Vetting API Usages in C Programs with IMChecker

Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Yu Jiang, Ming Gu, Jiaguang Sun
School of Software, Tsinghua University, Beijing China

***Abstract*—Libraries offer reusable functionality through application programming interfaces (APIs) with usage constraints such as call conditions and orders. Constraint violations, i.e., API misuses, commonly lead to bugs and even security issues. In this paper, we introduce IMChecker, a constraint-directed static analysis toolkit to vet API usages in C programs powered by a domain-specific language (DSL) to specify the API usages. First, we propose a DSL, which covers most API usage constraint types and enables straightforward but precise specification by studying real-world API-misuse bug patches. Then, we design and implement a static analysis engine to automatically parse specifications into checking targets, identify potential API misuses and prune the false positives with rich semantics. We have instantiated IMChecker for C programs with user-friendly graphic interfaces and evaluated the widely used benchmarks and real-world projects. The results show that IMChecker outperforms 4.78-36.25% in precision and 40.25-55.21% w.r.t. state-of-the-arts toolkits. We also found 75 previously unknown bugs in Linux kernel, OpenSSL and applications of Ubuntu, 61 of which have been confirmed by the corresponding development communities.**
Video: https://youtu.be/YGDxeyOEVIM
Repository: https://github.com/tomgu1991/IMChecker

***Index Terms*—API Misuse, Static Analysis, Bug Detection**

I. Introduction

Libraries provide application programming interfaces (APIs) to increase productivity, and these APIs often have usage constraints such as restrictions on call orders or call conditions. Violations of these constraints, which are called API misuses [1], is a prevalent cause of software bugs, crashes, and vulnerabilities [2]. To understand the nature of API-misuse bugs that occur in widely used C programs, we conduct an empirical study of API-misuse bugs and fixes (the details of our empirical study can be found in our repository). The results indicate that API misuses commonly occur because of the following three reasons: incorrect/missing parameter using (IPU), incorrect/missing error handling (IEH) and incorrect/missing causal calling (ICC). Figure 1 shows an example of these misuse bugs, where the missing parameter validation of *fopen* at Line 5 will result in a null pointer dereference bug (IPU); returning SUCCESS when *fopen* fails at Line 10 and incorrectly checking the error code status of *fgets* at Line 15 will break the error handling mechanism (IEH), whereas failure to close the opened file handler *"pFile"* at Line 19 will cause a memory leak bug (ICC).

Many different tools, techniques and methodologies have been proposed to address the above problems. In particular, approaches with code mining and static analysis techniques have proven to be effective. For example, PR-Miner [3] encodes usages as the set of all function names and uses the frequent-itemset mining to find violations with a minimum support of 15 usages of a single API. Ray et al. [4] proposed ErrDoc to explore the error handling bugs by under-constrained symbolic execution using specification, and Yun et al. [5] presented APISan for the causal relation and semantic relation on arguments by semantic cross-checking on intraprocedural paths.

```
1    int foo(char *fileName){
2       char buffer[100] = "";
3       FILE *pFile;
4       // 1. missing parameter validation, resulting NPD
5    +  if(fileName == NULL) return ERROR;
6       pFile = fopen(fileName, "r");
7       if (pFile == NULL){
8       Log("Error open file");
9       // 2.1 incorrect error propagation
10   -      return SUCCESS
11   +      return ERROR;}
12      // 2.2 incorrect error code status checking
13   -  if (fgets(buffer, 100, pFile) < 0){
14   +  if (fgets(buffer, 100, pFile) == NULL){
15         Log("Error read file");
16      // 3. incorrect causal calling, resulting memory_leak
17   +     fclose(pFile);
18         return ERROR;}
19      ...
20      fclose(pFile);
21      return SUCCESS; }
```

Fig. 1: Motivating example of API-misuse bugs.

Despite the vast amount of work on API-misuse detection, these bugs remain widespread in practice [6], [7]. Based on the performance of the existing detectors, we observe that there are two main obstacles to efficient API-misuse detection. **Sparse usage problem.** Mining techniques leverage the key idea that correct usages frequently appear in large corpora, and deviations are bugs with a predefined minimum support to filter out false positives, e.g., PR-Miner only analyzes usages over 15. However, a challenge for such a belief is the "sparse usage problem" [8], where these tools miss the API-misuse bugs under the usage threshold, which is severe for program-specific APIs. **Insufficient semantic analysis.** Most methods are built on abstract syntax trees, i.e., matching the usage based on syntactic information with less semantic information, e.g., Errdoc only supports error handling with constant integers. Moreover, they commonly apply an intraprocedural analysis strategy, e.g., APISan, to address the path-exploration problem in large-scale source code. These approaches will generate false positives and false negatives when the usage contains the point-to relationship or cross-functions (such as *malloc* and *free* in two separate functions).

In this paper, we present IMChecker, which is a constraint-directed static analysis toolkit to augment the current API-misuse detection abilities for large-scale C programs. We propose a domain-specific language, IMSpec, to specify common API usage constraints that leverage the empirical study result

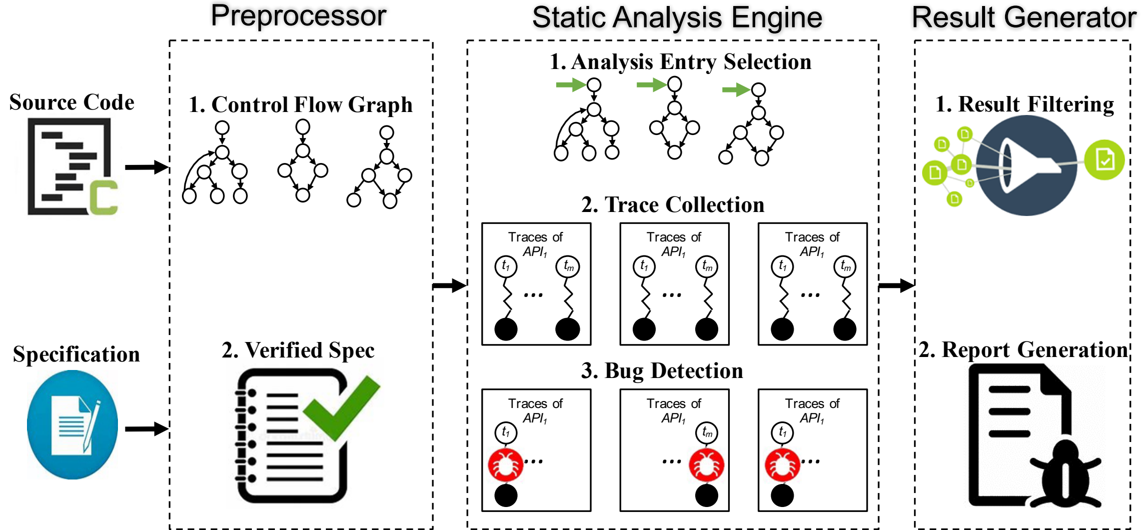


Fig. 2: Framework of IMChecker

of real-world API-misuse bug patterns. Thus, we provide an explicit API usage pattern to direct bug detection to overcome the sparse usage problems. Then, we design and implement IMChecker, a static analysis engine that automatically parses IMSpec into checking targets and detects API-misuse bugs. IMChecker uses the under-constrained symbolic execution [9] to address large-scale programs and performs a soundy [10] approach, which indicates that our technique is mostly sound to achieve a higher precision. For potential API misuses, IMChecker prunes the false positives using rich semantics and multiple usage instances. We package the IMChecker static analysis engine with two user-friendly GUI for IMSpec creation and bug detection result audit as a toolkit, which can be used in a single command line.

For the evaluation, we conduct our experiments on a widely used benchmark, Juliet Test Suite[1]. The results demonstrate that IMChecker has a better precision and recall, which improve by 4.78-36.25% and 40.25-55.21% w.r.t. the state-of-the-arts. We also apply IMChecker to real-world projects such as Linux kernel, Openssl and 5 packages in Ubuntu, which use the OpenSSL library. IMChecker detects 75 previously unknown bugs, 61 of which have been confirmed or fixed by the corresponding development communities.

II. IMCHECKER DESIGN

A. Framework

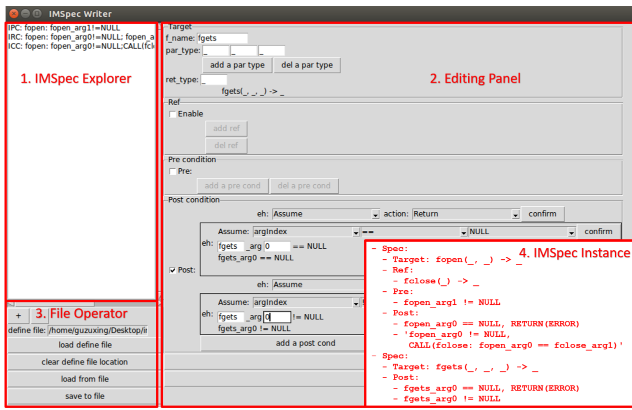
As presented in Figure 2, IMChecker consists of three components. First, the **Preprocessor** parses the source code into an extended control flow graph (CFG) and verifies the API usage specification defined in the IMSpec language. Then, the **Static Analysis Engine** uses the CFG and specifications to select the target analysis entries, collect path traces with rich semantic and detect API-misuse bugs along the traces. Finally, the **Result Generator** filters the bug detection results according to the semantic information and usage statistics and produces the final bug reports.

Preprocessor. The input of our tool contains two parts: source code and target API specification. First, the Preprocess will parse the source code into control flow automata (CFA), which is an extension of CFG, where we classify the edges into two types: (1) *ControlEdge* to carry the concrete statements to conduct semantic computation for the static analysis, and (2) *SummaryEdge* to maintain the program summary information, which is pre-computed to skip the loops and function calls for large-scale programs. Then, the Preprocessor will parse the IMSpec specification according to the syntax of IMSpec and verify the semantic conflicts among the specification. To help the users build specifications, we have implemented a GUI client name IMSpec Writer (see Section II-C). More details of IMSpec, including the syntax, the semantic and examples, can be found in our repository.

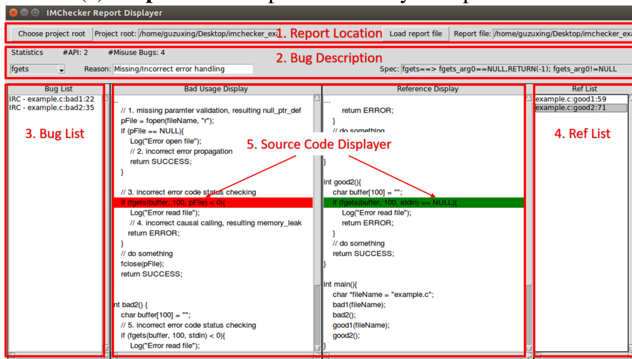
Static Analysis Engine. The static analysis engine is built to conduct API-misuse bug detection using the CFA and specification. Similar to the traditional static analysis, the key challenge of large and complex programs is to overcome the path-explosion problem. We make two design decisions to achieve scalability without sacrificing substantial accuracy: limiting the interprocedural analysis and unrolling loops according to preliminary experiments. In details, our engine consists of three steps. (1) An entry selection algorithm is proposed to select the analysis targets. (2) Then, our engine performs under-constrained symbolic execution to generate program path traces that capture rich semantic information for each target API defined in the specifications. (3) Finally, the engine uses the constraints defined in IMSpec and the program path traces to detect API-misuse bugs.

Result Generator. To achieve the scalability for real-world programs, we use a limiting interprocedural strategy to address the path-explosion problem. Taking the bug traces generated from the static analysis engine, the result generator will first

[1]https://samate.nist.gov/SRD/testsuite.php



(a) **Step 1:** Write specification by IMSpec Writer



(b) **Step 3:** Audit detection results by Report Displayer

Fig. 3: Usage of IMChecker Toolkit

filter out the false positives according to the interprocedural semantics and usage statistics. In the end, we produce the final bug report in a well-defined format. We implement a Report Displayer to help the user audit the bug report.

B. Implementation

IMChecker is built in Java language. We preprocess the source code into LLVM-IR 3.9[2], which provides a typed, static single assignment (SSA) and well-suited low-level language. We parse the LLVM-IR by javacpp[3] and build the CFAs. To this end, we can conduct a path-, field- and context-sensitive analysis. We implement a point-to and range analysis based on abstracted AccessPath proposed by Bodden et al. [11] to collect rich semantic information. Our specification language IMSpec and bug report is formatted in the human-readable data serialization language YAML[4]. We build our GUI using the Python-Tkinter package[5].

C. Usage

We package IMChecker with two user-friendly GUI (IMSpec Writer and Report Displayer) into a toolkit, which can be used in simple command lines. Our toolkit can be used in the following three steps (with the code in Figure 1 as an example):

```
1 ubuntu@~: python3 imspec_writer.py
2 ubuntu@~: python3 engine.py --spec=spec.yaml
      [--specDefine=define.h] --input=example.c
3 ubuntu@~: python3 report_displayer.py
```

Step1: ***imspec_writer.py*** is used to call the writer client as shown in Figure 3a. We produce the specification into the YAML format as illustrated in the *IMSpec Instance* box. Therefore, the users can direct the write specification according to IMSpec syntax.

Step2: ***engine.py*** is used to call the static analysis engine. The engine requires the specification 'spec.yaml' and a compilable c file or a LLVM-IR file, which is compiled by clang with the '-S -emit-llvm -g' options. Parameter 'specDefine' is used to import the macros defined in 'spec.yaml', such as including error codes. For real-world projects that can be built by clang, we provide a build-capture tool 'bcmake' to generate the input files (see our repository for details). We output the analysis status into the terminal. Bug detection results are output in the format of YAML.

Step3: ***report_displayer.py*** is used to call the bug report visualizer to audit the results as shown in Figure 3b. When the users select a target API, the Report Displayer will list all potential bugs on the left Bug List panel. For each bug instance, we provide the bug description and reference usages on the right Ref List panel.

For more details of IMSpec and our tools, the users can refer to the user manual in our repository at https://github.com/tomgu1991/IMChecker.

TABLE I: Evaluation Results on API-Misuse Benchmark

Case Info		APISan		Clang-SA		IMChecker	
Type	Total	Report	TP	Report	TP	Report	TP
IPU	510	310	0	107	105	490	423
IEH	612	446	173	0	0	580	506
ICC	1050	447	435	710	565	1012	878
Total	2172	1203	608	817	670	2082	1807
Precision%		50.54		82.01		86.79	
Recall(A-R^{α})%		27.99-36.58		30.84-42.95		83.20-83.20	

A-R^{α} is recall results with **A**ll cases and **R**efined result that we remove the types where a tool fails to detect all cases, such as IPU for APISan.

III. Evaluation

We evaluate IMChecker on a controlled dataset from the Juliet Test Suite benchmark. IMChecker is compared with two state-of-the-art tools: APISan [5] and Clang Static Analyzer[6], both of which are designed for multiple types of API-misuse bugs and frequently mentioned in other works (Errdoc performs well on real-world projects but only supports IEH bugs). We also apply IMChecker to the latest versions of real-world projects, including Linux kernel-4.18-rc4, OpensSSL-1.1.1-pre8 and packages using the OpenSSL library (e.g. dma, exim, hexchat, httping and open-vm-tools) in Ubuntu 16.04. All tools run on Ubuntu 16.04 LTS (64-bit) with a Core i5-4590@3.30 GHz Intel processor and 16 GB memory.

[2]http://releases.llvm.org/3.9.0/docs/ReleaseNotes.html
[3]https://github.com/bytedeco/javacpp
[4]http://yaml.org/
[5]https://docs.python.org/3.6/library/tkinter.html
[6]http://clang-analyzer.llvm.org/

Table I shows the controlled evaluation results. In total, we select 2172 single cases, each of which contains a bug and several correct usages, to evaluate the precision and recall. There are three types of API-misuse bugs and 13 different Common Weakness Enumeration types (i.e., IPU-CWE121/122/131/476/590, IEH-CWE252/253/390, ICC-CWE401/404/415/690/775).

From the true positive (TP) columns of each tool, we observe that APISan fails to detect the IPU bugs and Clang-SA fails in IEH on this dataset. We investigate the cases and algorithms behind the tools. The result shows that APISan only supports the bugs with an explicit validation checking, which indicates that it will fail in bugs that require a semantic inferring, such as CWE-590 free-memory-not-on-heap. Although Clang-SA provides many checkers targeted at finding API usage bugs, it fails to detect the error handling bugs. Similar to most universal static analysis tools, it hard-codes the detecting algorithm and hardly considers the program-specific semantic. Moreover, Clang-SA prefers a conservative strategy, where they only report the bugs with high confidence to improve the precision. Leveraging a DSL to capture the program-specific properties and a static analysis engine to compute rich semantic, IMChecker finds 1137-1199 more bugs with a more accurate result, which improves 4.78-36.25% in precision and 40.25-55.21% in recall.

The main motivation of IMChecker is to detect API-misuse bugs in real-world programs. In total, IMChecker detects 75 previously unknown bugs. We are currently attempting to create issues and patches for all bugs and send them to the developers of each program. Until now, 61 of the new bugs have been confirmed by the developers (See our repository for details.) For example, in Figure 4, we present a memory-leak bug found in OpenSSL (Issue #6781), which was fixed in two stable versions and the master branch within eight hours after we submitted the issue with a bug description, an explanation of the bug traces and a potential fix strategy.

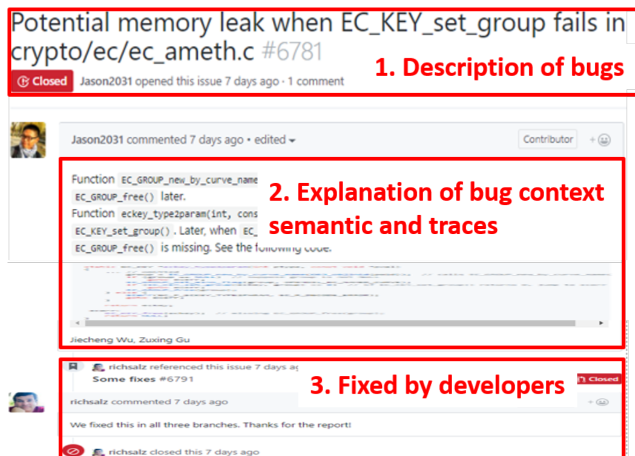


Fig. 4: Screenshot of a memory leak bug found by IMChecker, which is fixed at eight hours on the two stable versions and mainline branch.

IV. Conclusion

Modern APIs are rapidly evolving and error-prone. The incorrect usages of APIs will cause severe bugs. In this paper, we propose IMChecker to vet the API usages in C programs. We evaluated our approach on a widely used benchmark and real-world projects. Our results show that our methods perform better than the current state-of-the-art techniques. We also find 75 previously unknown bugs, 61 of which have been confirmed by the developers. In the future, we will conduct experiments on more programs and investigate the heuristics to automatically rank bug reports based on the potential severity.

Acknowledgment

The author would like to thank the anonymous reviewers. This research is sponsored in part by 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] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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," vol. 1, no. OOPSLA, pp. 83:1–83:29, 2017.

[9] 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.

[10] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[11] 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.