Ares: Inferring Error Specifications through Static Analysis

Chi Li[1], Min Zhou[1], Zuxing Gu[1], Ming Gu[1], Hongyu Zhang[2]
[1]*School of Software Engineering*, Tsinghua University, Beijing, China
[2]*School of Eletrical Engineering and Computing*, The University of Newcastle, Callaghan, NSW, Australia
chi-li18@mails.tsinghua.edu.cn

***Abstract*—Misuse of APIs happens frequently due to misunderstanding of API semantics and lack of documentation. An important category of API-related defects is the error handling defects, which may result in security and reliability flaws. These defects can be detected with the help of static program analysis, provided that error specifications are known. The error specification of an API function indicates how the function can fail. Writing error specifications manually is time-consuming and tedious. Therefore, automatic inferring the error specification from API usage code is preferred. In this paper, we present Ares, a tool for automatic inferring error specifications for C code through static analysis. We employ multiple heuristics to identify error handling blocks and infer error specifications by analyzing the corresponding condition logic. Ares is evaluated on 19 real world projects, and the results reveal that Ares outperforms the state-of-the-art tool APEx by 37% in precision. Ares can also identify more error specifications than APEx. Moreover, the specifications inferred from Ares help find dozens of API-related bugs in well-known projects such as OpenSSL, among them 10 bugs are confirmed by developers.**
Video: https://youtu.be/nf1QnFAmu8Q.
Repository: https://github.com/lc3412/Ares.

***Index Terms*—Error Handling, Error Specification, Static Analysis**

I. Introduction

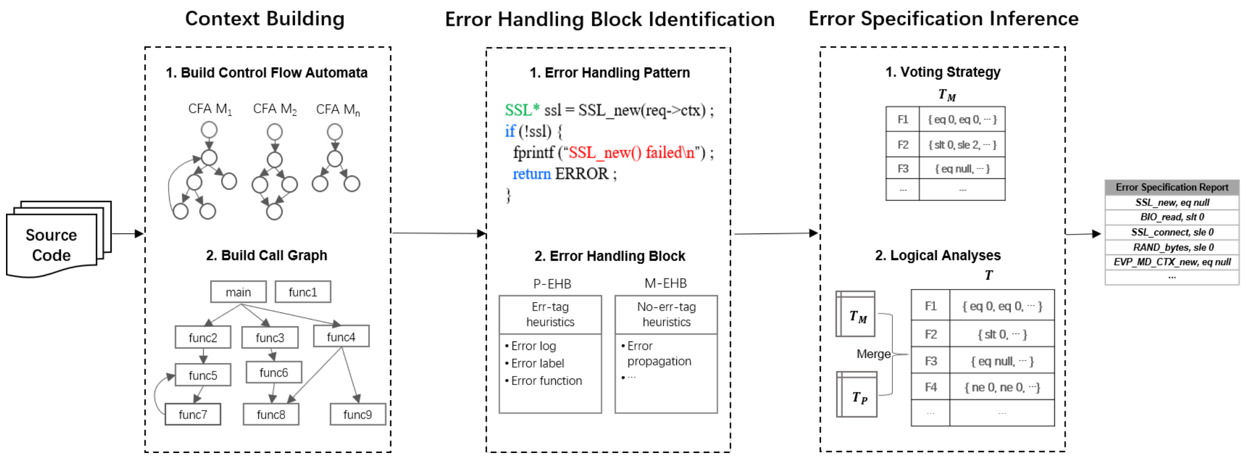
Misuse of APIs happens frequently in programs written in C language, which has no exception handling mechanism. One important category of API-related defects is improper error handling, which is listed as one of the top ten sources of security vulnerabilities [1]. For example, as shown in Figure 1, `SSL_new` creates a new SSL structure, which is needed to hold the data for a TLS/SSL connection [2]. Failing to check the returned value of `SSL_new` (line 5) can put the code in danger of dereferencing a <u>null</u> pointer, which may crash the whole program. Moreover, incorrectly checking the returned value of `SSL_connect` (line 13) may also cause exceptions in the caller. Both of the defects can be eliminated with the help of error specifications produced by static analysis. The error specification of an API function indicates how the function can fail [1]. For instance, when the specification ***SSL_new, eq null*** is provided, it means that `SSL_new` fails when the returned value is <u>null</u>.

Research on automatic error specification inference can be divided into two categories: data mining approaches and program analysis approaches. Research in the first category [3]–[5] aims to find the hidden rules in programs with data mining techniques. EAntMiner [5] extracts statements related to the critical operation as a sub-repository by performing sophisticated preprocessing and then uses frequent item mining to find misuse of API. Techniques based on data mining are fast, but they treat programs as pieces of syntactic elements and suffer from employing various methods to recover semantic information. Research in the second category [1], [6] uses a certain strategy in identifying error handling blocks (EHBs) that contain the error handling code. The corresponding conditions at API call sites are extracted to infer error specifications through statistical analysis. Acharya et al. [6] identify EHBs via the function `exit`, with the insight that `exit` always occurs on the error paths. Kang et al. [1] examine EHBs via the belief error path has fewer statements and branches than regular code. While the techniques used in this type have helped to find error specifications, they rely on a single heuristic which often incorrectly recognizes EHBs and thus have low precision and recall. Dynamic fault injection [7] relies on dynamic analysis of program traces under a lot of test cases and thus has good precision but may have the problem of low recall and time consumption.

To address the above challenges, this paper presents Ares (**A**PI **R**elated **E**rror **S**pecification Inference), a framework with multiple heuristics for inferring error specifications based on static analysis. Ares employs multiple heuristics to achieve accurate identification of EHB and infers error specifications by logical analyses and a voting strategy. We evaluate Ares

```
1   int ssl_operation(thread_t * thread){
2      SOCK *sock_obj = THREAD_ARG(thread);
3      int err;
4      // 1. missing error handling check
5   -  SSL_new(req->ctx);
6   +  sock_obj->ssl = SSL_new(req->ctx);
7   +  if (!sock_obj->ssl) {
8   +     fprintf("SSL_new() failed\n");
9   +     return ERROR;
10  +  }
11     // 2. incorrect error handling check
12     err = SSL_connect(config.ssl)
13  -  if (err < 0)
14  +  if (err <= 0) {
15        fprintf("remote delivery deferred: SSL handshake
             failed fatally: %s");
16        return ERROR;
17     }
18     ...
19     return SUCCESS; }
```

Fig. 1. Motivating example of Error Handling bugs.



```
SSL* ssl = SSL_new(req->ctx) ;
if (!ssl) {
  fprintf ("SSL_new() failed\n") ;
  return ERROR ;
}
```

Fig. 2. The workflow of Ares

on 19 real world projects. Experimental results show that our tool can achieve an overall improvement of 37% in precision compared with the state-of-the-art tool APEx. Our tool can also identify more error specifications (1.24x) than APEx. Based on the inferred specifications, we are able to report dozens of API-related bugs in well known projects, 10 of which are confirmed by the developers.

The major contributions of this paper include:

- A framework as well as multiple practical heuristics for accurate EHB identification and error specification inference.
- A tool named Ares which can be applied to real world projects and the project is available at *https://github.com/lc3412/Ares*.

II. Design of Ares

A. Context Building

Unlike data mining based approaches, our analysis is performed directly on structures like control flow automata (CFA) and call graph (CG). The build process of input project is first captured to obtain source level dependencies, then the source code is parsed and converted to an intermediate representation (LLVM-IR). Construction of CFA is a function-wise conversion from the structured text (as LLVM-IR) to graph representation that is suitable for static analysis. The global CG is then constructed based on the CFA and the function pointer analysis.

B. Error Handling Block Identification

Error handling blocks are identified by multiple heuristics. First, as shown in Figure 2, we find error handling patterns with the help of CFAs and CG. Error handling pattern is the place where error handling possibly happens, which has the similar program structure with error handling code, code for checking the return value of API function, and code for taking some necessary actions. From the CG, Ares collects all the call dependencies ($func : caller1, caller2...$) for each function. Through the call dependencies, we find out whether $func$ satisfies error handling patterns in its callers ($caller1, caller2...$), which can be realized by the traversing the corresponding CFAs of the callers. With the help of CG and CFAs, we can find the error handling patterns of all functions.

TABLE I
Heuristics for identifying EHB

Heuristic	EHB Features	Example
H1	Logs indicating error messages.	fprintf("invalid ..")
H2	Labels indicating error messages.	goto err
H3	Functions indicating error messages.	exit()
H4	Return statements propagate.	return ...

Finding error handling patterns alone is still inadequate to discriminate EHBs. We then identify the features in candidate blocks that indicate the existence of EHBs. For instance, if a "goto err" statement exists inside a block, the block is most likely to be an EHB. To identify the EHB-related features, we performed an empirical study on error handling commits of a number of open source projects (namely Linux kernel, OpenSSL, FFmpeg, Curl, FreeRDP and Httpd). First, we extract commits with commit messages and patches within the studied period ranging from 3 months to 4 years (more details in our repository) related to the *.c files. Next, we select all the error handling commits through keywords (such as "error handling", "missing check", "incorrect check", etc) of the commit messages. Finally, we manually analyze over 300 error handling commit patches and identify four common types of error handling features: Error indicating logs (H1), Error indicating labels (H2), Error indicating functions (H3), and Error propagations (H4). Each of these features can be considered as heuristics for identifying the EHBs, as shown in Table I.

According to the empirical study, EHBs are identified by the following two rules: (1) If a candidate block can be identified by the heuristic H1, H2, or H3, it is regarded as a P-EHB (meaning highly probable EHB). (2) If a candidate block can be identified by the heuristic H4 (meaning there is no obvious error tags), it is regarded as a M-EHB (meaning that the block may be EHB).

C. Error Specification Inference

Error specifications are inferred from logical relationships among conditions of identified P-EHBs and M-EHBs. Static analysis is performed to retrieve corresponding conditions for identified P-EHBs and M-EHBs. Technically, the condition for an EHB is the control dependency which determines whether an EHB will be executed. The condition (c) is a check for the return value of certain function (f) and conditions at different error handling spots are collected as a multimap from f to c. As shown in Figure 2, conditions for identified P-EHBs are stored in T_P and those for M-EHBs are stored in T_M.

First, a voting strategy is used on T_M, because M-EHBs have less error tags for indicating error handling code than P-EHBs. Thus, we apply the following two rules: (1) $min\_support$ is 3, which means the number of occurrences of c is at least 3 times, and (2) $confidence$ threshold is 75%, which means the frequency of occurrence of c is at least 75%.

Then, we merge T_P and T_M into a new table called T, as shown in Figure 2, based on the same function f. Logical analyses are performed in T to validate and refine the conditions: (1) Finding logically inconsistent conditions. For example, if `OBJ_nid2sn` is labeled with ***eq null*** and ***ne null***, which is obviously contradictory, we mark the error specification of `OBJ_nid2sn` as invalid, and (2) Refining conditions. For instance, `setsocket` has conditions: ***slt 0*** (signed integer less than 0) and ***ne 0*** (not equal to 0). Both of them are correct, but the latter condition is entailed by the former one, i.e., $\forall x.x < 0 \rightarrow x \neq 0$. The stronger condition ***slt 0*** is therefore preferred. According to the conditions in T, an error specification report is generated.

III. Implementation and Usage

Ares is implemented in Java. The source code is parsed into LLVM-IR 3.9, which is an intermediate representation in the form of static single representation (SSA). The static analysis algorithm is implemented based on a modified configurable program analysis framework [8]. Deployment of Ares requires the Unix environment with JDK 1.8. All tools run on Ubuntu 16.04 LTS with 16 GB physical memory and Intel Core i7-8700@3.20GHz.

To run Ares, only a few steps of commands are required:

- Step 1: `tsmart-bcmake`
- Step 2: `infer-spec [Source_Dir]`

Our tool is fully automatic. In Step 1, we build the project and capture its build sequence. The captured results are located under `bcmake_output` directory where each file is preprocessed by expanding the macros and in-lining header files. The Step 2 infers error specifications. It first generates merged LLVM-IRs and parses them into CFAs and CG, then performs static analysis. Inferred specifications are written to the *errspec.txt* file. For more details of Ares, welcome to our repository at *https://github.com/lc3412/Ares*.

IV. Evaluation

A. Evaluation on Real World Projects

As shown in Table II, Ares is evaluated on 19 widely-used projects from 5 libraries. The results are compared with APEx [1], which is a state-of-the-art error specification inference tool to date. Both inferred error specifications from APEx and Ares are verified manually against the official documentation. The accuracy of Ares is measured by precision, as defined in $equation(1)$. We also compare the number of correctly identified error specifications returned by Ares and APEx. We compute the $ratio$ according to $equation(2)$.

$$precision = \frac{number\ of\ correct\ errSpecs}{number\ of\ all\ identified\ specifications} \tag{1}$$

$$ratio = \frac{number\ of\ errSpecs\ of\ Ares}{number\ of\ errSpecs\ of\ APEx} \tag{2}$$

1) Precision: Precision values achieved by Ares for all libraries are all higher than those achieved by APEx. Ares has an overall precision of 90.74% while APEx has 65.82%. The improvement is 37.8%. Especially, for the $keepalived$ project in the libc library, our tool finds 32 error specifications and out of which 31 are correct. By contrast, APEx finds 61 error specifications, out of which only 29 are correct. The reason is that the heuristics shown in Table I can effectively recognize EHBs. However, APEx identifies EHBs by counting the number of statements, with the belief that error path has fewer statements, which is not accurate enough for EHB identification. Moreover, it is difficult for APEx to handle the circumstance where EHB has more statements than regular code, which commonly happens in cleanup functions [1].

2) Ratio: Ares can also identify more error specifications than APEx. For each library, the ratio ranges from 0.89 to 10.75. The overall ratio is 1.24. Especially, for the zlib library, our tool finds 45 error specifications, out of which 43 are correct. However, APEx only finds 4 correct ones. The number of identified error specifications reported by Ares for the zlib library exceeds the number reported by APEx 10.75 times. The reason is that the multiple heuristics framework adopted by Ares can identify more EHBs than a single heuristic.

3) Scalable: Ares is hundred times faster than APEx on most of the evaluated projects. For example, in the keepalived project, Ares spends 13 seconds in compiling and 14 seconds in analyzing, while APEx spends 1466 seconds in compiling and 14157 seconds in analyzing. Moreover, APEx terminates unexpectedly in the OpenSSL project due to out of memory.

B. Usefulness in Detecting Error Handling Bugs

To explore the usefulness of our tool, we input the inferred error specifications into the IMChecker [9], which is a tool designed for finding API misuse bugs. As shown in Table III, we have already reported 10 bugs which are all confirmed by developers of the corresponding projects. For example, we

TABLE II
EVALUATION RESULT OF ARES

Project	libc		OpenSSL		libnet		libpcap		zlib		Total	
	Ares	*APEx*	*Ares*	*APEx*	*Ares*	*APEx*	*Ares*	*APEx*	*Ares*	*APEx*	*Ares*	*APEx*
dma	14 / 18	24 / 38	9 / 10	5 / 9	-	-	-	-	-	-	24 / 31	29 / 47
httping	13 / 14	15 / 32	5 / 5	1 / 4	-	-	-	-	-	-	20 / 24	16 / 36
irssi	12 / 15	26 / 47	8 / 8	18 / 21	-	-	-	-	-	-	21 / 26	44 / 68
sslsplit	36 / 37	30 / 44	11 / 11	28 / 33	9 / 9	8 / 10	4 / 4	5 / 5	-	-	60 / 63	71 / 92
thc-ipv6	6 / 6	19 / 27	1 / 1	1 / 2	-	-	3 / 3	3 / 3	-	-	10 / 10	23 / 32
open-vm-tools	66 / 69	57 / 93	1 / 2	22 / 23	-	-	-	-	-	-	69 / 75	79 / 116
OpenSSL	16 / 16	ERROR	125 / 150	ERROR	-	-	-	-	-	-	141 / 167	ERROR
nast	2 / 5	5 / 11	-	-	9 / 9	7 / 7	7 / 7	6 / 6	-	-	18 / 21	18 / 24
rdesktop	23 / 24	22 / 37	5 / 5	0 / 0	-	-	-	-	-	-	27 / 28	22 / 37
keepalived	31 / 32	29 / 61	1 / 1	6 / 7	-	-	-	-	-	-	33 / 33	35 / 68
dpkg	42 / 51	23 / 46	-	-	-	-	-	-	2 / 3	0 / 0	45 / 55	23 / 46
net-speeder	-	-	-	-	1 / 1	0 / 0	4 / 4	0 / 0	-	-	5 / 5	0 / 0
arping	1 / 1	5 / 8	-	-	0 / 0	8 / 8	4 / 4	0 / 0	-	-	5 / 5	13 / 16
bonesi	0 / 0	5 / 10	-	-	5 / 5	4 / 4	2 / 2	4 / 4	-	-	7 / 7	13 / 18
bedops	6 / 6	7 / 8	-	-	-	-	-	-	19 / 19	0 / 0	25 / 25	7 / 8
genwqe-user	15 / 16	15 / 21	-	-	-	-	-	-	6 / 6	4 / 5	21 / 22	19 / 26
SZ	2 / 2	8 / 9	-	-	-	-	-	-	13 / 13	0 / 2	15 /15	8 / 11
npk-tools	4 / 4	13 / 15	-	-	-	-	-	-	1 / 1	0 / 0	5 / 5	13 / 15
matio	3 / 3	8 / 9	-	-	-	-	-	-	2 / 3	0 / 1	5 / 6	8 / 10
Total	292 / 319	311 / 516	166 / 193	81 / 99	24 / 24	27 / 29	24 / 24	18 / 18	43 / 45	4 / 8	549 / 605	441 / 670
Precision(%)	**91.53**	60.27	**86.01**	81.82	**100.00**	93.10	**100.00**	**100.00**	**95.56**	50.00	**90.74**	65.82
Ratio	0.93	**1.00**	**2.05**	1.00	0.89	**1.00**	**1.33**	1.00	**10.75**	1.00	**1.24**	1.00

The Ratio achieved by APEX is regarded as 1.00.
x/y means that y error specifications are reported by the tool and x out of which are confirmed correct.

find a null pointer dereference bug in the project *keepalived* (Issue #1003) using the error specification ***SSL_CTX_new, eq null***, which is fixed by the developer of the project within one day (*https://github.com/acassen/keepalived/issues/1003*).

TABLE III
CONFIRMED BUGS BY DEVELOPERS

Project	Issue#	Error Specification
keepalived	1003	***SSL_CTX_new, eq null***
keepalived	1004	***SSL_new, eq null***
httping	41	***SSL_CTX_new, eq null***
irssi	943	***BIO_read, slt 0***
irssi	944	***SSL_get_peer_certificate, eq null***
sslsplit	224	***SSL_CTX_use_certificate, eq 0***
sslsplit	225	***SSL_CTX_use_PrivateKey, sle 0***
thc-ipv6	28	***BN_new, eq null***
OpenSSL	6567	***RAND_bytes, sle 0***
OpenSSL	6973	***EVP_MD_CTX_new, eq null***

V. CONCLUSION

This paper proposes Ares, a framework for inferring error specifications based on multiple heuristics. It is evaluated on 19 real world projects in 5 libraries. The overall improvement is 37% in precision compared with the state-of-the-art tool APEX. Our tool can also identify more error specifications (1.24x) than APEX. Specifications automatically inferred by Ares also help find 10 bugs in 6 open source projects (keepalived, irssi, thc-ipv6, sslsplit, httping and OpenSSL), all of which are confirmed by the developers. These results show that Ares is effective.

Though our tool can achieve high precision, it still misses some error specifications. In the future, we plan to develop a deep learning based model for identifying error handling code to further improve the identification accuracy.

REFERENCES

[1] Y. J. Kang, B. Ray, and S. Jana, "Apex: automated inference of error specifications for C apis," in *Proceedings of the 31st ASE 2016, Singapore, September 3-7, 2016.* ACM, 2016, pp. 472–482.

[2] "SSL_new." [Online]. Available: https://www.openssl.org/docs/man1.0.2/man3/SSL_new.html

[3] D. Defreez, A. V. Thakur, and C. Rubio-González, "Path-based function embedding and its application to specification mining," in *FSE 2018, FL, USA, November 04-09*, 2018.

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

[5] B. Pan, B. Liang, Z. Yan, C. Yang, W. Shi, and C. Yan, "Detecting bugs by discovering expectations and their violations," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–1, 2018.

[6] M. Acharya and T. Xie, "Mining API error-handling specifications from source code," in *12th International Conference of FASE, York, UK, March 22-29.*, M. Chechik and M. Wirsing, Eds., vol. 5503. Springer, 2009, pp. 370–384.

[7] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer, "Inferring better contracts," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011.

[8] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *23rd CAV, Snowbird, UT, USA, July 14-20.* Springer, 2011.

[9] Z. Gu, J. Wu, C. Li, M. Zhou, Y. Jiang, M. Gu, and J. Sun, "Vetting api usages in c programs with imchecker," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE, Montreal, QC, Canada, May 25-31*, 2019.

[10] H. Zhong and Z. Su, "Detecting api documentation errors," in *OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA*, 2013.

[11] C. Rubio-González and B. Liblit, "Expect the unexpected: error code mismatches between documentation and the real world," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH, Indianapolis, IN, USA, October 26-31*, 2013.

[12] 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.