IMChecker: API Misuse Checker for C Programs

***Abstract*—Libraries offer reusable functionality through Application Programming Interfaces (APIs) with usage constraints, such as call conditions or orders. Constraint violations, i.e., API misuses, commonly lead to bugs and security issues. Although researchers have developed various API-misuse detectors in the past few decades, recent studies show that API misuse is prevalent in real-world projects. The existing approaches either suffer from the sparse usage problem (i.e., bugs that rarely occur) or report false alarms due to inaccurate semantics. In this paper, we introduce IMChecker to effectively detect API-misuse bugs. The key insight behind IMChecker is a constraint-directed static analysis technique powered by a domain-specific language (DSL) for specifying API usage constraints. Through studying real-world API-misuse bugs, we propose IMSpec DSL, which covers a majority of API usage constraint types and enables simple but precise specification. Furthermore, we design and implement IMChecker to automatically parse IMSpec into checking targets and employ a static analysis engine to identify potential API misuses and prune false positives with rich semantics. We have instantiated IMChecker for C programs and evaluate it on widely used benchmarks and large-scale real-world programs. The results show that IMChecker detects API-misuse bugs with 78.0% to 93.1% precision and approximately 91% recall, outperforming 63.2% and 50% w.r.t. the state-of-the-art. We also found 47 previously unknown bugs in Linux kernel and OpenSSL, 32 of which have been confirmed by the corresponding development communities.**

***Index Terms*—API misuse detection, domain-specific language, static analysis**

I. Introduction

Libraries provide Application Programming Interfaces (APIs) to increase productivity. Correct usage is required to avoid buggy code based on rich constraints, such as restrictions on call orders or call conditions. Violations of these constraints, called *API misuses* [1], may result in bugs and even significant security problems. For example, missing error handling code will cause a denial of service by remote attackers (CVE-2016-2182 [2]), and misuses of SSL/TLS APIs can result in confidential data leakage or man-in-the-middle attacks [3], [4]. Traditional bug detection approaches, such as code review [5] and testing [6], are useful to find a large number of bugs. However, they require considerable manual effort and the preparation of a test environment. These factors limit the detection ability to API-misuse bugs whose usage sites are huge and whose usage patterns are complex, especially for misuses occurring in error handling paths, which do not show up during regular executions [7]. Moreover, because of the complex API usage patterns, the patch to a known issue may result in a new bug. For example, a commit(sha:1c4221) in OpenSSL was created to "fix memory leak in crl2pkcs7 app", but it caused a double free problem, which was fixed in the following commit(sha:d285b5) as "avoid a double-free in crl2pl7".

Many different tools, techniques and methodologies have been proposed to address the above problems. Particularly, Amann et al. [8] propose that approaches with code mining and static analysis techniques have proven to be effective. For example, Ray et al. [7] propose ErrDoc for missing error code checking and Yun et al. [9] present APISan for causal relation and semantic relation on arguments. By leveraging the strength of static analysis (such as control dependency analysis) and code mining (such as frequent sub-itemsets mining algorithm), these tools provide accurate detection and can be applied to scale real-world system programs at an early stage of the development process.

However, recent studies reveal that API misuses remain widespread [10], [11]. Brun et al. [12] present that even in source code written by experienced developers introduce vulnerabilities for misuses of library APIs. According to our empirical investigation of real-world API misuses (see Section II for more details), the main obstacles hindering the existing approaches are:

- Sparse usage problem. Mining techniques leverage the key idea that correct usages appear frequently in large corpora and that deviations are API-misuse bugs with a minimum support predefined to filter out false positives. However, a challenge for such a belief is the "sparse usage problem" [13], where these tools will miss API-misuse bugs under the usage threshold, which is especially severe for program-specific APIs.
- Insufficient semantic analysis. Most methods are built on abstract syntax trees, namely, matching the usage based on syntactic information with less semantic analysis. Moreover, most methods apply an intra-procedural analysis strategy to address the large-scale source code. These approaches generate false positives and false negatives when the usage contains point-to relationship or cross functions.

In this paper, we present ***IMChecker***, an automatic API-misuse checker to augment the current analysis abilities for large-scale C programs [1]. The key insight behind IMChecker is a constraint-directed static analysis technique powered by a domain-specific language for specifying API usage constraints. To understand the true nature of API-misuse bugs in practice, we begin with an empirical study of real-world API-misuse bugs in widely used open-source programs. Leveraging this knowledge, we design a domain-specific language called IMSpec to specify the API-usage constraints. In this way, we specify an explicit API usage pattern to cover sparse usage problems. Then, we design and implement IMChecker, which automatically parses IMSpec into checking targets and detects API-misuse bugs via a static analysis engine employing under-

[1] A temporary repository, including the original dataset, tools and results, can be downloaded from https://github.com/imchecker/IMChecker, where we omitted information which may reveal authors' identities for a double-blind review process.

constrained symbolic execution [14] with traditional static analysis strategies, such as point-to, range and path-sensitive analysis. For potential API misuses, IMChecker prunes false positives by rich semantics and multiple usage instances.

We evaluated our approach on a widely used benchmark, i.e., the Juliet Test Suite [15], and on real-world open-source programs, including Linux-kernel and OpenSSL. The results show that IMChecker can detect API-misuse bugs with 78.0% to 93.1% precision and approximately 91% recall, improvements of up to 63.2% and 50% compared to the state-of-the-art methods. Moreover, IMChecker found 47 previously unknown bugs, of which 32 have been confirmed by the developers and (see Table V).

In summary, our paper makes the following contributions:

- We conduct an empirical study to understand the characteristics of API-misuse bugs in real-world C programs. Leveraging this knowledge, we design a lightweight domain-specific language called IMSpec to specify the API-usage constraints.
- We design and implement IMChecker, which combines static analysis and source code mining to automatically locate API misuses using IMSpec for large-scale C programs.
- We present an extensive evaluation of our approach on a widely known benchmark and two large open-source programs. We find 47 previously unknown bugs, 32 of which have been confirmed by corresponding development communities, and create patches for all of them.

The rest of this paper is organized as follows. Section II provides an overview of our approach, including a motivating example and the workflow. Section III presents our lightweight domain-specific language for API-usage constraints. Section IV describes the framework of IMChecker, followed by an evaluation in Section V. We discuss related work in Section VI and conclude in Section VII.

II. Overview

In this section, we present an overview of our approach for finding API-misuse bugs. First, we introduce a brief background of API misuse and motivate our approach with examples. Then, we outline how our approach addresses the obstacles hindering existing studies.

A. Motivating Example

An API *usage* is a piece of code that uses a given API to accomplish a task. It is a combination of basic program elements, including checking parameter properties, method calls, checking return values, and error handling. The combination of these elements in an API usage is subject to constraints, which depend on the nature of the API. These constraints are called *usage constraints*. When a usage violates one or more such constraints, we call it a *misuse* [1]. Figure 1 shows an example of an API-misuse bug of OpenSSL reported in CVE-2015-0288 [16]. Function *X509_get_pubkey()* attempts to decode the public key for certificate x. If an error occurs, it will return NULL. In function *X509_to_X509_REQ()*, the return value *pktmp* of *X509_get_pubkey()* is used without checking the error code, which results in a NULL Pointer Dereference bug.

```
1  // Location: crypto/x509/x509_req.c: 70
2  X509_REQ *X509_to_X509_REQ(...){
3  ...
4  pktmp = X509_get_pubkey(x);
5  // missing error code check of pktmp
6  +  if (pktmp == NULL)
7  +      goto err;
8  i = X509_REQ_set_pubkey(ret, pktmp);
9  EVP_PKEY_free(pktmp);
10 ...
11 }
12
13 // Location: /crypto/x509/x509_cmp.c: 390
14 int X509_chain_check_suiteb(...){
15 ...
16 // check error value in usage
17 pk = X509_get_pubkey(x);
18 rv = check_suite_b(pk, -1, &tflags);
19 ...
20 }
21 // Location: /crypto/x509/x509_cmp.c: 359
22 static int check_suite_b(EVP_PKEY *pkey,...){
23 ...
24 // ensure pkey not NULL
25 if (pkey && pkey->type == EVP_PKEY_EC)
26 ...// usage of pkey
27 }
```

Fig. 1: Motivating example of an API-misuse bug in OpenSSL reported in CVE-2015-0288 [16], where the missing error code check of *X509_get_pubkey()* resulted in a NULL Pointer Dereference bug.

Tools based on usage-based mining and static analysis techniques, such as Chucky [17], Antminer [18] and APISan [9], can detect this type of bug when there are enough usages of *X509_get_pubkey()*. However, the number of occurrences of *X509_get_pubkey()* is 5, indicating this bug will be ignored by Chucky and Antminer, which require a minimum support greater than 10. Lowering the threshold could significantly increase the number of false positives [19]. Thus, an approach that can address the sparse usage problem to find faults with low support is needed, especially for program-specific APIs.

However, missing error checks do not directly imply an API-misuse bug. The following usage is correct because the return value of *X509_get_pubkey()* is used as the first argument at Line-18, and *check_suite_b()* validates the first parameter against NULL at Line-25. Therefore, APISan will produce a false positive because it performs an intra-procedural analysis that cannot capture semantic cross functions. In addition, context semantic information under subtle constraints should be taken into consideration. For example, the usage constraint at Line-6 states that *EVP_PKEY_free()* must be called only when the result of *X509_get_pubkey()* is not NULL. Therefore, an approach that can provide a more precise analysis with both intra- and inter-procedural context semantics is needed.

B. Our Approach

In this section, we present a brief introduction to our approach using the example without the patch code (Line6-7) in Figure 1. An overview of IMChecker's workflow is presented in Figure 2. IMChecker takes the source code and API usage constraints as input and generates bug reports with concrete locations and reasons as output.

First, API usage constraints are written in a lightweight domain-specific language named IMSpec (§III); for example, *"the return value of X509_get_pubkey() has to be checked with NULL"*. By employing these specifications, IMChecker

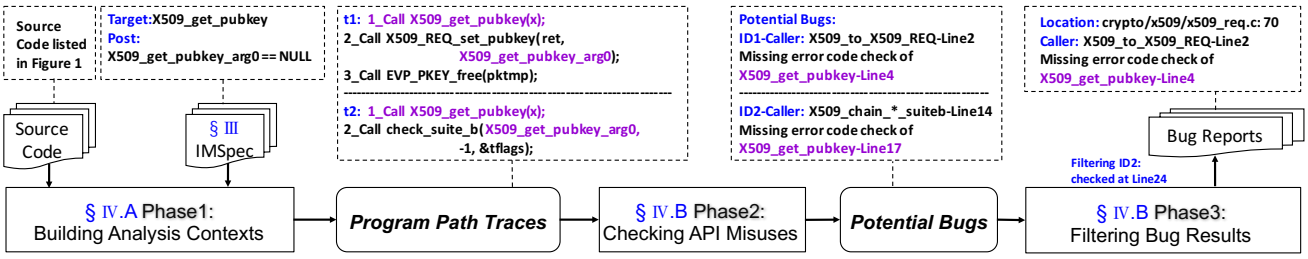

Fig. 2: Overview of IMChecker's workflow.

directly validates the usages of the target API, which relieves the sparse usage problem and guides the bug detection process.

Then, we detect API-misuse bugs in three phases. (1) In **Phase-1**, the analysis context is built by constructing the control flow graph and creating *program path traces* for each target API defined in the specifications by employing under-constrained symbolic execution with point-to, range and path-sensitive analysis. In this example, two traces, $t1$ and $t2$, are generated, as shown in the box above ***Program Path Traces***. In this way, IMChecker can successfully capture the usage context of *X509_get_pubkey()*, *EVP_PKEY_free()* and those in between. (2) In **Phase-2**, IMChecker employs the *traces* to detect violations of the constraints as potential bugs (§IV-B). For example, two API-misuse instances of *X509_get_pubkey()* are found for missing error code checks labeled as *Potential Bugs*. (3) In **Phase-3**, IMChecker improves the detection precision by leveraging multiple usage instances and the semantic information (§IV-C). Then, the second misuse is filtered out for the check conducted in the *X509_to_X509_REQ()* at Line-25. We discuss the details of our approach in the following sections.

III. Design of IMSpec

Behavioral specifications describing API usage constraints have been shown to be useful for developers to effectively utilize APIs as well as to cope with the sparse usage problem by ensuring the validation of usages of the target APIs. For example, Bodden et al. [20] present CRYSL to bridge the cognitive gap between cryptography experts and developers. However, current specification languages are either designed for full program properties, such as BLAST [21], JML [22], or are too specific to be applied to generic API-usage detection [23]. In this section, we introduce a lightweight domain-specific language for API usage constraints named IMSpec. First, to better understand the API-misuse bugs of C code, we conduct a manual study of real-world API-misuse bugs. Then, leveraging this knowledge, we present the design details of IMSpec.

A. Empirical Study of API-misuse Bugs

To better understand what type of API-misuse bugs occur in real projects and how developers fix them in practice, we manually studied two years' version histories of three open-source projects and one-half year of Linux-kernel in 2017, as shown in Table I. These histories are chosen because of the ongoing development and because they are frequently mentioned in diverse bug detection works. In total, we have studied approximately 13.57M LOC and 51.1K commits.

TABLE I: Empirical Study Subjects of API-misuse Bugs

Project	LOC	Studied Period	Total Commits	Bug Fix	API Misuses
Curl	112801	20160101	2613	135	38 (28.1%)
GnuTLS	35782	-	2769	86	30 (34.9%)
OpenSSL	454217	20171231	6487	344	115 (33.4%)
Linux	12.96M	0701-1231	39295	995	326 (32.8%)
Total	13.57M	two years	51.1K	1560	509 (32.6%)

For each project, we begin by extracting all the commits within the study period. Then, we collect bug-fix related commits by matching the commit messages with keywords, such as "bug" and "fix". Next, we further search for API-misuse bug-related commits by approximate string matching with keywords, such as "missing check" and "error handle". For example, we identify an OpenSSL commit (sha: 1ff7425d6) with commit message "Fix a missing NULL check in dsa_builtin_paramgen" as an API-misuse bug-related commit.

We found a total of 509 API-misuse bugs across the four projects. We investigate both the commit messages and patches to understand the nature of API misuses. Based on this study, we identify three generic categories of API-misuse, incorrect parameter use (IPU), incorrect error handling (IEH) and incorrect causal calling (ICC), as shown in Table II. We use the code snippets in Figure 3 to discuss the details of these categories below.

a) Incorrect Parameter Use: APIs simplify programming by abstracting the underlying implementation for a specific target. Certain conditions, called the *preconditions* of the API, must hold whenever an API is invoked. For example, when using a pointer as an argument in a method call, it usually has to be ensured that the pointer is not NULL. However, we find that developers often forget to guarantee this type of constraint. Figure 3a shows an example from Curl. The function *BN_print()* writes the hexadecimal encoding of bn to memory mem. Not validating the parameter against the NULL pointer will cause a NULL Pointer Dereference bug. Moreover, relations on arguments should also be taken into consideration. Typical examples are memory copy APIs, such as *memcpy(d,s,n)*, where the size of the destination buffer **d**

TABLE II: Studied API-misuse Bugs and Patches

Project	IPU	IEH	ICC	Other	Total
Curl	9	11	12	6	38
GnuTLS	3	9	13	5	30
OpenSSL	18	35	36	26	115
Linux	45	148	91	42	326
Summary	75 14.73%	203 39.89%	152 29.86%	79 15.52%	509 100%

```
2016-4-26 Curl:lib/vtls/openssl.c
CommitID: ab691309ce...4de6dfce66
Log: Avoid BN_print a NULL bignum.
   ...
   // null pointer dereference
-  BN_print(mem, bn);
+  if(bn)
+     BN_print(mem, bn);
   push_certinfo(namebuf, num);
   ...
```

(a) Incorrect Parameter Using

```
2017-12-1 Linux:samples/bpf/bpf_load.c
CommitID: 0ec9552b43...e4bd7afc9f
Log: add error checking for perf ioctl ...
   ...
   event_fd[prog_cnt - 1] = efd;
   // miss error status checking and error handling
-  ioctl(efd, PERF_EVENT_IOC_ENABLE, 0);
+  err = ioctl(efd, PERF_EVENT_IOC_ENABLE, 0);
+  if (err < 0) {
+     printf(ERROR_MSG);
+     return -1;
+  }
   return 0;
```

(b) Incorrect Error Handling

```
2016-8-19 GnuTLS:src/srptool.c
CommitID: 39cdaed454...89c649c707
Log: Fix HANDLE_LEAK and memory leak issues.
static int generate_create_conf(*){
   ...
   FILE *fd;
   fd = fopen(tpasswd_conf, "w");
   if (fd == NULL) {
      fprintf(...);
      return -1;
   }
   if (gnutls_srp_base64_encode_alloc(&n, &str_n) < 0) {
      fprintf(stderr, "Could not encode\n");
      // memory leak
+     fclose(fd);
      return -1;
   }
   ...
```

(c) Incorrect Causal Calling

Fig. 3: Examples of API-misuse bugs and fix patches.

should be larger that or equal to the copy length **n**.

b) Incorrect Error Handling: Secure and reliable software must handle all possible failure conditions correctly. Unfortunately, C does not have any error handling primitives. Therefore, certain values are conventionally used to represent execution status, especially for large-scale systems such as Linux [24]. Without correct error handling, a comprehensive failure policy is useless. In Figure 3b, we list a widely used error handling strategy. First, the caller function has to check whether the callee *ioctl* fails. Then, the caller has to propagate the error code upstream using an appropriate return value to notify the rest of the system about the failure. Missing any of these steps will result in error handling misuse. Moreover, in addition to the error handling path, the caller may log an appropriate error message so that the users become aware of the failure.

c) Incorrect Causal Calling: Causal function calls such as *fopen/fclose* are common in API usages. In practice, they are usually subject to context constraints; that is, only when the return value of *fopen* is not NULL should a *fclose* be used to close the opend file handler, as shown in Figure 3c. Previous research, such as PR-Miner [25], focuses only on finding "direct" causal relationships without considering the context constraint between two API calls. Therefore, a false positive is produced in the path where the **fd** is NULL from Line-8 to Line-10. Missing the causal relationship or lack of context constraint validation can both cause an API-misuse bug. Moreover, redundant use of the second function will crash the program, for example, resulting in a double free bug.

d) Others: A total of 79 of 509 (15.52%) studied patches arise due to program-specific issues that cannot be directly applied to generic API usages. These patches include changing function definitions, refactoring styles of error handling code, such as updating logging functions or error messages, and incorrect usages caused by typos.

B. Design of IMSpec

Leveraging the empirical study results, we design a lightweight domain-specific language named IMSpec to present API-usage constraints. IMSpec simultaneously ensures that the target APIs are validated, even with few usages, and guides the misuse detection. An instance of IMSpec is a pattern filled with a set of constraints to correctly use the API, and any violation will result in a API-misuse bug.

The abstract syntax of IMSpec is shown in Figure 4. The specification consists of a set of instances, each of which binds to a target API $target$ with pre-constraints ahead of the $target$ and post-constraints after it. For a causal call, we use $fDef$ to mark the related functions. The pre-constraint is defined after **Pre**, where we design constraints for the parameters, and the post-constraints are defined following **Post**, which is used mostly for error handling and causal function calls.

In Figure 5, we illustrate the API-usage constraints of the bugs in Figure 3 in the form of IMSpec. The pre-constraint is composed of a set of comparison expressions called *assume*. In the expression, we use the symbolic variable $argIndex$, which is a compound of the function name, keyword **arg** and parameter index **n**, to denote the return value or formal parameter of the function. Specifically, zero is used for the return value. For example, *ioctl_arg1* denotes the first formal parameter of *iotcl()*, and *ioctl_arg0* denotes the return value.

The post-constraint includes the context constraint of the return and the following actions under the context. From the empirical study, we learn that the most widely used actions

$$
\begin{aligned}
Specs &:= spec^{+} \\
spec &:= \textbf{Spec:}\ target\ ref^{?}\ pre^{?}\ post^{?} \\
target &:= \textbf{Target:}\ fDef \\
ref &:= \textbf{Ref:}\ fDef^{+} \\
pre &:= \textbf{Pre:}\ (assume)^{+} \\
post &:= \textbf{Post:}\ (eh, action^{*})^{+} \\
eh &:= assume \mid \epsilon \\
action &:= return \mid call \mid endwith \\
return &:= \textbf{RETURN}\ (n) \\
call &:= \textbf{CALL}\ (fName : assume^{+}) \\
endwith &:= \textbf{ENDWITH}\ (fName : assume^{+}) \\
assume &:= opd1\ cmpop\ opd2 \mid \epsilon \\
opd &:= argIndex \mid iop(argIndex) \mid \textbf{NULL} \mid n \\
argIndex &:= fName\_\textbf{arg}n \\
iop &:= \textbf{LEN} \mid \textbf{TYPE} \mid \textbf{MEMTYPE} \mid \textbf{SIZE} \\
cmpop &:= != \mid == \mid >= \mid > \mid <= \mid < \\
fDef &:= fName(type^{*}) \rightarrow type^{?} \\
fName, type &:= id \\
n &\in (natural)\mathbb{N}
\end{aligned}
$$

Fig. 4: Abstract syntax of IMSpec.

```
// IMSpec for BN_print() in Curl
Spec:
Target: BN_print(_, _) -> _
Pre:
 - BN_print_arg1 != NULL,
 - BN_print_arg2 != NULL
// IMSpec for ioctl() in Linux
Spec:
Target: ioctl(_, _, _) -> int
Pre: ioctl_arg1 > 0
Post:
 - ioctl_arg0 < 0, CALL(printf:_), RETURN(-1)
// IMSpec for fopen() in GnuTLS
Spec:
Target: fopen(_, _) -> _
Ref: fclose(_) -> _
Post:
 - fopen_arg0 == NULL, CALL(printf:_), RETURN(-1);
 - fopen_arg0 != NULL, CALL(fclose: fopen_arg0 ==
     fclose_arg1)
```

Fig. 5: IMSpec for the API-usage constraints of the bugs shown in Figure 3.

are **CALL** actions, indicating a function call, and **RETURN** actions, indicating that the caller of the target function has to return a specific value under this context. For example, the third *Spec* instance describes the usage constraints for *fopen()*. The first action after invoking *fopen()* is to check the return value status, where a NULL pointer will be returned for a failure. Then, if the check fails, the caller of *fopen()* has to output an error message by *fprintf()* and propagate the error code to the rest of the program by returning -1. However, if *fopen()* succeeds, the caller has to call function *fclose()* to close the same file descriptor generated by this *fopen()*, which is depicted in constraint "Call(fclose: fopen_arg0 == fclose_arg1)". *We use "_" as a placeholder to represent actions or values that are not important.*

During the empirical study, as well as discussions with developers, we also find implicit constraints [13] that are not explicitly written in the code with if-statements. For example, the function *free()* deallocates memory previously allocated by a call to *calloc()*, *malloc()*, or *realloc()*. Therefore, the parameter of *free()* should be on heap. Otherwise, a bug classified as "Free of Memory not on the Heap" [26] will occur. To support more semantic API-usage patterns, we design an additional action *endwith* for no more than twice usage of a specific function call, which results in "Use After Free" [27]. In addition, four inner operators are created to capture more semantics: **LEN** for the length of a variable, **TYPE** for the type of a variable, **MEMTYPE** for the memory and **SIZE** for the size of the memory pointed to by a pointer.

IV. Design of IMChecker

A correct API-usage has to satisfy a set of usage constraints, that is, violations of the constraints may result in API-misuse bugs. IMChecker automatically detects these bugs in source code using the specifications of IMSpec. To process complex, real-world programs, IMChecker's underlying mechanisms must be scalable while sacrificing the minimal amount of accuracy. In this section, we elaborate the design details of IMChecker, including under-constrained symbolic execution with static analysis techniques to build analysis context (§IV-A), methodologies to detect API-misuse bugs in the analysis context (§IV-B) and a method to filter false positives using semantic information and usage statistics (§IV-C).

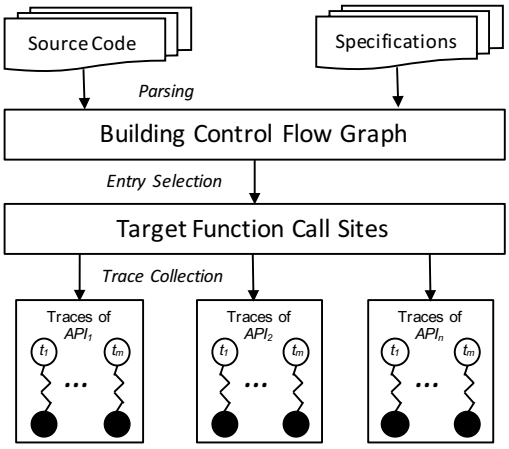

Fig. 6: Workflow of building analysis contexts.

A. Building Analysis Contexts

IMChecker performs symbolic execution to generate program path traces that capture rich semantic information for each target API defined in the specifications. In Figure 6, we illustrate the workflow for building analysis context, which consists of three steps. First, IMChecker parses the source code and specifications $Specs$ to build a control flow graph to conduct a flow- and path-sensitive analysis. Then, for each target API f in the specifications, we select analysis entries as target function call sites by labeling the callers C, which invokes f. Next, for each caller $caller$ in C, symbolic execution with point-to and range analysis is employed to generate a series of program path traces T with semantic information of the usage of f.

In Figure 7, we formally describe the structure of the program path traces computed by IMChecker. Each trace t consists of a sequence of actions a^+ with a value map VM recording the semantic information from a symbolic variable sv to a concrete value cv. A symbolic variable is defined by an action ID and the index n. For example, $ID\_f\_arg\_i$ denotes the i^{th} parameter of f called in the ID^{th} action. We use f_arg_0 to represent the return value of f and arg_0 for the symbolic variable returned by the $caller$ of f used in the action **Return**. Currently, the concrete value cv can be an integer range $range$ computed by range analysis, a pointer in form of Accesspath [28] ap or $NULL$.

We illustrate the two traces of the code in Figure 3c without the fix-related statements in Figure 8. Both traces start from the action calling *fopen()*. Then, the first trace assumes that the return value labeled as 1_fopen_arg_0 is NULL, indicating

$$
\begin{array}{rcl}
\text{(Traces) } T & := & t^+ \\
\text{(trace) } t & := & (ID\_a)^+; VM \\
\text{(action) } a & := & \textbf{Assume}(exp) \mid \textbf{Call } f(sv^+) \mid \textbf{Return}(sv) \\
\text{(expression) } exp & := & sv1\ cmpop\ sv2 \\
\text{(value map) } VM & : & sv \rightarrow cv \\
\text{(symbolic variable) } sv & := & (ID\_f\_\textbf{arg}\_n) \\
\text{(operator) } cmpop & := & != \mid == \mid >= \mid > \mid <= \mid < \\
\text{(concrete value) } cv & := & range \mid ap \mid \text{NULL} \\
\text{(function) } f & \in & \mathbb{F} \\
\text{(index) } ID,\ n & \in & (natural)\mathbb{N}
\end{array}
$$

Fig. 7: Abstract syntax of program path traces.

```
t1 : 1_Call  fopen((1_fopen_arg1), (1_fopen_arg2));
     2_Assume(1_fopen_arg_0 == NULL) ;
     3_Call  fprintf(...);
     4_Return (-1);
t2 : 1_Call  fopen((1_fopen_arg1), (1_fopen_arg2));
     5_Assume(1_fopen_arg_0 != NULL) ;
     6_Call  gnu*alloc(...);
     7_Assume(6_gnu*alloc_arg_0 < 0) ;
     8_Call  fprintf(...);
     9_Return (-1);
```

Fig. 8: Program path traces of the code in Figure 3c.

the failure of *fopen()*, and records the error handling actions of printing the error message and propagating the error code -1. By contrast, the second trace continues because of the success of *fopen()*. Similarly, the second trace checks the return value of *gnu*alloc()* and performs error handling actions. However, the second trace fails to close the opened file descriptor 1_fopen_arg_0, resulting in a memory leak bug (i.e., an incorrect causal call bug).

Similar to the traditional analysis, the key challenge of building such path traces in large and complex programs is to overcome the path-explosion problem. We make two design decisions to achieve scalability without sacrificing substantial accuracy. (1) Limiting inter-procedural analysis. From the empirical study result, we find that most of API-misuse bugs occurred in a single caller function. Therefore, IMChecker performs symbolic execution intra-procedurally for each $site$ to track the semantic context of the target API f and refines the bug detection results by a filtering phase with the inter-procedural semantics presented in Section IV-C. (2) Unrolling loops. IMChecker unrolls each loop only once to reduce the number of paths explored. While this restriction can limit the accuracy of the semantic computation, it does not noticeably affect the accuracy of API-misuse bug detection for only a small number of API usage constraints related to loop variables.

B. Checking API misuses

In the checking phase, IMChecker employs the constraints defined in specifications $Specs$ and the program path traces T to detect API-misuse bugs. We illustrate our detecting algorithm in Algorithm 1. First, we extract all the target functions into $APISet$. For each API f, we detect API-misuse bugs along the traces T' that invoke f using the specification $spec$. Then, for each trace t in T', we validate whether the constraints in $spec$ are satisfied along t. If a path fails, IMChecker labels the call site of f along this t as a API-misuse bug.

To check the satisfaction of the $spec$ along t, we conduct the following steps. First, we label the action a that calls f. Then, we check the pre-constraints pre in the actions ahead of a and the post-constraints $post$ after a. For the $assume$ constraint, we try to match it with an **Assume** action whose operator and operands are all equivalent. Specifically, we have to handle semantically equivalent expressions, such as $a == b$ and $b == a$, and the implications between expressions such as $a == -1$ satisfies $a < 0$. For the $call$ and $endwith$ constraints, we match with a **Call** action, where the callee f'

Algorithm 1 Algorithm for checking API-misuse bugs

Input: program path traces T, specifications $Specs$
Output: bug report R

```
1: R ⇐ ∅
2: APISet ⇐ extractTargetAPISet(Specs)
3: for each API f ∈ APISet do
4:     spec ⇐ extractSpec(Specs)
5:     T' ⇐ extractPathTraces(spec, T)
6:     for each trace t ∈ T' do
7:         result ⇐ satisfy(t, spec)
8:         if (!result) then
9:             R ⇐ addBug(t, spec)
10:        end if
11:    end for
12: end for
13: return R
```

and relationship are satisfied. To check the **endwith** constraint, we check whether more than one action matches. For the $return$ constraint, we match with the **Return** action, which contains the same return value. If any of the constraints fail to match, IMChecker reports a bug.

For example, the second specification defined in Figure 5 specifies that if the return value ret of *fopen()* is not NULL, *fclose()* has to be called with ret used as the first formal parameter. In Figure 8, the second trace satisfies the first $assume$ that ret is not NULL. However, it fails to call the *fclose()* to close ret. Therefore, IMChecker produces a bug report as: "src/srptool.c, Caller:generate_create_conf-Line4; missing CALL(fclose) after fopen at Line-7."

C. Filtering Bug Reports

To achieve the scalability required to support real-world programs, we employ a limiting inter-procedural strategy to address the path-explosion problem. The strategy generates false positives when a usage cross functions. However, developers dislike using tools with low precision [29]. Therefore, we apply inter-procedural semantics and usage statistics to reduce false positives.

First, we conduct semantic-based filtering. We attempt to infer the missing assumes in the pre-constraints pre of f by the semantic information at call action a, such as the memory location of the parameter of *free()*, which is difficult to check in an explicit if-statement in C, and the buffer size of pointers of *memcpy()*. For the missing error status check assume eh in post-constraints $post$, we further check the function call sites used in the return value, such as the second case in Figure 1. For causal call patterns such as f-con-g, if the return value ret of f is directly assigned to the parameter of the caller c of f or returned by c, we filter it out. If f is called and con is satisfied but the function call of g is missing, we filter by checking whether g is called in the function bar, where ret of f is an actual argument of bar.

Then, we conduct usage-based filtering. If the rate of misuses to all usages is larger than the minimum confidence α, we ignore all the bugs in this file because the file has high potential for testing purpose, as mentioned in OpenSSL [30].

If more than β of the usages of f are misuses, we ignore the bugs because the specification may be buggy or the developers may intend to ignore the constraints due to a lack of approaches to recover from a failure [31]. In this paper, we set α and β to 1/3 based on the empirical study results.

D. Implementation

IMChecker is built in Java language. We preprocess the source code into LLVM-IR 3.9 [32], which provides a typed, static single assignment (SSA) and well-suited low-level language. Then, we parse the LLVM-IR by javacpp [33] and construct an extended control flow graph, which classifies the edges into control edges for semantic computation and summary edges to provide a mechanism to support large-scale programs. To this end, we can conduct a path-sensitive and context-sensitive analysis. We implement a point-to analysis and range analysis based on abstracted AccessPath proposed by Bodden et al. [28] to collect rich semantic information. In addition, our specification language IMSpec is written in the human-readable data serialization language Yaml [34].

V. Evaluation

In this section, we evaluate our approach to detect API-misuse bugs. Specifically, we are interested in IMChecker's detection performance in real-world programs. We attempt to answer the following research questions:

- **RQ1:** How effective is IMSpec in relieving the sparse usage problem?
- **RQ2:** How effective is IMChecker in detecting API-misuse bugs?

A. Experimental Setup

We select a widely used benchmark, i.e., Juliet Test Suite V1.3, and two real-world programs in their latest versions: Linux kernel-4.18-rc4 released on 2018-7-9 [35] and OpenSSL-1-1-1-pre8 released on 2018-6-20 [36]. We evaluate our approach from two perspectives:

- Controlled dataset. We employ 2010 API-misuse bug instances from the Juliet Test Suite and 50 randomly selected evolutionary bugs and patches from API-misuse bug-fix commits that we identified in the empirical study of OpenSSL (see Section III-A). These cases are used for comparison with state-of-the-art methods in terms of false positives and false negatives.
- New bugs in real-world programs. We also apply IMChecker to the latest versions of two real-world programs to evaluate whether our approach can find new bugs.

To evaluate the ability of IMSpec to address the sparse usage problem, we manually count the usage statistics of the APIs from the subjects of the empirical studies. We measure IMChecker's ability to detect bugs in terms of precision, recall and F-Score, which are defined as follow:

$$\mathrm{P} = \frac{|\text{reported bugs} \cap \text{ground truth}|}{|\text{reported bugs}|} \tag{1}$$

$$\mathrm{R} = \frac{|\text{reported bugs} \cap \text{ground truth}|}{|\text{ground truth}|} \tag{2}$$

$$\text{F-Score} = \frac{2 \times P \times R}{P + R} \tag{3}$$

We compare with APISan [9], a state-of-the-art API usages detection tool through semantic cross-checking supporting multiple types of API-misuse bugs (Others are either designed to a specific type of misuses or unavailable). In our experiments, both tools ran on Ubuntu 16.04 LTS (64-bit) with a Core i5-4590@3.30GHz Intel processor and 16GB memory. We set time overheads as 6 hours at most to detect bugs beyond preprocessing source code. In the following, we present the detailed results.

TABLE III: Usage Statistics of Misused APIs

Number of usages	Number of unique APIs	Proportion
< 5	19	22.62%
5 - 9	13	15.48%
10 - 14	6	7.14%
≥ 15	46	54.76%
Total	84	100%

B. Results

In this part, we first evaluate the effectiveness of IMSpec in relieving the sparse usage problem. Then, we evaluate the ability of IMChecker in detecting API-misuse bugs. Finally, we share the lessons learned from applying IMChecker to real-world programs.

RQ1: How effective is IMSpec in relieving the sparse usage problems?

API-misuse detectors with code mining techniques follow a macroscopic strategy to identify common usages by a global scan of the code repository. However, this strategy typically limits the reported results to the most frequently occurring operations, that is, the usage number of an API has to exceed the predefined minimum support value. For example, APISan uses 0.8 to indicate that an API has been called at least five times [9], Antminer employs 10 [18] and PR-Miner requires at least 15 usages [18]. However, ignoring APIs under the threshold will result in the sparse usage problem, which we address by applying IMSpec. To evaluate the ability of IMSpec to handle the sparse usage problem, we randomly collected 200 API-misuse-related patches from subjects of the empirical studies and counted the unique API usage statistics.

Table III summarizes the results. Nineteen of these APIs are called fewer than five times, indicating APISan will miss 22.62% of the bugs. Similarly, Antimer will miss 38.10 (22.62+15.45)% and PR-Miner will miss 45.24%. Twelve of the nineteen APIs with fewer than 5 usages are program-specific. Therefore, seven misuses may be found when detecting cross-program in a large-scale code corpus, but the other twelve API-misuse bugs will still be missed. Moreover, the misuses are difficult to separate from correct uses with a small number of call sites based on usage statistics. For example, f is used in two sites, and the correctness of the usages cannot be determined without prior knowledge. By contrast, IMSpec can be extensively applied to any API to ensure that the usages of a target API are valid. In addition, we create IMSpec for all the APIs in a well-defined format shown in Figure 5. In total, there are 476 lines with an average of 5.67 lines for each.

For this dataset, the sparse usage problem is not a corner-case problem, and IMSpec is a user-friendly way to relieve this problem. IMSpec aims to complement, rather than replace, the current API-misuse detection abilities. We also employ the written IMSpec to detect new bugs in real-world projects.

TABLE IV: Bug Detection Results

	Juliet Test Suite cases			Evolutionary cases		
	Total bugs = 2010			Total bugs = 50		
Tool	APISan	IMC–	IMC	APISan	IMC–	IMC
Detected	781	2044	1966	142	150	59
False Positive	80	214	136	121	102	13
False Negative	1249	180	180	29	2	4
Precision	0.898	0.895	0.931	0.148	0.320	0.780
Recall	0.349	0.910	0.910	0.420	0.906	0.920
F-Score	0.502	0.903	0.920	0.219	0.480	0.844

Evaluation results on the controlled dataset, where IMC is short for IMChecker and IMC– is IMChecker without the filtering phase.

The results show that we can successfully find 47 previously unknown bugs (See Table V).

RQ2: How effective is IMChecker in detecting API-misuse bugs?

Controlled Dataset Results. We evaluate IMChecker on a controlled dataset to compare the accuracy with that of the state-of-the-art API-misuse detection tool APISan, as described in Section V. We summarize the controlled dataset results in Table IV, where IMC is short for IMChecker and IMC– is IMChecker without the filtering phase, which will be discussed in *Effectiveness of the filtering phase*.

In the Juliet Test Suite cases, IMChecker reports 1966 bugs, of which 136 are false positives, and misses 180 API misuses. Thus, IMChecker's overall precision and recall are 93.1% and 91.0%, improvements of 3.3% and 56.1% compared to those of APISan. In the evolutionary cases from real-world API-misuse patches, IMChecker successfully finds 92% of the ground truth with a false-positive rate of 22.0%(1-78.0%), whereas the APISan detects 42% with a high false-positive rate of 85.2%. The evaluation results show that both methods perform better in the Juliet Test Suite cases, which are simple examples used to explain the core behavior of API misuses with dozens of lines. However, when applied to real-world programs with large-scale codes and complex semantics, the accuracies decrease to some extent.

In summary, IMChecker detects 56.1% and 50% more bugs than does APISan on the Juliet cases and evolutionary cases, respectively. One cause of the improvement in recall is due to IMSpec being used for the sparse usage problem discussed above. Another improvement is due to the rich semantic information captured. However, APISan uses function calls with variable names as an identifier to match usages without considering point-to relationships. Therefore, the accuracy of APISan suffers when a storage location can be accessed in more than one way, which is called alias analysis [37].

Moreover, the lack of precise semantic information will bring irrelevant statements under consideration, which will severely impact the bug detection algorithm of APISan. These limitations have also been noted by Jason2031 [38]. IMChecker can successfully address these limitations, as discussed in Section IV-A.

```
1  // OpenSSL:crypto/bn/bn_gf2m.c
2  int BN_GF2m_mod_mul_arr(*){ // Line-410
3      ...
4      BN_CTX_start(ctx);
5      // check the error code
6      if ((s = BN_CTX_get(ctx)) == NULL)
7          goto err;
8      ...
9  }
10 static int BN_GF2m_mod_inv_vartime(*){// Line-552
11     ...
12     BN_CTX_start(ctx);
13     b = BN_CTX_get(ctx); // fasle positive
14     c = BN_CTX_get(ctx); // fasle positive
15     u = BN_CTX_get(ctx); // fasle positive
16     // sufficient to check the last usage
17     v = BN_CTX_get(ctx);
18     if (v == NULL)
19         goto err;
20     ...
21 }
```

Fig. 9: False Positives Filtered by IMChecker.

Effectiveness of the filtering phase. We note that one of the major improvement in the accuracy of IMChecker compared to that of APISan is the moderate false-positive rate (22.0% compared to (85.2%), which is augmented by the filtering phase, as described in Section IV-C. To investigate the effectiveness of the filtering phase, we evaluate IMChecker without this phase as IMC–. As shown in Table IV, the filtering phase eliminates 78 false positives, improving the precision by 3.6% in the Juliet cases, and 89 false positives in the evolutionary cases, improving the precision by 46.0 %.

In particular, 74 false-positives are eliminated for *BN_CTX_get()* in OpenSSL, as shown in Figure 9. The method is used to obtain temporary BIGNUM variables from a BN_CTX and returns NULL in the occurrence of an error. The manual specification of OpenSSL mentions that "A function must call BN_CTX_start() first. Then, BN_CTX_get() may be called repeatedly. It is sufficient to check the return value of the last BN_CTX_get() call." [39]. Therefore, all

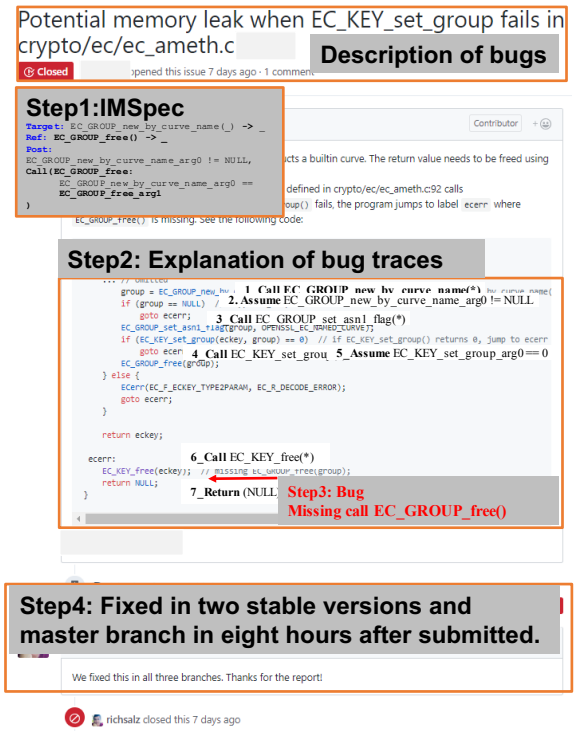

Fig. 10: Screenshot of a memory leak bug found by IMChecker, which is fixed at eight hours on the two stable versions and mainline branch.

TABLE V: Previously Unknown Bugs Comfirmed by Developers

Index	Program	Bug ID	Misuse API	Location (FileName: caller)	Category	Status
1	Linux-4.18-rc4	*	kzalloc	arch/x86/platform/uv/tlb_uv.c: **init_per_cpu**	ICC	✓
2		*	alloc_disk	drivers/block/pktcdvd.c: **pkt_setup_dev**	IEH	✓✓
3		*	kzalloc	drivers/clk/pxa/clk-pxa.c: **clk_pxa_cken_init**	ICC	✓
4		*	devm_clk_get	drivers/bluetooth/hci_bcm.c: **bcm_get_resources**	IPU	✓
5		*	kzalloc	arch/mips/sgi-ip22/ip22-gio.c: **ip22_check_gio**	ICC	✓
6		*	nla_nest_start	net/ncsi/ncsi-netlink.c: **ncsi_pkg_info_all_nl**	IPU	✓✓
7		*	nla_nest_start	net/openvswitch/conntrack.c: **ovs_ct_limit_cmd_get**	IPU	✓✓
8		*	nla_nest_start	net/openvswitch/datapath.c: **queue_userspace_packet**	IPU	✓✓
9		*	alloc_skb	drivers/crypto/chelsio/chtls/chtls_cm.c: **chtls_close_conn**	ICC	✓
10		*	__send_and_alloc_skb	drivers/net/team/team.c: **team_nl_send_options_get**	ICC	✓
11		*	devm_kzalloc	drivers/gpio/gpio-tegra.c: **tegra_gpio_probe**	IEH	✓
12		*	devm_kzalloc	sound/soc/sh/rcar/core.c: **rsnd_probe**	IEH	✓
13		*	pci_find_ext_capability	drivers/net/ethernet/cavium/thunder/nic_main.c: **pci_find_ext_capability**	IPU	✓✓
14		*	pci_find_ext_capability	drivers/pci/pcie/dpc.c: **dpc_probe**	IPU	✓✓
15		*	devm_regmap_init_i2c	drivers/iio/magnetometer/hmc5843_i2c.c: **hmc5843_i2c_probe**	IPU	✓
16		*	devm_ioremap	drivers/ata/pata_pxa.c: **pxa_ata_probe**	IEH	✓
17		*	alloc_workqueue	drivers/net/ethernet/intel/fm10k/fm10k_main.c: **fm10k_init_module**	ICC	✓
18		*	ida_pre_get	fs/namespace.c: **mnt_alloc_id**	IPU	✓✓
19		*	wm831x_reg_read	drivers/clk/clk-wm831x.c: **wm831x_fll_is_prepared**	IEH	✓
20		*	dma_mapping_error	drivers/infiniband/hw/qib/qib_sdma.c: **qib_sdma_verbs_send**	IEH	✓
21		*	get_zeroed_page	arch/s390/kernel/sysinfo.c: **sysinfo_show**	IEH	✓
22		*	kzalloc	arch/arm/mach-mvebu/board-v7.c: **i2c_quirk**	ICC	✓
23		*	kzalloc	arch/arm/mach-mvebu/coherency.c: **armada_375_380_coherency_init**	ICC	✓
24	OpenSSL-1.1.1-pre8	*	ASN1_INTEGER_get	crypto/asn1/tasn_utl.c: **asn1_do_adb**	IPU	✓
25		*	ASN1_INTEGER_set	crypto/pkcs12/p12_init.c: **PKCS12_init**	IEH	✓✓
26		*	OBJ_nid2obj	crypto/asn1/asn_moid.c: **do_create**	IEH	✓✓
27		*	BN_set_word	ssl/t1_lib.c: **ssl_get_auto_dh**	IEH	✓✓
28		*	EVP_PKEY_get0_DH	ssl/statem/statem_srvr.c: **tls_process_cke_dhe**	IPU	✓✓
29		*	EC_GROUP_new_by_curve_name	crypto/ec/ec_ameth.c: **eckey_type2param**	ICC	✓✓
30		*	ASN1_INTEGER_set	crypto/x509v3/v3_tlsf.c: **v2i_TLS_FEATURE**	IEH	✓✓
31		*	ASN1_INTEGER_to_BN	crypto/ts/ts_lib.c: **TS_ASN1_INTEGER_print_bio**	IPU	✓✓
32		*	BN_sub	crypto/rsa/rsa_ossl.c: **rsa_ossl_private_encrypt**	IPU	✓✓

We are creating issues and patches for all of 47 previously unknown bugs detected by IMChecker (listed on our repository). Now, 32 have been already confirmed by corresponding developersas listed above, where ✓is confirmed and ✓✓is already applied to mainline branch. ***We omit Bug ID (Bugzilla ID for Linux and Issue ID for OpenSSL) for double-blind reviewing.***

the usages in Figure 9 are correct. We filtered these false positives by usage-based strategy because most of the usages of *BN_CTX_get()* violate the constraints. However, APISan generated 86 false positives for this function. Unfortunately, the filtering phase produced two additional false negatives, which is acceptable relative to the total number of false positives. Other false positives are filtered by a semantics-based strategy for the usages crossing functions, as described in Section IV-C.

New Bugs. The main motivation of IMChecker is to detect API-misuse bugs in real-world programs, namely, to determine whether IMChecker can find previously unknown bugs. Therefore, we apply IMChecker to the latest versions of two well-known open-source programs: Linux kernel-4.18-rc4 and OpenSSL-1-1-1-pre8. Target APIs are selected from the misused ones from the empirical study. In total, IMChecker detected 47 previously unknown bugs. We are currently attempting to create issues and patches for all the bugs and send them to the developers of each program. Up to now, 32 of the new bugs have been confirmed by the developers as listed in Table V. For example, in Figure 10 we present a memory leak bug found in OpenSSL, which was fixed in two stable versions and the master branch within eight hours after we submitted the issue with a bug description, explanation of the bug traces and potential fix strategy. Function *EC_GROUP_new_by_curve_name()* constructs a built-in curve, and the return value needs to be freed using *EC_GROUP_free()*. However, if the result of EC_KEY_set_group is 0 (Actions 4 and 5), the function will proceed to the error label without calling *EC_GROUP_free()*, resulting in a memory leak problem.

C. Discussion

While investigating the bug reports generated by IMChecker, we find several interesting bugs and gain useful experience in the bug reporting process with open-source developers. We share our following experience. **(1) API misuses are not corner cases.** In total, we find 47 previously unknown bugs in APIs that have been previously misused in the same programs. These bugs may result from the lack of a bug information sharing mechanism among developers and the lack of API usage specifications. We believe that bug fixing is an essential activity during the entire life cycle of software development. Automatic bug finding tools, such as IMChecker, with large-scale analysis capability can be integrated into the development cycle. The IMSpec used in IMChecker can be customized to incrementally address misuses. **(2) Accelerating manual auditing.** API misuses usually have similar behavior patterns. For example, many types of vulnerabilities result from insufficient validation of input or missing error code checks. However, discovering all the missing checks by human is tedious and time consuming. Automatic tools can efficiently accelerate the manual auditing with differences extracted as good usages and bad usages. For example, two of the API misuses were fixed within 8 hours after we created the issues with possible fixing patches, as shown in Figure 10. **(3) Intentional choices.** We also find that many misuses are not mistakes but intentional choices. Many error code checks of return values are ignored by developers. During the bug reporting process with the OpenSSL developers, we learned that they intentionally ignore some error code checks for performance considerations or due to the lack of an error handling mechanism in C.

D. Threats to Validity

Some inaccuracy may arise in IMChecker due to the specific features of LLVM because IMChecker parses C code into LLVM-IR to conduct a precise semantic analysis. However, our evaluation shows that these cases are rare. Inaccuracy may also arise due to the imprecision of the API usage constraints we summarize from documentation on the official websites and patches. We refine our specifications incrementally by reporting new bugs identified by IMChecker. Another reason for inaccuracy is the limited inter-procedural analysis. Currently, we are employing the inter-procedural context semantics and usage statistics to improve the analysis results.

We investigate the API-misuse bugs for a given time period based on approximate keyword matching. Therefore, not all types of API-misuse bugs may be covered in the studied programs, and developers may not put the keywords into the commit messages. We evaluate IMChecker on four programs and one benchmark. Thus, our results may not generalize. To minimize this threat, we evaluate the results by creating an evolutionary dataset and by applying IMChecker to the latest versions of these programs.

VI. Related Work

Incorrect usage of APIs is a prevalent cause of software bugs, crashes, and vulnerabilities. Particularly, Nadi et al. [40] propose that it is severe for APIs with complex usage patterns, such as cryptography APIs. Many attempts have been made to address the problem of API misuse, such as testing [6], static analysis [8] and runtime verification [41]. Static source code analysis has prevailed as one of the most promising techniques since it requires access only to the source code, which is typically available at an early stage of the developing process. In this section, we survey related work on static analysis for API misuse detection in C.

Checking via the usage-based mining technique. The usage-based specification mining and detecting technique has shown great promise in solving API misuses through a data-driven approach. The usages of APIs in a large corpora are leveraged to understand the rules of the APIs and to infer usage constraints. For example, PR-Miner is an API-misuse detector for C [25] that encodes usages as the set of all function names called within the same function and then employs frequent-itemset mining to find patterns with a minimum support of 15 usages. Violations are strict subsets of a pattern that occur at least ten times less frequently than the pattern. Chucky [17] is proposed to automatically detect missing checks in software by statically tainting source code and identifying anomalous or missing conditions linked to security-critical objects. To increase the detection precision, APISan [9] automatically infers correct API usages from the source code by semantic cross-checking in the form of relaxed symbolic execution, which limits path exploration to a bounded number of intra-procedural paths. AntMiner [18] applies a divide-and-conquer method to reduce false positives by carefully preprocessing the source code. AntMiner employs a slicing technique to decompose the whole program into independent sub-repositories according to a set of critical operations and carefully normalizes statements into canonical forms as far as possible.

However, a challenge for such a technique is the sparse usage problem, where insufficient usages are ignored. For example, Saha et al. [19] report that for Linux, only 3% of the protocols have both support of 15 or more, so PR-Miner will fail to find these bugs. Unfortunately, lowering the threshold can significantly increase the number of false positives. IMChecker complements these tools by employing a lightweight domain-specific language, IMSpec, to specify the target APIs and the constraints. Furthermore, IMChecker can leverage the results of mining techniques listed in a survey of automated API-property inference techniques by Bodden et al. [42].

Checking by specifications. The behavioral specifications (pre- and post-conditions) of APIs could help developers to effectively utilize the APIs and check the correctness of usages. Static Driver Verifier (SDV) [43] is a toolset developed by Microsoft Corporation to check correct usages of Microsoft Windows kernel module APIs by means of a specification language named "SLIC" [23]. SDV automatically obtains information about source files and uses static verification techniques to detect violations of API usages. Sparse [44] is a static analysis tool that uses developers' annotations to find certain types of bugs, such as lock/unlock. SSLINT [4] is a scalable, automated, static analysis system for detecting incorrect use of SSL/TLS APIs. SSLINT is capable of performing automatic logic verification based on predefined rules with high efficiency and good accuracy. Jana et al. [45] proposed a method called EPEx to detect error-handling bugs by introducing some error specifications. EPEx explores error paths and uses under-constrained symbolic execution to decide whether the error is correctly handled (e.g., logging error messages or propagating the error value upstream). Ray et al. [7] extended EPEx to support resource lease bugs.

In contrast to SDV, which is intended only for static verification of kernel modules for the Microsoft Windows OS whose kernel API is stable, IMChecker employs a generic API-usage specification language designed from real-world misuse bug instances. Moreover, IMChecker can support multiple types of API-misuse bugs, including resource bugs by Sparse and Tian, SSL/TLS APIs by SSLINT, and error-handling bugs by EPEx. Furthermore, IMChecker's specification language is a separate component from the checker and can easily be extended and written by developers, even without the source code.

VII. Conclusion

Modern APIs are rapidly evolving and error-prone. Incorrect usages of APIs will produce API misuse bugs, which can cause serious security vulnerabilities. In this paper, we propose IMSpec, a lightweight domain-specific language, to specify the API usage constraints and IMChecker, an automatic API-misuse checker for large-scale C programs. We evaluated our approach on 13.57M lines of code. Our results show that our methods perform better than the current state-of-the-art techniques. We also find 47 previously unknown bugs, of which 32 have already been confirmed by the developers. In the future, we will conduct experiments on more programs and investigate heuristics to automatically rank reported bugs based on potential severity.

REFERENCES

[1] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "Mubench: a benchmark for api-misuse detectors," in *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 464–467. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903506

[2] "Cve-2016-2182," http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2182, 2016.

[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 *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, 2012, pp. 38–49. [Online]. Available: http://doi.acm.org/10.1145/2382196.2382204

[4] B. He, V. Rastogi, Y. Cao, Y. Chen, V. N. Venkatakrishnan, R. Yang, and Z. Zhang, "Vetting SSL usage in applications with SSLINT," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 519–534. [Online]. Available: https://doi.org/10.1109/SP.2015.38

[5] 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. [Online]. Available: http://doi.acm.org/10.1145/253228.253461

[6] M. Wang, J. Liang, Y. Chen, Y. Jiang, X. Jiao, H. Liu, X. Zhao, and J. Sun, "SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 61–64. [Online]. Available: http://doi.acm.org/10.1145/3183440.3183494

[7] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in C," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, 2017, pp. 752–762. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106300

[8] 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.

[9] 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. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/yun

[10] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky, "You get where you're looking for: The impact of information sources on code security," in *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, 2016, pp. 289–305. [Online]. Available: https://doi.org/10.1109/SP.2016.25

[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 *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, 2018, pp. 886–896. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180260

[12] 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.

[13] S. K. Samantha, H. A. Nguyen, T. N. Nguyen, and H. Rajan, "Exploiting implicit beliefs to resolve sparse usage problem in usage-based specification mining," *PACMPL*, vol. 1, no. OOPSLA, pp. 83:1–83:29, 2017. [Online]. Available: http://doi.acm.org/10.1145/3133907

[14] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, 2015, pp. 49–64. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/ramos

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

[16] "Cve-2015-0288," https://www.cvedetails.com/cve/CVE-2015-0288/, 2015.

[17] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: exposing missing checks in source code for vulnerability discovery," in *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, 2013, pp. 499–510. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516665

[18] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "Antminer: mining more bugs by reducing noise interference," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 333–344. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884870

[19] S. Saha, J. Lozi, G. Thomas, J. L. Lawall, and G. Muller, "Hector: Detecting resource-release omission faults in error-handling code for systems software," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Budapest, Hungary, June 24-27, 2013*, 2013, pp. 1–12. [Online]. Available: https://doi.org/10.1109/DSN.2013.6575307

[20] S. Krüger, J. Späth, K. Ali, E. Bodden, and M. Mezini, "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs," in *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), T. Millstein, Ed., vol. 109. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, pp. 10:1–10:27. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2018/9215

[21] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, "The blast query language for software verification." in *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, 2004, pp. 2–18. [Online]. Available: https://doi.org/10.1007/978-3-540-27864-1_2

[22] G. T. Leavens, J. R. Kiniry, and E. Poll, "A JML tutorial: Modular specification and verification of functional behavior for java," in *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, 2007, p. 37. [Online]. Available: https://doi.org/10.1007/978-3-540-73368-3_6

[23] T. Ball and S. K. Rajamani, "Slic: A specification language for interface checking (of c)," 2002.

[24] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, 2009, pp. 270–280. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542506

[25] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 306–315. [Online]. Available: http://doi.acm.org/10.1145/1081706.1081755

[26] "Free of memory not on the heap," https://cwe.mitre.org/data/definitions/590.html, 2018.

[27] "Use after free," https://cwe.mitre.org/data/definitions/416.html, 2018.

[28] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 619–629. [Online]. Available: https://doi.org/10.1109/ASE.2015.9

[29] 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. [Online]. Available: http://doi.acm.org/10.1145/1646353.1646374

[30] "Openssl speed tool for performance testing." https://github.com/openssl/openssl/issues/6575, 2018.

[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 *6th USENIX Conference on File and Storage Technologies, FAST 2008, February 26-29, 2008, San Jose, CA, USA*, 2008, pp. 207–222. [Online]. Available: http://www.usenix.org/events/fast08/tech/gunawi.html

[32] "The llvm compiler infrastructure," http://releases.llvm.org/3.9.0/docs/ReleaseNotes.html, 2018.

[33] "Javacpp: efficient access to native c++ inside java," https://github.com/bytedeco/javacpp, 2018.

[34] "Yaml: a human friendly data serialization standard for all programming languages," http://yaml.org/, 2018.

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

[36] "Source code of openssl_1_1_1-pre8," https://github.com/openssl/openssl/releases/tag/OpenSSL_1_1_1-pre8, 2018.

[37] A. Moller and M. I. Schwartzbach, "Static program analysis," 2018. [Online]. Available: https://cs.au.dk/~amoeller/spa/spa.pdf

[38] "Limitations of apisan," https://github.com/sslab-gatech/apisan/issues/7, 2018.

[39] "Manual specification of bn_ctx_get in openssl," https://www.openssl.org/docs/manmaster/man3/BN_CTX_get.html, 2018.
[40] S. Nadi, S. Krüger, M. Mezini, and E. Bodden, "Jumping through hoops: why do java developers struggle with cryptography apis?" in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 935–946. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884790
[41] 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 *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, 2016, pp. 602–613. [Online]. Available: http://doi.acm.org/10.1145/2970276.2970356
[42] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 5, pp. 613–637, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.63
[43] T. Ball, V. Levin, and S. K. Rajamani, "A decade of software model checking with SLAM," *Commun. ACM*, vol. 54, no. 7, pp. 68–76, 2011. [Online]. Available: http://doi.acm.org/10.1145/1965724.1965743
[44] C. L. Linus Torvalds, Josh Triplett, 2013.
[45] S. Jana, Y. J. Kang, S. Roth, and B. Ray, "Automatically detecting error handling bugs using error specifications," in *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, 2016, pp. 345–362. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/jana