IMSpec: An Extensible Approach to Exploring the Incorrect Usage of APIs

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

***Abstract*—Application Programming Interfaces (APIs) usually have usage constraints, such as call conditions or call orders. Incorrect usage of these constraints, called API misuse, will result in system crashes, bugs, and even security problems. It is crucial to detect such misuses early in the development process. Though many approaches have been proposed over the last years, recent studies show that API misuses are still prevalent, especially the ones specific to individual projects.**

In this paper, we strive to improve current API-misuse detection capability for large-scale C programs. First, We propose IMSpec, a lightweight domain-specific language enabling developers to specify API usage constraints in three different aspects (i.e., parameter validation, error handling, and causal calling), which are the majority of API-misuse bugs. Then, we have tailored a constraint guided static analysis engine to automatically parse IMSpec rules and detect API-misuse bugs with rich semantics. We evaluate our approach on widely used benchmarks and real-world projects. The results show that our easily extensible approach performs better than state-of-the-art tools. We also discover 19 previously unknown bugs in real-world open-source projects, all of which have been confirmed by the corresponding developers.

***Index Terms*—domain-specific language, API usage validation, static analysis, bug detection**

I. Introduction

Today, large and complex software is developed with integrated components using application programming interfaces (APIs). While APIs encapsulate the internal states, correct usage is required to satisfy rich constraints, such as call conditions or call orders. Violation of these constraints, called API misuses [1], is a prevalent cause of software bugs, crashes, and vulnerabilities. For example, improper parameter validation allows remote attackers to cause denial-of-service via crafted parameters to an endpoint (CVE-2018-5803 [1]). A recent study on cryptography-related application interfaces (Crypto APIs) of 11748 Android apps shows that almost 88% of them misuse at least one basic Crypto API [2].

Many attempts have been proposed to address the API-misuse problems [3]–[7]. In particular, static analysis techniques have long prevailed as one of the most promising techniques, since they only require access to source code, which is typically available early in the development process. However, existing approaches are insufficent for several reasons. First, to work on different application domains (e.g., real-time, server and driver), static analysis tools usually employ checkers [3], which are shipped with hard-coded rules to search for specific patterns or pattern violations in API misuses, such as memory leaks or null pointer dereference for APIs of standard C library. While these general checkers are useful, their recall may be quite low, since there are many programming rules that are specific to a software project. Second, to automatically infer the correct usage pattern of an API from other uses of the API, approaches integrating code mining and static analysis have been proven to be effective [4]. But, their detection capability depends on sufficient correct usages for pattern mining, which is unsatisfactory in practice [8]. Moreover, most methods are built on syntactic checks and an intraprocedural strategy, which yield fast analysis times at the cost of exposing a high number of false positives and missing real misuses when the usage contains point-to relationships or cross-functions. Therefore, API misuses still remain widespread [7], [9], even in source code written by experienced developers [10] and patches to known bugs.

In this paper, we strive to improve the current API-misuse detection capability for large-scale C programs. First, we propose IMSpec, a domain-specific language that enables API designers to specify the correct usages constraints in a lightweight special-purpose syntax. IMSpec is specifically designed for three of the most common causes of API misuses: improper parameter using (IPU), improper error handling (IEH) and improper causal calling (ICC). We also present an efficient, yet precise, static analysis engine which parses the IMSpec rules and automatically detects API-misuse bugs with path-, context-sensitive semantics.

We evaluate our approach on a widely used benchmark Juliet-Test-Suite [11], and on the latest version of open-source programs, including OpenSSL and applications of Ubuntu. The results show that our approach improves F-Score up to 35.83-40.96% compared to the state-of-the-art methods. Moreover, our approach detects 19 previously unknown bugs, all of which have been confirmed by the developers (see Table II for details).

In summary, this paper presents the following contributions:

- We introduce IMSpec, a domain-specific language to define correct API usages in a lightweight speicial-purpose syntax, aiming to bridge the cognitive gap between API developers and users.
- We present a static analysis engine, which automatically parses IMSpec rules and detects API-misuse bugs for large-scale projects with rich semantics. Moreover, we

[1]https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-5803

integrate it with a GUI client to help the developers in practice to write IMSpec.
- We evaluate our approach on widely used benchmarks as well as open-source projects, including OpenSSL and applications of Ubuntu 16.06 on the latest version. We find 19 previously unknown bugs, all of which have already been confirmed and fixed by the corresponding developers.

The rest of our paper is organized as follows. Section II illustrates a motivating example of API misuses. Section III introduces our specification language IMSpec and the design of our tailored static analysis engine. We evaluate our approach in Section IV and discuss related work in Section V. We conclude our approach in Section VI.

II. Illustrated 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, error handling, etc. The combination of these elements in an API usage is subject to certain 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].

To classify the misuse, we refer to GENERAL DEFECT CLASSIFICATION (which is proposed for entire domain of all types of software defects) [12], API-MISUSE CLASSIFICATION (which is summarized for Object-Oriented language Java) [4], and the nature characteristics of C language. We summarize three main root causes of API misuse from the changes required to turn the bug into a correct usage by a hand-craft example as shown in Figure 1:

- Improper parameter using (IPU), which occurs if a usage does not ensure the parameter properties that are mandated by the usage constraints. One case is missing null checks, e.g., missing parameter validation of `fopen` at Line-11 will result in a null pointer dereference bug. Moreover, in many APIs, parameters are semantically inter-related. Typical examples are memory operation APIs, such as `memcpy(d, s, n)`, where the size of destination buffer `d` and source buffer `s` should be larger or equal to the copy length `n`. Otherwise, it will cause a buffer overrun bug.
- Improper error handling (IEH), which occurs if a usage does not check the error code status or missing exception handling actions from a possible error that are mandated by the usage constraints. Without an error handling primitives in C language, developers usually employ certain values conventionally to represent the execution status. For example, `fgets` returns `NULL` pointer in case of read error. Checking the error code status of `fgets` at Line-20 incorrectly (i.e. comparing the return value agains 0) at Line-20 will break the error handling mechanism. Moreover, it is important to correctly handle the exception, such as propagate error status upstream

```
1  #define SUCCESS 1
2  #define FILEERR -1
3  #define IOERR -2
4
5  int foo(char *fileName){
6     char buffer[100] = "";
7     FILE *pFile;
8
9     // 1. missing parameter validation, resulting NPD
10 +   if(fileName == NULL) return ERROR;
11    pFile = fopen(fileName, "r");
12    if (pFile == NULL){
13
14    // 2.1 incorrect error propagation
15 -     return IOERR
16 +     return FILEERR;
17    }
18
19 // 2.2 incorrect error code status checking
20 -  if (fgets(buffer, 100, pFile) < 0){
21 +  if (fgets(buffer, 100, pFile) == NULL){
22       goto err;
23    }
24    ...
25    fclose(pFile);
26    return SUCCESS;
27 err:
28    Log("Error read file");
29
30 // 3. incorrect causal calling, resulting memory_leak
31 +  fclose(pFile);
32    return IOERR;
33 }
```

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

using a predefined error code at Line-15/32 or log out the error messages at Line-28.
- Improper causal calling (ICC), which occurs if a usage does not call a certain method that is mandated by the usage constraints. Causal relationship, also known as the a-b pattern is common in API usage. Missing the second function call will result in undetectable problems, such as resource leak. For example, in Figure 1, it forgot to `fclose` the file handler `pFile` along the error handling path at Line-31. It may result in a denial of service in practice. Moreover, there are many constrained causal relationships, for instance only when the return value of `fopen` is not NULL, the file handler `pFile` should be closed.

The detection of API misuses as shown above can be approached through static analyses of source code at the beginning of development process. A key challenge that hinders the existing approaches it "*what is correct*", namely, it requires either specifications of correct API usage or of misuses to find instances of violation. However, it is impractical to hard-code all rules or infer them for diverse project-specific APIs. Moreover, approaches which implement mostly lightweight syntactic checks will produce a lot of false positives and expose a high number of false negatives. For instance, to correctly use an API, the parameter properties have to be guaranteed , where conducting a context-sensitive analysis to compute pointer and integer values is a must.. In addition, path sensitive analysis is needed to correctly handle error handling actions as well as causal callings, and interprocedural analysis is required to handle usages across multiple function calls.

III. Design of IMSpec

As we discuss above, it requires to know "*what is correct*" to detect API misuses. Behavioral interface specification languages allow programmers to express the intended behavior of APIs and have been shown to be useful in assisting the bug finding process [13]. Moreover, it can effectively direct the static analysis tool to validate the manually predefined target APIs, when there is insufficient data to mine the specifications. However, current specification languages for the C code are either designed for general program properties that require a deep understanding of the language to specify a single API's properties (such as BLAST [14] and ACSL [15]), or are too specific to be applied to generic API-misuse detection (such as SLIC [16] for Windows kernel module APIs and Epex [17] for finding error handling bugs). In this section, we introduce IMSpec, a domain-specific language with a lightweight syntax to specify API usage constraints, which covers a majority of the API-misuse patterns. We illustrate the IMSpec syntax elements and a formal treatment of semantics using the cases in Figure 1.

$$
\begin{aligned}
\mathit{Specs} &::= \mathit{Spec}^* \\
\mathit{Spec} &::= \texttt{Spec:}\ \ \mathit{Target}\ \mathit{Pre}\ \mathit{Post} \\
\mathit{Target} &::= \texttt{Target:}\ \ \mathit{FunSig} \\
\mathit{Pre} &::= \texttt{Pre:}\ \ \mathit{Cond}^+ \\
\mathit{Post} &::= \texttt{Post:}\ \ (\mathit{Cond}, \mathit{Action}^*)^+ \\
\mathit{Cond} &::= \texttt{true} \mid \mathit{Opd}\ \mathit{CmpOp}\ \mathit{Opd} \mid \mathit{Opd}\ \mathit{MemberOp}\ (\mathit{Set}) \\
\mathit{Action} &::= \mathit{Return} \mid \mathit{Call} \\
\mathit{Return} &::= \texttt{RETURN}(n) \mid \texttt{RETURN}(\texttt{NULL}) \\
\mathit{Call} &::= \texttt{Call}(\mathit{FunName}:\mathit{Cond}^*) \\
\mathit{Opd} &::= \mathit{Arg} \mid \mathit{UnOp}(\mathit{Arg}) \mid \texttt{NULL} \mid n \\
\mathit{FunSig} &::= \mathit{FunName}(\mathit{Type}^*)\ \texttt{->}\ \mathit{Type} \\
\mathit{Arg} &::= \mathit{FunName}\texttt{\_arg\_}i \\
\mathit{UnOp} &::= \texttt{LEN} \mid \texttt{TYPE} \mid \texttt{MEMTYPE} \\
\mathit{CmpOp} &::= \texttt{!=} \mid \texttt{==} \mid \texttt{>=} \mid \texttt{>} \mid \texttt{<=} \mid \texttt{<} \\
\mathit{MemberOp} &::= \texttt{IN} \mid \texttt{NOTIN} \\
\mathit{Set} &::= id^+ \mid n^+ \\
\mathit{FunName} &::= id \\
\mathit{Type} &::= id
\end{aligned}
$$

Fig. 2: Abstract syntax of IMSpec.

A. IMSpec Design

1) Design Decisions: We aim at providing a purpose-specific specification language to cover majority root causes of API misuse in a lightweight but precise specification language. Therefore, we designed IMSpec with the help of open-source community developers as well as the misuses in practice. In particular, we made the following major design decisions:

- Whitelisting. For a given API, there are many ways to misuse it, but only a few correct usages. We decided to use whitelisting (i.e., specifying the usage constraints explicitly and violations of these constraints are misuses).
- Data flow analysis. Data-flow properties are important in reasoning for the API usages. Many misuses occur when developers forget to manipulate data consistently (e.g., missing resource release when a pointer has been assigned to another location or along every single error handling path).
- API-specific elements. To provide a simple but powerful specification language for API misuse detection, special elements have to be created, especially for properties that are difficult to depict by the syntax of C code (e.g., free memory on heap).
- Tool-independent semantics. We equip IMSpec with tool-independent semantics, which will enable us and others to build more effective tools for diverse purposes (e.g., documentation, test generation, code instructions for dynamic checkers to identify misuses at runtime)

2) Syntax: We use $\mathbb{N}$, $\mathbb{Z}$ to denote the set of non-negative and all integers respetively, and $\mathbb{ID}$ for the set of identifiers. The abstract syntax of IMSpec is shown in Figure 2, where $i \in \mathbb{N}$, $n \in \mathbb{Z}$ and $id \in \mathbb{ID}$.

An IMSpec rule instance ($Spec$) is defined on the level of individual usage of a single API. Therefore, the rule begins by specifying the target API ($Target$) that it is defined for (e.g., `fopen, fgets` as illustrated in Figure 3). For each IMSpec instance, usage constraints are specified with a composition of preconditions (Pre) aimed at parameter using, and post-conditions ($Post$) for error handling and causal calling. The integration of these two parts can cover the three major causes of API misuses as discussed above. *Specifically, we use "_" as a placeholder to represent the elements that are not related to this usage constraint.*

The precondition (Pre) is composed of a set of conditions ($Cond$) to ensure the correct parameter usage. In each condition, we use symbolic variable (Arg) to denote the return value or the formal parameter of the function, which is a compound of the function name ($FunName$), the keyword **arg** and the

```
1  // IMSpec for fopen, which opens the file specified in
       the first parameter. If fails, a NULL pointer will
       be return.
2  Spec:
3  Target: fopen(char*, char*) -> FILE*
4  Pre:
5    - fopen_arg_1 != NULL,
6    - fopen_arg_2 IN (r, w, a, r+, w+, a+)
7  Post:
8  // failure status and error handling actions specific in
       example
9    - fopen_arg_0 == NULL, RETURN(FILEERR);
10 // success status, close file handler
11   - fopen_arg_0 != NULL, CALL(fclose: fopen_arg_0 ==
         fclose_arg_1)
12
13 // IMSpec for fgets, which reads charactors and stores
       them. If a read error occurs, a null pointer is
       returned.
14 Spec:
15 Target: fgets(char*, int, FILE*) -> char*
16 Pre: // omit single parameter validation
17   - LEN(fgets_arg_1) >= fgets_arg_2
18 Post:
19 // failure status
20   - fgets_arg_0 == NULL, CALL(log: true), RETURN(IOERR)
```

Fig. 3: IMSpec instances for misused APIs in Figure 1.

parameter index i. For example,

```
fopen_arg_1 != NULL
```

denotes that the first parameter of `fopen` should not be NULL. Specifically, index 0 is used for the return value (e.g., `fopen_arg_0`). To support the numerical relationship and implicit constraints, which are difficult to explicitly write in the code with the C syntax (e.g., Free of Memory Not on the Heap), we create three built-in unary operators ($UnOp$) to capture these semantics. **`LEN`** is used for the length of the memory pointed to by a pointer; **`TYPE`** presents the type of a variable; and **`MEMTYPE`** indicates where the pointer is pointing (e.g., stack or heap). Therefore,

$$\textbf{LEN}(\texttt{fgets\_arg\_1}) \geq \texttt{fgets\_arg\_2}$$

specifies that the buffer `fgets_arg_1` has to contain more than `fgets_arg_2` bytes. Moreover, some parameters should be constrained in a set of specific values, such as a file access mode being only in $(r, w, a, r+, w+, a+)$. To support such property, we create membership operators ($MemberOp$), where **`IN`** indicates the operand (Opd) should be one of the values in the set (Set) and **`NOTIN`** is the opposite. Therefore,

$$\texttt{fopen\_arg\_2}\ \textbf{IN}\ (r, w, a, r+, w+, a+)$$

specifies the valid file access mode. Note that, all conditions in the precondition should be validated to ensure the correct usage of target API.

The postcondition ($Post$) is used to specify the usage constraints on error handling as well as the causal relationship. For the former, it consists of an error status code checking condition ($Cond$) and a sequence of actions ($Action$) required to handle this error. As shown in Figure 3,

```
fopen_arg_0 == NULL, RETURN(FILEERR)
fgets_arg_0 == NULL, CALL(log: true),
                     RETURN(IOERR)
```

are defined as the error handling actions in our illustrated example. It states that, when `fopen` fails to open the file handler, developers have to propagate **FILEERR** upstream; when `fgets` fails, it has to propagate the error status **IOERR** along the error handling path as well as to output error message by calling `log` function. For the successful execution of the target API, there is no need to provide error handling actions.

For causal calling patterned as `a-b`, we treat the first function `a` as the target and specify the second function `b` in the postcondition. For example, functions `fopen` and `fclose` are causally related in Figure 3. When the causal relationship has contextual constraints, we specify them in the condition part ($Cond$) of the postcondition. That is,

```
fopen_arg_0 != NULL
```

indicates that only when `fopen` is successful, `fclose` is required to close the file handler. Moreover, the data flow of the causal relationship

```
fopen_arg_0 == fclose_arg_1
```

has to be taken into consideration to avoid incorrect close/release or double close/release, which will result in a memory leak or a double free bug. As for calling sequence (e.g. `f_1, f_2, ..., f_n`), we can specify them by a sequence of actions.

3) Semantics: In this section, we present the semantics of IMSpec on programs in form of static single assignment [18], where each variable is assigned exactly once and every variable is defined before it is used.

Definition 1 (Transition State): An execution *transition state* $s_i = (l_s, l_e, inst, \Omega)$ is a tuple of four components, where l_s and l_e are the start and the end program locations respectively, $inst$ is a concrete execution instruction, and Ω is memory valuation to all variables that are in scope at l_e.

Intuitively, l_e is the location after executing $inst$ at l_s. In this paper, we abstract execution instructions by three types, `FCall_fName` for calling function $fName$, `Return_v` for returning v upstream, `Check_cond` for computing a condition $cond$ in an if-statement, and `EMPTY` for others. During the transition, memory valuation Ω is updated depending on concrete semantics of $inst$.

Definition 2 (Trace): An execution *trace* $\tau = s_0, s_1, \ldots, s_n$ is a finite sequence of transition states, starting from program location $s_0.l_s$ and ending at $s_n.l_e$, which satisfies that for any two consecutive states $s_i, s_{i+1} \in \tau$, $s_i.l_e = s_{i+1}.l_s$.

Given an IMSpec instance $spec$ with target API f and an execution trace τ, we use s_Δ to mark the state, whose instruction is a function call to f, and use τ_{post} to denote the trace after s_Δ. Therefore, τ can be represented as $\tau = s_0, s_1, \ldots, s_{\Delta-1}, s_\Delta, \tau_{post}$. Then, the semantics of $spec = (f, pre, post)$ can be formalized with respect to τ: if the function sat, defined in Equation 1, evaluates to false, the function f is misused along the trace τ.

$$sat(spec, \tau) = sat_{pre}(pre, s_{\Delta-1}) \wedge sat_{post}(post, s_\Delta, \tau_{post}) \tag{1}$$

The function sat consists of two components, a validation function sat_{pre} for the precondition pre and sat_{post} for the postcondition $post$.

$$sat_{pre}(pre, s) = \bigwedge_{cond \in pre} eval(cond, s.\Omega) \tag{2}$$

$$eval(cond, \Omega) = \begin{cases} True, & cond \text{ is satisfied in } \Omega \\ False, & \text{otherwise} \end{cases} \tag{3}$$

$$sat_{post}(post, s, \tau) = \bigwedge_{(c, acts) \in post} \Big(eval(c, s.\Omega) \implies match(acts), \tau) \Big) \tag{4}$$

As defined in Equations 2 and 3, when all the conditions in pre are satisfied in the memory valuation before calling f, f is used correctly. Otherwise, a violation will result in a misuse. For each $(c, acts)$ in the postcondition $post$, when condition c holds in the memory valuation $s_\Delta.\Omega$ just after calling f, f is used correctly if the $match$ function defined in Equation 4

is true. Let $as = a_1, a_2, \ldots, a_n$, the function $match(as, \tau)$ is to check the existence of a sequence $\rho = s_{q_0}, s_{q_1}, s_{q_2}, \ldots, s_{q_n}$ with $q_0 < q_1 < \ldots < q_n$ and $s_{q_i} \in \tau$, where the type of each $s_{q_i}.inst$ corresponds to the type of a_i. In particular, if a_i is `Call`$(fName, conds)$, the type of $s_{q_i}.inst$ should be `FCall_fName` and the validation $sat_{pre}(conds, s_{q_i-1})$ should be true.

B. Detecting API Misuses

Static analysis techniques have long prevailed as one of the most promising techniques since it only requires access to the source code and can be applied early in the development process. In this section, we present our static analysis engine, which is tailored to parse IMSpec rules and automatically detect API-misuse bugs with rich semantic and usage statistics.

As presented in Figure 4, our engine consists of three components. First, the **Preprocessor** parses source code into an extended control flow graph (CFG) and verifies that the API usage specification is defined in the IMSpec language. Then, the **Analysis Engine** uses the CFG and specifications to select the target analysis entries, collect path traces with rich semantics and detect API-misuse bugs along the traces. For bug traces, the **Result Generator** filters the bug detection results according to interprocedural semantics and usage statistics, and generates the final bug report.

Preprocessor. The input of our tool consists of two parts: source code and target API specifications. 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:

- *ControlEdge* carries concrete statements to conduct semantic computation
- *SummaryEdge* maintains program summary information, which is pre-computed to skip loops and function calls to support large-scale programs.

Then, the Preprocessor will parse the IMSpec specification and verify the syntax and semantic inconsistency in the specification. To help the users build specifications, we have implemented a GUI client, named IMSpec Writer, as shown in Figure **??**.

Analysis Engine and **Result Generator**. The analysis engine component is built to conduct API-misuse bug detection. Similar to traditional static analysis, one of the key challenge to large and complex programs analysis is path-explosion problem. We make two significant design decisions to achieve scalability yet without sacrificing substantial accuracy: limiting the interprocedural analysis and unrolling loops according to the preliminary experiments.

We present our misuse detection workflow in Algorithm 1. Then input of our algorithm is the control flow automata $\mathbb{CFA}$ and verified IMSpec instance set $\mathbb{SPEC}$ from our preprocessor component. First, for each IMSpec instance *spec*, we parse it and label the target API f. We choose all the callers $\mathbb{CALLER}$ of f as entries to collect symblic traces. Our static analysis engine employs an under-constrained symbolic execution [19] to generate traces with context-, path-sensitive semantics as well as concrete values. For example, we will generate three pathes for the illustrated example in Figure 1 without the patched code:

$$\tau_1 : Line - 6, 7, 11, 12, 15$$
$$\tau_2 : Line - 6, 7, 11, 12, 20, 25, 26$$
$$\tau_3 : Line - 6, 7, 11, 12, 20, 22, 28, 32$$

Because a caller c_f may also contain other target API f' (e.g. `foo` calls `fopen` and `fgets`), we use a static trace map $\mathbb{T}_{map}$ to reuse the traces. Then, we validate the satisfiability by the specification instance $spec$ and the trace τ. For the illustrated example, we will find that: (1) all three traces miss validating the first parameter of `fopen`, (2) τ_1 returns a wrong error code **IOERR** instead of **FILEERR** when `fopen` fails, (3) τ_2 incorrectly checks the return value of *fgets* against 0, and (4) τ_3 forgets to `fclose` the file handler. Therefore, we put all these traces τ_i and $spec$ into bug report R.

Taking the bug traces generated from the static analysis engine, the result generator will first filter out the false positives according to the usage statistics. For example, f is called 10 times in different context. However, more than 2/3 of the usages validate the specification. We will remove these bug reports and warn the users, for the specification may be mis-defined. In the end, we produce the final bug report in a well-defined format with target f, $spec$, misuse location as shown below:

> *API: fopen, Type: IPC, Spec: fopen_arg1!=NULL*
> *Reason: Missing or incorrect validation of parameter*
> *Error: example.c:foo:10*

C. Implementation

IMSpec is written in a human-readable data serialization language named YAML [2]. We build our static analysis engine in the Java language. Source code is preprocessed into

[2]http://yaml.org/

Algorithm 1 Algorithm for checking API-misuse bugs

Input: control flow automata $\mathbb{CFA}$, IMSpec instance set $\mathbb{SPEC}$
Output: bug report $\mathbb{R}$

```
1: ℝ ⇐ ∅
2: 𝕋_map ⇐ ∅
3: for each spec in 𝕊ℙ𝔼ℂ do
4:     f ⇐ parseSpec(spec)
5:     ℂ𝔸𝕃𝕃𝔼ℝ_f ⇐ entrySelection(f, ℂ𝔽𝔸)
6:     for each c_f ∈ ℂ𝔸𝕃𝕃𝔼ℝ_f do
7:         if c_f ∉ 𝕋_map then
8:             𝕋_map ⇐ 𝕋_map ∪ symblicTrace(c_f)
9:         end if
10:        for each τ ∈ 𝕋_map(c_f) do
11:            b ⇐ sat(spec, τ)
12:            if (!b) then
13:                R ⇐ R ∪ addBug(t, spec)
14:            end if
15:        end for
16:    end for
17:    R' ⇐ filterByStatistics(R)
18:    ℝ ⇐ ℝ ∪ R'
19: end for
20: return ℝ
```

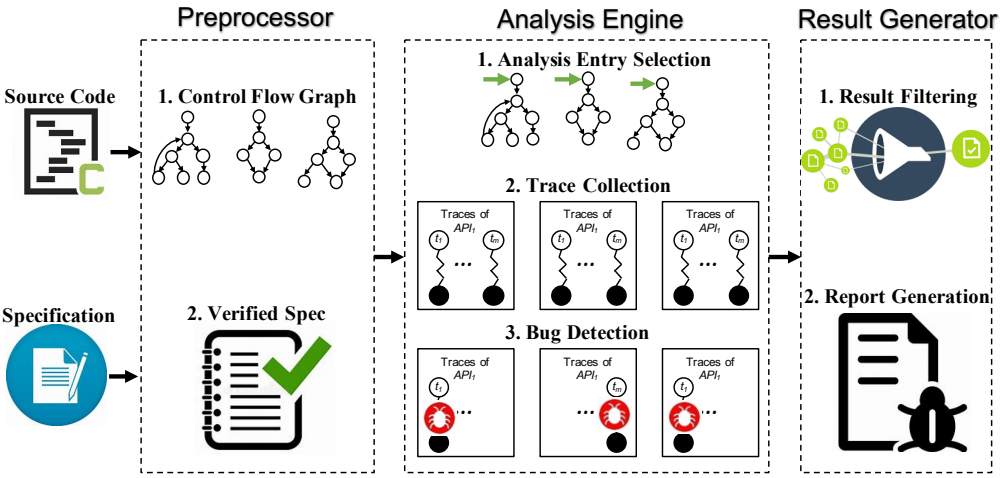

Fig. 4: Framework of tailored static analysis engine.

LLVM-IR 3.9 [3], which provides a typed, static single assignment (SSA) and well-suited low-level language. We parse the LLVM-IR by javacpp [4] and construct the CFA. Point-to analysis and range analysis employ abstracted AccessPath which is proposed by Bodden et al. [20] to collect the rich semantic information.

IV. Evaluation

In this section, we evaluate our approach using the following research questions:

RQ1: How does IMSpec perform on benchmark test cases compared to the state-of-the-arts?

RQ2: Can IMSpec be applicable to real-world projects?

Experiment Setup: We employ a widely used benchmark, Juliet Test Suite V1.3, to build a controlled dataset, where each test case contains a previously known bug and several correct usages. In total, we collect 2172 cases related to API misuse bugs covering 13 different Common Weakness Enumeration (CWE) types [5] (i.e., IPU-CWE121/122/131/476/590; IEH-CWE252/253/390; ICC-CWE401/404/415/690/775) as shown in Table I. IMSpec is compared with two state-of-the-art tools, Cppcheck [6] and Clang-Static Analyzer (Clang-SA) [7], both of which are capable of supporting multiple kinds of API-misuse bugs (by providing user configuration) and frequently mentioned in other works. Moreover, both of them detected real-world bugs that we found in the commits message from the empirical study. (Other tools are either unavailable online or suffer from extremely low precision and recall.) To answer the second research question, we also apply our approach to the latest versions of real-world projects, including OpenSSL-1.1.1-pre8 and packages using OpenSSL library in Ubuntu 16.04. We manually create IMSpec rules, Cppcheck user configuration file and checker configuration of Clang-SA for the APIs used in a controlled dataset. All tools run on Ubuntu 16.04 LTS (64-bit) with a Core i5- 4590@3.30 GHz Intel processor and 16 GB memory.

TABLE I: Evaluation Results on API-Misuse Benchmark

Case Info		Cppcheck		Clang-SA		IMSpec	
Type	Total	Report	TP	Report	TP	Report	TP
IPU	510	145	127	127	105	490	423
IEH	612	298	270	0	0	580	506
ICC	1050	373	337	746	565	1012	878
Total	2172	816	734	873	670	2082	1807
Precision%		89.95		76.74		86.79	
Recall%		33.79		30.84		83.20	
F-Score%		49.13		44.00		84.96	

A. Comparison with Existing Tools

Table I shows the controlled evaluation results. We find that both Cppcheck and Clang-SA miss a large proportion of bugs in this dataset with a recall of 33.79% and 30.84%, respectively. Specifically, from the true positive (TP) columns of each tool, we observe that Clang-SA fails in IEH. To find the reasons, we investigated the cases and algorithms behind the tools. Clang-SA provides many checkers targeted at finding API usage bugs [8] (e.g., security.insecureAPI.UncheckedReturn is built on uses of functions whose return values must always be checked), but it fails to detect the error handling bugs. Similar to most universal static analysis tools, it hard-codes the detection algorithm and fails to support the APIs in this dataset, which is extremely severe for the project-specific APIs. Unfortunately, we find it difficult to extend these checkers to project-specific APIs, such as providing a configuration or target API lists. By contrast, Cppcheck and IMSpec provide a specification language to address this problem. However, Cppcheck only supports syntactic checking, resulting in missing bugs in a complex context. Compared to these two tools, we

[3]http://releases.llvm.org/3.9.0/docs/ReleaseNotes.html
[4]https://github.com/bytedeco/javacpp
[5]https://cwe.mitre.org/
[6]http://cppcheck.sourceforge.net/
[7]https://clang-analyzer.llvm.org/
[8]https://clang-analyzer.llvm.org/available_checks.html

TABLE II: Previously Unknown Bugs Detected by IMSpec

Index	Program	Bug ID	Misuse API	Location (FileName: caller)	Category	Cppcheck
1	OpenSSL-1.1.1-pre8	6567	RAND_bytes	apps/speed.c: **RAND_bytes_loop**	IEH	x
2		6569	ASN1_INTEGER_set	crypto/pkcs12/p12_init.c: **PKCS12_init**	IPU	x
3		6572	BN_set_word	ssl/t1_lib.c: **ssl_get_auto_dh**	IEH	✓
4		6574	EVP_PKEY_get0_DH	ssl/statem/statem_srvr.c: **tls_process_cke_dhe**	IPU	x
5		6781	EC_GROUP_new_by_curve_name	crypto/ec/ec_ameth.c: **eckey_type2param**	ICC	x
6		6789	ASN1_INTEGER_set	crypto/x509v3/v3_tlsf.c: **v2i_TLS_FEATURE**	IEH	✓
7		6820	ASN1_INTEGER_to_BN	crypto/ts/ts_lib.c: **TS_ASN1_INTEGER_print_bio**	IPU	✓
8		6822	BN_sub	crypto/rsa/rsa_ossl.c: **rsa_ossl_private_encrypt**	IEH	x
9		6973	EVP_MD_CTX_new	crypto/ocsp/ocsp_srv.c: **OCSP_basic_sign**	IPU	x
10		6977	ASN1_INTEGER_set	crypto/pkcs7/pk7_lib.c: **PKCS7_set_type**	IEH	x
11		6982	OBJ_nid2obj	crypto/asn1/asn_moid.c: **do_create**	IEH	x
12		6983	BN_sub	crypto/bn/bn_x931p.c: **BN_X931_generate_Xpq**	IEH	✓
13	httping	41	SSL_CTX_new	httping/mssl.c: **initialize_ctx**	ICC	x
14	keepalive	1003	SSL_CTX_new	genhash/ssl.c: **build_ssl_ctx**	ICC	x
15		1004	SSL_new	genhash/ssl.c: **ssl_connect**	ICC	x
16	thc-ipv6	28	BN_new	thc-ipv6-lib.c: **thc_memstr**	ICC	x
17		29	BN_set_word	thc-ipv6-lib.c: **thc_memstr**	IEH	✓
18	FreeRADIUS	2309	BIO_new	src/lib/tls/session.c: **tls_session_pairs_from_x509_cert**	ICC	x
19		2310	i2a_ASN1_OBJECT	src/lib/tls/session.c: **tls_session_pairs_from_x509_cert**	IEH	✓

find 991-1199 more bugs, improving the recall up to 49.41-52.36%.

From the precision result, we find IMSpec achieves a better precision than Clang-SA. The improvement is due to the rich semantics captured by our static analysis engine. In contrast, Clang-SA performs bug detection by diverse domain-specific checkers (e.g. unix.API checks calls to various UNIX/POSIX functions). However, each checker maintains its own abstract states. Thus, it will generate false positives and miss real bugs when the analysis requires deep semantics from multiple domains. For example, pointer analysis usually needs intervals to calculate an explicit offset. Cppcheck achieves the highest precision, for it prefers a conservative strategy, where it only reports the bugs with high confidence, resulting a low recall.

RQ1: In our controlled dataset, IMSpec performs better than the state-of-the-art approaches, improving F-Score up to 35.83-40.96%.

B. New Bugs

The main motivation of IMSpec is to improve the current API-misuse detection capability for real-world projects, namely, to determine whether our approach can find previously unknown bugs. Therefore, we apply our approach to the latest versions of OpenSSL and the applications using it in Ubuntu 16.04. We build the call graph of OpenSSL using a tool named GNU cflow [9]. From the call graph, we choose 30 top functions that are frequently called by other functions in ssl/crypto modules as well as used in target applications. For each API, we create the IMSpec according to user manual of OpenSSL [10]. In total, we found 19 previously unknown bugs from OpenSSL and four applications. We tried our best to submit issues and created patches for all the bugs. Up to now, all of the new bugs have been confirmed and fixed by the developers. We also applied Cppcheck and Clang-SA to these projects. Clang-SA missed all of them and Cppcheck only found six of them without error status code checking.

[9]http://www.gnu.org/software/cflow/

[10]https://www.openssl.org/docs/manmaster/man3/

RQ2: IMSPec can successfully applied to real-world projects. We find 19 previously unknown bugs and all of them been fixed by the developers.

C. Discussion

IMSpec is designed for API-misuse detection purpose and supports three majority causes. To handle large-scale programs, our static analysis engine employs an limiting interprocedural analysis and unrolling loops finite times. Therefore, our approach has false positives as well as false negatives. In the future, we will conduct more experiments to balance the precision and recall. IMSpec relies on user to provide rules, which may be time-costing. But it can be incrementally instantiated and use anywhere by writing once.

V. Related Work

API misuse is a well-known source of bugs and there have been many attempts to address this problem. Static source code analysis has prevailed as one of the most promising techniques since it is typically available at an early stage of development process. In this section, we discuss related work on static analysis for API misuse detection in C.

The behavioral interface specification language can help developers to specify diverse program properties, especially for API usage constraints [13]. The BLAST [14] query language has been proposed by Beyer et al. to specify temporal safety properties of program executions or traces. The ANSI/ISO C Specification Language (ACSL) [15] which is inspired by the Java Modeling Language, can express a wide range of functional properties by means of function contracts and has been implemented in the Frama-C framework [21]. However, these languages focus on the implementation of API instead of usages and requires users to provide explicit annotations or complex syntax structures, which is not so easy to present.

Beyond the general purpose languages, many domain-specific languages are proposed to focus on domain-specific APIs. SLIC [16] is a specification language designed by Microsoft Corporation to specify the properties of Windows kernel module APIs. Sparse [22] uses the developers' annotations

to find certain types of bugs, such as lock/unlock. Cppcheck allows users to provide a configuration to specify the behavior of functions and how they should be used. Jana et al. [17] proposed a method named EPEx to detect error-handling bugs using error handling specifications. IMSpec is proposed to support multiple kinds of API misuses, including resource leak by Sparse, and error-handling bugs by EPEx. Moreover, our static analsis engine automatically detect misuses with rich semantics and performs better than the tools with syntactic checks, such as Cppcheck.

As an alternative to specifying API-usage properties manually, one can attempt to infer them from the existing program code [23] and treat the violations as API misuses. For example, PR-Miner [24] uses a parser to convert source code into itemset database and employs frequent-itemset mining to mine function-pairing rules with a minimum support of 15 usages. Complementary to these approaches, our approach employs IMSpec to overcome the sparse usage problem and tailors a static analysis engine to conduct a rich semantic computation. In addition, the mining results [23] can be used by IMSpec to conduct a more accurate API-misuse detection.

VI. Conclusion

API misuses are common and can have a significant effect on software reliability and security. In this paper, we present IMSpec, a domain-specific language to specify API usage constraints in a lightweight syntax. A static analysis engine is tailored to automatically parse IMSpec rules and detect API-misuse bugs with rich semantics. Evaluating on widely used benchmarks reveals that our approach outperforms the precision of state-of-the-art tools. We also apply IMSpec to open-source projects and find 19 previously unknown bugs, all of which have been confirmed and fixed by the corresponding developers. In the future, we will apply our approach to more projects and try to integrate the results from specification mining techniques to assist users in creating IMSpec rules.

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

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

[3] A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders - test and measurement of static code analyzers," in *COUFLESS*, 2015, pp. 14–20.

[4] 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.

[5] 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 *ICSE*, 2018, pp. 61–64.

[6] R. A. B. Jr., "Code reviews enhance software quality," in *ICSE*, 1997, pp. 570–571.

[7] 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, pp. 602–613.

[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," *PACMPL*, vol. 1, no. OOPSLA, pp. 83:1–83:29.

[9] 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, pp. 886–896.

[10] 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 *(SOUPS)*, August 2018.

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

[12] 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, pp. 470–481.

[13] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. J. Parkinson, "Behavioral interface specification languages," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 16:1–16:58, 2012.

[14] 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.

[15] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, "Acsl: Ansi c specification language," 2008.

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

[17] 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.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *(TOPLAS)*, vol. 13, no. 4, pp. 451–490, 1991.

[19] 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.

[20] J. Lerch, J. Späth, E. Bodden, and M. Mezini, "Access-path abstraction: Scaling field-sensitive data-flow analysis with unbounded access paths (T)," in *ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 619–629.

[21] N. Kosmatov and J. Signoles, "Frama-c, A collaborative framework for C code verification: Tutorial synopsis," in *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, 2016, pp. 92–115.

[22] "Sparse: a semantic parser for c." [Online]. Available: https://sparse.wiki.kernel.org/index.php/Main_Page

[23] 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.

[24] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," in *FSE*, 2005, pp. 306–315.