIs are still misused in the same program. Moreover, part of them (17.23%) may be misused multiple times.*

B. Static Detector Performance

To understand the capabilities and limitations of state-of-the-arts, we evaluate three typical detectors on APIMU4C. That is APISan for approaches that automatically infer usage constraints from source code and report deviations, Cppcheck and Clang-SA for approaches that hard-code rules and employ program analysis techniques to reason about semantics. In addition, Cppcheck provides an interface for users to specify usage constraints of project-specific APIs. In this section, we discuss the evaluation results.

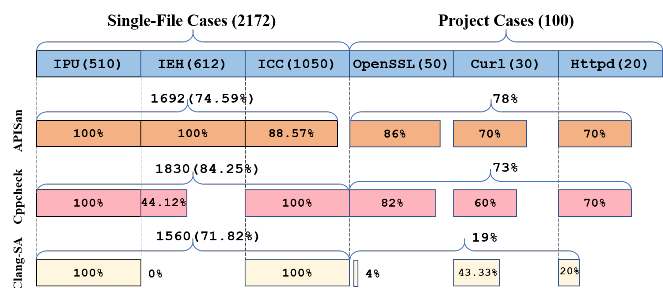

Fig. 5: Conceptual recalls of studied static detectors.

1) Conceptual Recall: We use conceptual recall (defined in Section III-C1) to assess the detection capabilities of subject detectors, i.e., to measure an upper bound to their recall under the assumption that they can always find the supported misuses. By carefully reading the academic publications and user manuals online of each tool, we provide the conceptual recalls of each detector with respect to APIMU4C. As shown in Figure 5, even though all of the three detectors are capable

[8]https://www.openssl.org/docs/manmaster/man3/BN_CTX_get.html

of multiple types of API-misuse bugs, none of them can support all test cases.

> ***Finding 7:*** *Existing static detectors can support multiple types of API-misuse bugs, but none of them can support all test cases.*

APISan detects API misuses by inferring usage constraints through semantic cross-checking. However, it fails to consider 11.43% double-free bugs of single-file ICC cases with negative association pattern like $A \rightarrow \neg B$ (i.e., when `A` appears, `B` should not appear). Moreover, it requires sufficient data to learn the correct usage patterns, (i.e., more correct usages comparing to misuses). Unfortunately, 22% of the cases in real-world programs cannot satisfy the minimum support value (i.e., at least three invocations with two correct usages).

> ***Finding 8:*** *APISan fails to support 22% real-world API-misuse bugs because of insufficient data to learn the correct usage patterns.*

Clang-SA and Cppcheck encode API usage specifications into individual checkers. Even though Clang-SA provides diverse checkers, none of them is designed to validate the error status code, resulting in missing all 612 IEH cases. Project-specific APIs usually have similar behaviors to C Standard Library. For example, functions named `*_new` are usually designed to allocate resources. Therefore, it needs to release the resources after its lifecycle. Without interfaces to specify specifications of these APIs, Clang-SA has a limited capability in real-world programs and only supports 19% of real-world cases misusing APIs provided by C Standard Library. By providing an interface to define project-specific usage constraints in an XML-format configuration, Cppcheck supports 54% more bugs compared to Clang-SA as well as itself without configurations. However, it fails to provide a mechanism to specify the concrete error status code checking (e.g., return value of f should be -1 or -2). Moreover, Cppcheck does not support the context constraints between parameters of two functions. For example, $\texttt{fNew}(p1, p2) \rightarrow \texttt{fFree}(p2)$ indicates that the resource allocated into *p2* by `fNew()` should be released by `fFree()`.

> ***Finding 9:*** *By providing an interface to specify usage constraints of project-specific APIs, Cppcheck is capable of supporting 54% more bugs compared to Clang-SA and itself without configuration.*

2) Precision and Recall: We list all the evaluation results in Table V, where *Bug* is the number of misuses in each type of APIMU4C, *Report* is the number of reports produced by each detector and *TP* is the number of bug reports which correctly detect the misuses. As shown in Table V, all of the three detectors miss certain (17-54%) of bugs in APIMU4C comparing to its conceptual recalls, respectively. In addition, all the detectors perform better (both precision and recall) on single-file cases than real-world programs. We investigate the root causes of each detector and discuss them as follow.

Table V shows that Cppcheck and Clang-SA have a quite low recall value (17-54% lower than conceptual recall), but achieves a higher precision value compared to its recall. Particularly, Cppcheck achieves the best precision of 89.95% in single-file cases and 86.36% in real-world project cases. We find that these two tools prefer a conservative strategy, i.e., they only report the bugs with high confidence to improve usage experience [32]. Therefore, API misuses with same root causes in a complex program structure may be ignored.

> ***Finding 10:*** *Static detectors may prefer a conservative strategy to ensure high precision, which may result in a quite low recall value.*

Similar to many tools based on mining techniques, APISan searches the program for explicit constraints in if-statements, function calls or return statements. However, when such constraints hide into the context and require semantic reasoning, APISan fails to learn them. Therefore, APISan fails to detect all IPU cases (23.48% of single-file cases). Moreover, some program properties cannot be explicitly represented in C syntax, such as misuses caused by "cwe590-free of memory not on the heap" in IPU, where it has to reason the context of

TABLE V: Evaluation results of studied static detectors.

APIMU4C			APISan				Cppcheck				Clang-SA			
Case	Type	Bug	Report	TP	Precision	Recall	Report	TP	Precision	Recall	Report	TP	Precision	Recall
Single-File-Case	IPU	510	0	0	0	0	145	127	87.59%	24.90%	127	105	82.68%	20.59%
	IEH	612	446	173	38.79%	28.27%	298	270	90.60%	44.12%	0	0	0	0
	ICC	1050	447	435	97.32%	41.43%	373	337	90.35%	32.10%	746	565	75.74%	53.81%
	Total	2172	893	608	68.09%	27.99%	816	734	**89.95%**	**33.79%**	873	670	76.75%	30.85%
Project-Case	Openssl	50	143	21	14.69%	42%	13	12	92.31%	24%	2	1	50%	2%
	Curl	30	10	0	0	0	5	5	100%	16.67%	0	0	0	0
	Httpd	20	62	6	9.68%	30%	2	2	100%	10%	1	1	100%	5%
	Total	100	215	27	12.56%	**27%**	22	19	**86.36%**	19%	3	2	66.67%	2%

Currently, we exclude reports irrelevant to the misused APIs in real-world projects, for it demands plenty of time to verify reports without priori knowledge.

the pointer to find where it points.[9]

> ***Finding 11:*** *APISan fails to detect improper parameter using misuses (23.48% of single-file cases), which demand to reason implicit context semantics.*

For all of the three detectors, the investigation results show that a large proportion of false positives and false negatives are caused by imprecise semantic analysis. (For the conservative strategy, we fail to count the concrete number.) In details, it consists of two perspectives: (1) **Lack of path-sensitive semantics.** Computing path reachability in static analysis is necessary. For example, it does not require to `free()` resource along the path where `malloc()` fails to allocate heap memory. Otherwise, it will produce a false positive. However, when a program catches an exception in the procedure after the memory allocation, it needs to take each path into consideration instead of matching `malloc/free` invocation counts. Because it demands to free the memory along each error handling path. (2) **Lack of inter-procedural semantics.** When the API usages cross functions (e.g., `malloc()` and `free()` are called in two callers), lack of inter-procedural analysis will produce a false positive of memory leak bug. In addition, APISan relies on static analysis to extract its usage representations. Imprecisions in these analyses may obscure relations between patterns and usages. Therefore, APISan fails to report 51% (78-27) of project cases because of the failure of correctly extracting usage contexts.

> ***Finding 12:*** *Imprecise semantic analysis (e.g., lack of path-sensitive and inter-procedural semantics) will produce a large proportion of false positives and false negatives.*

C. Discussion

Even though developers provide well-format documentation, API-misuse bugs are prevalent in source code (Finding 1). Moreover, part of them may be misused several times by different developers (Finding 6). We find that static detectors are practically capable of detecting many misuses in APIMU4C (Finding 7). However, they perform lower than the expectations as described in publications or tool descriptions (Finding 10-12). We summarize the root causes of low precision and recall of studied detectors in this paper. Therefore, we call researchers to consider the following observations to advance the state-of-the-art in API-misuse detection.

Detection: (1) To design a powerful automatic detector, constraints of parameters, error handling when the API fails and causal function calls need be taken into considerations (Finding 2-4). (2) Tools with mining techniques should go beyond the assumption that a deviation from the most-frequent usage corresponds to a misuse, for sufficient data may be hard to obtain in practice to learn correct usage patterns (Finding 8). (3) Tools with hard-coded rules can provide interfaces for users to provide project-specific usage constraints (Finding 9). (4) Instead of the conservative strategy, we speculate that a good ranking algorithm, retrieving more usages across programs and using probabilistic models may be helpful (Finding 10). (5) Detectors need to capture more semantics (e.g., path-sensitive and inter-procedural) to conduct more precise analyses instead of simply syntactic checking (Finding 11, 12), which are also useful to guide the bug-fix phase (Finding 5).

Maintenance (1) An API misuse tracking system may be useful to recommend and reminder developers while coding, especially with a similar usage context to guide the development (Finding 1). (2) When an API-misuse patch is accepted, we need an automatic searching engine (e.g., employing code clone detection techniques [33]) to detect similar bugs which may conceal in the code base (Finding 6).

[9]https://cwe.mitre.org/data/definitions/590.html

V. Related Work

Researchers conduct many empirical studies to understand API usages from diverse perspectives. Okur et al. [34] analyzed the usage of parallel libraries to direct the future development of these libraries. Nadi et al. [4] find that while developers think it is difficult to use certain cryptographic algorithms correctly, they feel surprisingly confident in selecting the right cryptography concepts. Beyond practice on specific-purpose APIs, other studies cover the knowledge on natural-language statements to make developers aware of usage constraints [35], deprecated APIs after evolution [36], obstacles to learning usages of APIs [37] and API usages to motivate research of specification mining [38]. Complementing the prior studies, we present a comprehensive empirical study on 830 API misuses from six open-source programs of different domains to understand the nature of API misuses in C code.

Over the last 20 years, researchers proposed a multitude of automated bug-detection methods. Particularly, static analysis tools have become an important approach [12], [39]. To compare the detection performance of these detectors, several datasets of software bugs have been crated in the past. BugBench [40] is a bug-detection benchmark of C programs, consisting of 17 buggy programs from open source repositories, w.r.t 13 memory-related bugs, concurrent bugs and semantic bugs. DEFECTS4J [41] consists of 357 source code bugs from 5 real-world open source programs in Java. Amann et al. [2] present MUBENCH, a dataset of 89 real-world misuses to benchmark API-misuse detectors in Java. Our study provides a dataset, named APIMU4C, of API misuses in C code based on our empirical study results of API-misuse bugs. It consists fo both single-file cases covering diverse syntax structures of C and real-world cases. We qualitatively and quantitatively evaluate three static analysis detectors which employ different analysis techniques and strategies on APIMU4C to understand the capabilities and limitations of state-of-the-arts.

VI. Conclusion

API misuse is a well-known source of bugs. Despite the existence of many static detectors over the last two decades, API misuses are still prevalent. In this paper, we present

an in-depth study of 830 API misuses in six popular open-source programs from different domains to understand the nature of API misuse. We create APIMU4C, a dataset of API-misuse bugs in C code, and evaluate three widely-used open-source static analysis detectors on it. Our study reveals many interesting findings. We publish all the original data and evaluation results, and hope our study can inspire more researchers to combat API misuses in practice.

Acknowledgment

We thank the anonymous reviewers for their helpful feedback. This research is sponsored in part by National Natural Science Foundation of China (Grant No. 61802259, 61402248, 61527812), National Science and Technology Major Project of China (Gran No. 2016ZX01038101), and the National Key Research and Development Program of China (Grant No. 2015BAG14B01-02, 2016QY07X1402).

References

[1] U. Dekel and J. D. Herbsleb, "Improving API documentation usability with knowledge pushing," in *ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, 2009, pp. 320–330.

[2] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: a benchmark for api-misuse detectors," in *MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 464–467.

[3] 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 *CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 38–49.

[4] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: why do java developers struggle with cryptography apis?" in *ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 935–946.

[5] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, 2012, pp. 826–836.

[6] C. Xu, X. Sun, B. Li, X. Lu, and H. Guo, "MULAPI: improving API method recommendation with API usage location," *Journal of Systems and Software*, vol. 142, pp. 195–205, 2018.

[7] R. A. B. Jr., "Code reviews enhance software quality," in *Pulling Together, Proceedings of the 19th International Conference on Software Engineering, Boston, Massachusetts, USA, May 17-23, 1997.*, 1997, pp. 570–571.

[8] K. Havelund and G. Rosu, "Runtime verification - 17 years later," in *RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings*, 2018, pp. 3–17.

[9] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, 2016, pp. 470–481.

[10] O. Legunsen, W. U. Hassan, X. Xu, G. Rosu, and D. Marinov, "How good are the specs? a study of the bug-finding effectiveness of existing java API specifications," in *ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 602–613.

[11] 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 *ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 886–896.

[12] M. Egele, D. Brumley, Y. Fratantonio, and C. Kruegel, "An empirical study of cryptographic misuse in android applications," in *CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 73–84.

[13] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, Y. Brun, and N. C. Ebner, "Api blindspots: Why experienced developers write vulnerable code," in *Proceedings of the USENIX Symposium on Usable Privacy and Security (SOUPS)*, Baltimore, MD, USA, August 2018.

[14] N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*, 2005, pp. 580–586.

[15] M. Soni, "Defect prevention: reducing costs and enhancing quality," *iSixSigma. com*, vol. 19, 2006.

[16] D. Zubrow, "Ieee standard classification for software anomalies," *IEEE Computer Society*, 2009.

[17] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini, "A systematic evaluation of static api-misuse detectors," *IEEE Transactions on Software Engineering*, pp. 1–1 (Early Access), 2018.

[18] "Juliet test suite," https://samate.nist.gov/SRD/testsuite.php, 2018.

[19] "Static analysis benchmarks from toyota itc." 2018. [Online]. Available: https://github.com/regehr/itc-benchmarks

[20] "Openssl: a cryptographic library implementing the transport layer security (tls) protocols (including sslv3)." https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1-pre8, 2018.

[21] "Curl: a command line tool and library for transferring data with url syntax." https://github.com/curl/curl, 2018.

[22] "Httpd: a powerful and flexible http/1.1 compliant web server." https://github.com/apache/httpd, 2018.

[23] "Source code of linux kernel v4.18-rc4." https://github.com/torvalds/linux/releases/tag/v4.18-rc4, 2018.

[24] "Openssl: a cryptographic library implementing the transport layer security (tls) protocols (including sslv3)." https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1-pre8, 2018.

[25] "Ffmpeg: a collection of libraries and tools to process multimedia content." https://github.com/FFmpeg/FFmpeg, 2018.

[26] "Freerdp: a free remote desktop protocol library and clients." https://github.com/FreeRDP/FreeRDP, 2018.

[27] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE*, 2013, pp. 672–681.

[28] 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, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 363–378.

[29] "Cppcheck," http://cppcheck.sourceforge.net/, 2018.

[30] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan, "Mining preconditions of apis in large-scale code corpus," in *(FSE-22), Hong Kong, China, November 16 - 22, 2014*, 2014, pp. 166–177.

[31] H. S. Gunawi, C. Rubio-González, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and B. Liblit, "EIO: error handling is occasionally correct," in *FAST 2008, February 26-29, 2008, San Jose, CA, USA*, 2008, pp. 207–222.

[32] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Gros, A. Kamsky, S. McPeak, and D. R. Engler, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[33] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug propagation through code cloning: An empirical study," in *ICSME 2017, Shanghai, China, September 17-22, 2017*, 2017, pp. 227–237.

[34] S. Okur and D. Dig, "How do developers use parallel libraries?" in *SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 54.

[35] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini, "What should developers be aware of? an empirical study on the directives of API documentation," *Empirical Software Engineering*, vol. 17, no. 6, pp. 703–737, 2012.

[36] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation?: the case of a smalltalk ecosystem," in *SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, 2012, p. 56.

[37] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[38] H. Zhong and H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[39] A. Arusoaie, S. Ciobaca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in c/c++ code," in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2017, pp. 161–168.

[40] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench : Benchmarks for evaluating bug detection tools," 2005.

[41] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *ISSTA*, 2014.