AURC: Detecting Errors in Program Code and Documentation

Peiwei Hu, Ruigang Liang, and Ying Cao, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China;* Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, School of Cyber Security, University of Chinese Academy of Sciences, China, and Beijing Academy of Artificial Intelligence, China;* Runze Zhang, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China*

https://www.usenix.org/conference/usenixsecurity23/presentation/hu

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

AURC: Detecting Errors in Program Code and Documentation

Peiwei Hu[1,2], Ruigang Liang[1,2], Ying Cao[1,2], Kai Chen[1,2,3,*], and Runze Zhang[1,2]
[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, China
[3]Beijing Academy of Artificial Intelligence, China
{hupeiwei,liangruigang,caoying,chenkai,zhangrunze}@iie.ac.cn

Abstract

Error detection in program code and documentation is a critical problem in computer security. Previous studies have shown promising vulnerability discovery performance by extensive code or document-guided analysis. However, the state-of-the-arts have the following significant limitations: (i) They assume the documents are correct and treat the code that violates documents as bugs, thus cannot find documents' defects and code's bugs if APIs have defective documents or no documents. (ii) They utilize majority voting to judge the inconsistent code snippets and treat the deviants as bugs, thus cannot cope with situations where correct usage is minor or all use cases are wrong.

In this paper, we present AURC, a static framework for detecting code bugs of incorrect return checks and document defects. We observe that three objects participate in the API invocation, the document, the caller (code that invokes API), and the callee (the source code of API). Mutual corroboration of these three objects eliminates the reliance on the above assumptions. AURC contains a context-sensitive backward analysis to process callees, a pre-trained model-based document classifier, and a container that collects conditions of if statements from callers. After cross-checking the results from callees, callers, and documents, AURC delivers them to the correctness inference module to infer the defective one. We evaluated AURC on ten popular codebases. AURC discovered 529 new bugs that can lead to security issues like heap buffer overflow and sensitive information leakage, and 224 new document defects. Maintainers acknowledge our findings and have accepted 222 code patches and 76 document patches.

1 Introduction

Nowadays, library-based programming has become the mainstream software development model, aiming to improve development efficiency, reduce program complexity, and simplify operations such as development and maintenance. Libraries expose Application Programming Interfaces (APIs) for easy use by other developers. Also, library developers use documentation that describes the usage of APIs to help software developers understand how to use the APIs, which also includes the example code sometimes. Lines 2~3 in Listing 1 show the description of `EVP_SealInit()` in the documentation of OpenSSL [4]. By stating zero is returned when errors happen, the documentation guides the software developer to conduct a return check and define the error-handling code for returned zero while invoking `EVP_SealInit()`. Line 17 shows an invocation of `EVP_SealInit()`. The function `openssl_seal` is the caller since it invokes `EVP_SealInit()`. Lines 5~13 partially display the source code of `EVP_SealInit()`, dubbed the callee. Thus, one can refer to the callee, the documentation, or the other callers to learn the usage of APIs, and we call them API Usage References (AURs). In Listing 1, `EVP_SealInit()`, as an initialization function for encryption, returns 0 and -1 while errors happen. However, both the caller in the PHP interpreter [5] and the documentation of OpenSSL omit the negative value. The callee is inconsistent with the caller and the documentation, leading to a Denial of Service (DoS) attack on the PHP interpreter (CVE-2017-11144).

```
1  // the document from OpenSSL(commit:8b9afb)
2  EVP_SealInit() returns 0 on error
3  or B<npupb> if successful.
4  // the callee from OpenSSL(commit:8b9afb)
5  int EVP_SealInit(EVP_CIPHER_CTX *ctx...) {
6      if (type) {
7          if(!EVP_EncryptInit_ex(ctx...))
8              return 0;
9      }
10     /* the error happens */
11     if (ekl[i] <= 0) return (-1);
12     return (npubk);
13 }
14 // the caller from PHP(commit:4b38fea)
15 PHP_FUNCTION(openssl_seal) {
16     /*  ignore the return value -1 */
17     if (!EVP_SealInit(ctx...)) {
18         goto clean_exit;
19     }
20 }
```

Listing 1: Example of Inconsistent AURs

*Corresponding Author

State-of-the-art approaches have found a lot of potential vulnerabilities based on consistency checks. However, they suffer from three main problems. (i) Limited number of APIs are covered by the documentation. Some approaches [41, 46, 48] detect potential bugs by extracting usage from documents and using it as the standard to locate the deviating code. However, many APIs are not documented and escape the detection. What is worse, even the documentation itself may contain defects. For example, the callees of 204 bugs we discovered do not have documentation, whereas the callees of 91 bugs have defective documentation. (ii) Majority voting may be unexecutable or incorrect. Several studies [30, 31, 35, 55] perform extensive code analysis of callers and detect potential bugs based on majority voting, i.e., the most frequent usage is correct. Unfortunately, it is limited to APIs invoked multiple times, and the most frequent usage may also be wrong. For example, the callees of 104 bugs we discovered are invoked too few to perform majority voting, whereas, in the callees of 311 bugs, the dominating usage is wrong. (iii) Correct usage may not exist in the contextual scale. Some work [37, 40, 51] tries to detect bugs based on similar function contexts. For example, they are using similar execution paths within the same function. However, it requires correct usages exist within the context. We observe that all AURs can provide usage implicitly or explicitly instead of utilizing only documents and callers, as in previous studies. Especially the callee, i.e., the source code of the API, exists even if the API is rarely invoked or is undocumented. Therefore, we argue that collecting usage from all AURs and inferring correctness by cross-checking consistency among them can address the above limitations. However, there are challenges in extracting usage and inferring correctness from AURs as follows.

Challenges. C1: The intricate data flow makes it difficult to predict usage from callees. To detect the incorrect return checks by cross-checking among AURs, we entail predicting the return values of callees. The intricate data flows influence this prediction. On the one hand, nested invocations are commonly used for return value assignments. For example, when predicting the return values of `pkey_ec_ctrl_str()` in OpenSSL one has to look through at least 53 functions to trace its origin. On the other hand, even inside the function, the return statements appear in the tails of execution paths, making the traditional analysis technologies, like value range analysis, have to go through many statements before reaching the return statements. Also, the pervasiveness of return statements makes the already heavy analysis even more burdensome. For example, `pkey_ec_ctrl_str()` contains only 32 lines of code, it is surprising that it owns 16 execution paths (ignore nested invocation) and 8 return statements.

C2: Documentation and code cannot be compared directly. The documents are human-oriented and written in natural language, while the callers and callees are code. We cannot compare them directly. Since we use numbers to represent the usage concluded from callees and callers, this challenge equals how to convert the documents to numbers. During the conversion, the fickle sentence structures in documentation impede the extraction of usage-related sentences. Previous studies [46, 48, 57] leverage manually designed templates to filter out the sentences. However, they are labor-intensive and difficult to cope with codebase migration. Moreover, the fickle vocabularies that imply numbers or ranges decrease the accuracy of the conversion. For example, "*BIO_seek() and BIO_tell() both return the current file position on success and -1 for failure*" is from the document of OpenSSL. Descriptive words that imply a range like *position* exist in the sentences. Humans can understand them with a glance but not for automatic analysis.

C3: Determining the defective one when inconsistency happens is a dilemma. After extracting usage from all AURs, finding a reasonable way to infer correctness when inconsistencies occur is crucial. However, the document, the caller, and the callee can all be defective. Assuming that one side is correct is straightforward but not reliable. Several studies [46, 48] locate inconsistency and perform the correctness inference manually, which is labor-consuming. Also, automatic correctness inference is critical since it can be the basis of automatic patching. Some approaches [41] assume one AUR (typically documentation) is correct, while others [30, 31, 35, 55] make the assumption that the most common usage is correct. However, both strategies are frequently incorrect, as we have found many bugs that violate them.

AURC. In this paper, we design an AUR consistency check approach called AURC[1], which aims to find the potential defects in both code and documents. AURC focuses on incorrect return checks, extracting the usage from all AURs to detect inconsistency and sending the found inconsistency to the correctness inference module to conclude the defective one. Also, AURC can overcome the above challenges based on several observations. Specifically, we found that most returned values can be determined by backtracking several statements from return statements, so there is no need to spend plenty of time analyzing the entire function from front to back. Despite the popularity of nested invocations, we can convert this into an intraprocedural problem by replacing invocations with their return values. In our research, we designed a Context-sensitive Backtrace Prediction (CBP) method based on the above observation. CBP predicts the return values by iteratively searching backward in the execution path for assignments of the returned variable. It also simplifies the nested invocations by predicting the invoked functions in advance and replacing the invocations with their return values. Our evaluation shows CBP can predict 90.8% return values with an accuracy of 96.3% (Section 5.3).

In addition, we observed that while the linguistic structure and vocabulary of usage-related sentences keep changing, the semantics, which is closer to human understanding,

[1] AUR consistency Checker.

remains constant. Thus, a semantic-based approach can better cope with codebase migration. Therefore, we proposed a pre-trained model-based classifier to filter out usage-related sentences. We also leverage a mapping table to convert the vocabularies that imply numbers or ranges into actual numbers or ranges. Our experiments show that AURC works effectively facing codebase migration (Section 5.3).

Also discovered in our research is that the mutual corroboration of three AURs eliminates the dependence on assumptions that documents or majority voting are correct. Specifically, if the callee and the document are consistent, the library developer's work is self-consistent, so inconsistent callers should be modified. If the callee and the caller are consistent, then the existing code executes well, so the inconsistent document should be updated. Based on these in-depth observations, we summarized four rules of correctness inference to find AURs with defects. Our correctness inference module has an excellent performance based on statistics of patches approved by maintainers (Section 5.3).

Discoveries. We implemented AURC and evaluated it on ten popular codebases: OpenSSL, libwebsockets, libzip, GnuTLS, net-snmp, mpg123, httpd, libgit2, libxml2, and curl. AURC found 529 new code bugs and 224 new document defects with an accuracy of 87.9%. These bugs can cause concerns like heap-buffer overflow and sensitive information leakage. Until now, maintainers have accepted 222 code patches and 76 document patches. We further detailed analyzed the discovered bugs and found that the callees of 204 bugs have no documents while another 91 callees have defective documents. Also, the callees of 104 bugs are invoked too few to perform majority voting. The callees of 311 bugs do not conform to majority voting. The bug types prove AURC's strength in detecting defects compared to previous work.

Contributions. The contributions of this paper are summarized as follows:

• *New technique.* We design a novel approach to automatically detect code bugs and document defects based on cross-checking AURs (documents, callers, and callees). Unlike previous work, we do not need to assume that documents or majority voting are correct. Our approach can detect bugs that have no documents and do not conform to majority voting. These innovations enable unbiased cross-checking and analysis capabilities between AURs, contributing to the codebase's code robustness and documentation reliability.

• *Implementation and discoveries.* We integrated our ideas into a prototype called AURC [9]. After testing several well-tested projects with AURC, we found 529 new bugs and 224 new document errors, of which 222 code patches and 76 document patches have been merged into repositories by maintainers. We refine both code and documents of widely used codebases and further improve the stability of applications that rely on them. We plan to release our dataset and code to help researchers in the community.

2 Background

2.1 Incorrect Return Checks

Due to the lack of primitive error handling mechanisms, C-based projects often utilize return values to indicate the execution status of functions. Since return values are diverse, it is critical to use the correct way to perform return checks. Otherwise, incorrect return checks happen. The return value indicates the execution status of the callee. For example, `X509_STORE_CTX_get1_issuer()` from OpenSSL could return a negative value in case of error, 0 in case of not found, and a positive value in case of success. A programmer may use the unary operator (!) to check the return value, which confuses the existence of a certificate with an internal error. This increases the risk of the encrypted communication process. Incorrect return checks can cause severe security impacts, which has been extensively discussed in previous studies [30, 31, 41, 55]. Thus, discovering incorrect return checks is security-critical.

2.2 Related Work & Limitations

Recent years have witnessed numerous studies detecting the defects in the codebases, which can be divided into the following classes.

Document/comment analysis. ❶ Some previous studies focus on detecting document errors. For example, Zhong et al. [56] leverage traditional NLP techniques to dig out syntax errors and inconsistent variable names between documents and example code. Zhou et al. [57] convert code and documents to FOL expressions and check the inconsistency by SMT solver to detect erroneous parameter constraints and exception throwing declarations in documents. ❷ Some approaches utilize documents to infer APIs' constraints and detect the deviation. For example, aComment [48] designs an annotation language and converts documents and code to this language to detect concurrency bugs. Advance [41] extracts IAs from documents and leverages the dereferenced IA to generate the CodeQL query statements to detect API misuse. Jdoctor [24] and Toradocu [28] infer API specifications through the Javadoc comments and generate test cases to dynamically detect the API violating the comments. ICON [42] converts the documents to FOL expression and leverages semantic graphs to infer the call order of APIs. Ren et al. [43] extract knowledge from the documents and construct API-constraint knowledge graphs to detect API misuse. However, the above methods assume the documents are correct to extract API usage. They cannot cope with the APIs with defective documents or no documents. ❸ Several studies [44,46,50] do not treat the documents or code as the oracle but discover inconsistencies between code and documents through the decision tree, predefined templates, and heuristic rules. However, they do not propose reliable inference rules to decide the correct one when inconsistency happens.

Code analysis. ❶ Detection of error handling bugs is close to our work since it can also discover some incorrect checks.

Table 1: Comparison of Tools.

Type	Document			Code												AURC
	❶	❷	❸	❶					❷			❸				
				ErrDoc	EPEx	APEx	Ares	Hero	IPPO	CPscan	Arbitrar	Crix	Vanguard	AutoISES	Chucky	
D1	✓	—	✓	—	—	—	—	—	—	—	—	—	—	—	—	✓
D2	—	—	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
D3	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	✓
D4	—	✓	✓	✓	✓	—	—	✓	✓	✓	✓	—	✓	—	—	✓

ErrDoc [49] and EPEx [30] identify error handling blocks and corresponding bugs with predefined error specifications and error report functions. However, providing prior knowledge is labor-consuming for huge codebases. APEx [31] and Ares [34] leverage heuristic rules and majority voting to automatically conclude the error specifications and find error handling bugs, but majority voting degrades the accuracy. Hero [51] leverages EHS stacks and function pairs to detect disordered error handling and reduces the dependence on majority voting. Still, these methods are subject to bugs on error handling paths, while AURC applies to a wider range. ❷ Not limited to error handling, some other approaches avoid majority voting. IPPO [37] assumes the subjects in similar execution paths should follow a similar operation to detect missed security operations. CPscan [27] uses the Linux kernel as the standard and reports the deleted security-critical operations in IoT kernels. Arbitrar [36] involves humans to conclude the correctness of inconsistencies, which owns higher accuracy but is labor-consuming for extensive codebase analysis. Compared with AURC, these approaches cannot detect incorrect checks and defects of documents. ❸ Another set close to our work is missing check detection [40, 45, 47, 54] since it also censors the checks. However, they pay more attention to ensuring the necessity of the existence of the checks while we focus more on whether the checks are performed in the right way. The existence of our targets shows their necessity.

In order to comprehensively and fairly evaluate the performance of the existing work and better demonstrate the effectiveness of AURC's core concepts, we try to leverage the principles of different static analysis frameworks to construct a comparison focusing on several hard-to-detect defects despite some of them do not focus on incorrect checks. Specifically, we highlight the following defects:

D1 *The defects of documents.*

D2 *The defects of callers, and the corresponding documents are wrong or do not exist.*

D3 *The defects of callees.*

D4 *The defects of callers, and the majority voting is not applicable. For example, majority use cases are wrong, or use cases are too rare to perform majority voting.*

We select the aforementioned studies as the targets and evaluate whether they can cope with these defects. The results are shown in Table 1. AURC handles these four types of defects while most tools can only process two types. AURC outperforms because of its ability to cross-check three AURs, as introduced in the following.

3 Approach

In this section, we propose the design of AURC, a novel approach to find the potential defects in both code and documents. We first give an overview of the whole design and an example to show how it works and then elaborate on the details of its components.

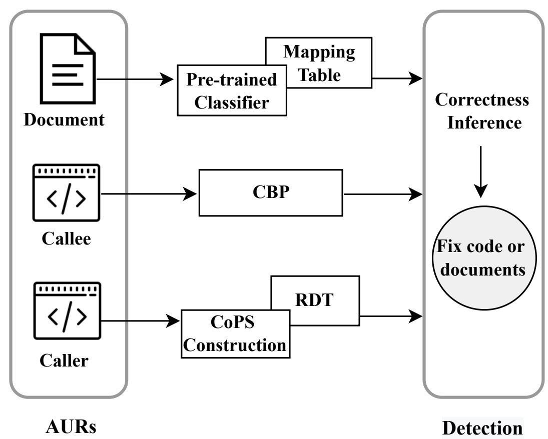

Figure 1: Architecture of AURC. CBP = Context-sensitive Backtrace Prediction, CoPS = Cut-off Point Set, RDT = Range Deduction Tree.

3.1 Overview

Architecture. Figure 1 illustrates the architecture of AURC, including six main modules: Context-sensitive Backtrace Prediction (CBP), Cut-off Point Set (CoPS) Construction, Range Deduction Tree (RDT), Pre-trained Classifier, Mapping Table, Correctness Inference, together with its workflow. Specifically, CBP analyzes the callee, i.e., the source code of the API, and concludes its return values. CoPS works on collecting return checks from the caller, capturing the conditions of if statements, and stores the operands and symbols of the comparisons. RDT deduces the ranges of the return values based on conditions in CoPS. Two modules are responsible for analyzing the documents: Pre-trained Classifier filters out the sentences related to return values; Mapping Table converts them into numbers for further comparison with the usage from callees and callers. After analyzing the AURs (the process can be implemented in parallel), AURC performs a cross-checking to locate inconsistencies and send them to the Correctness Inference, which contains four rules to infer the defective AUR.

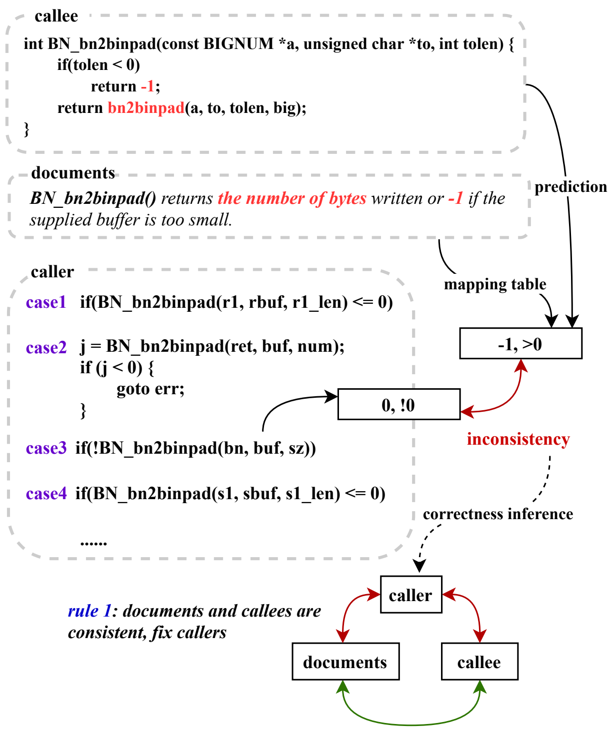

Figure 2: Example of Incorrect Return Check

Example. Figure 2 shows the example of an incorrect return check. The code snippets and documents are extracted from a popular library named OpenSSL. `BN_bn2binpad()` converts the absolute value of parameter `a`[2] into the big-endian form and stores it at parameter `to`. To analyze the callee, i.e., the source code of `BN_bn2binpad()`, AURC deduces its return values by CBP. CBP infers the return values can be -1 or `bn2binpad()`. Similarly, AURC deduces the return values of `bn2binpad()` contain -1 and >0. After taking the intersection, the return values of `BN_bn2binpad()` are -1 and >0. To analyze the documents, AURC first picks out the sentence describing the return values of `BN_bn2binpad()` by the fine-tuned pre-trained model-based classifier. The selected sentence includes a phrase (*the number of bytes*) and a number (-1) to indicate the return values. Then, the mapping table maps the phrase to range >0. Thus, the return values of `BN_bn2binpad()` are -1 and >0. AURC collects conditions of if statements to analyze the callers and picks out those containing comparisons with the callee's return values to construct CoPS, as shown in cases 1-4. Case 3 leverages a unary operator (!) for the return check. RDT further concludes that the return values of `BN_bn2binpad()` contain zero and nonzero based on this return check. They are inconsistent with the results of the callee and documents, i.e., -1 and >0. Thus, the correctness inference module starts working to solve the inconsistency. In this case, the document and the callee are consistent, which means the library developer's work is self-consistent. Thus, the correctness inference module concludes that the caller is defective. In case 3, the caller cannot distinguish two different execution statuses of `BN_bn2binpad()`, i.e., the success represented by positive return values and the exception that the argument `buf` is too small represented by the return value -1. There is a potential risk of operating invalid data in `buf` and causing a program crash. Here we present an example of the callee with one negative return value. Sometimes, the caller uses a range to check multiple return values of a callee. For example, the callee has two negative return values including -1 and -2. If this callee is checked by the symbol "< 0", AURC will not report it since these negative values are consistent with the range of less than zero.

[2]`BIGNUM` is a self-defined structure used for storing numbers.

```
1  int cms_main() {
2      /* ... 946 lines of code ... */
3      ret = SMIME_write_CMS(out, cms);
4      if (ret <= 0) {
5          ret = 6;
6          goto end;
7      }
8      ret = 0;
9  end:
10     /* ret equals to 0 or 6 */
11     return ret;
12 }
```

Listing 2: Simplified `cms_main` from apps/cms.c of OpenSSL

3.2 Analysis of Callee

Value range analysis [29] is an existing approach for calculating variable values. It predicts the possible values of the variable based on the variable type and operations conducted on the variable. Value range analysis plays a role in redundancy elimination and dead code elimination. However, two characteristics hinder its usage in predicting return values. First, value range analysis performs redundant analysis while deducing return values. The return statement comes at the end of the function and is influenced by the returned variable's last assignment. However, value range analysis calculates the variable ranges from front to back and analyzes all assignments of the returned variable. For example, in Listing 2, it is needless to analyze the assignment expression on line 3 to infer the return values, but value range analysis will analyze it. Second, value range analysis has a low efficiency facing nested invocations. For example, in Listing 2, value range analysis has to step into the function `SMIME_write_CMS()` to analyze the line 3. The workload of analysis increases exponentially with the depth of the nested invocations. The return value, as the medium for error propagation, tends to own a long call chain to propagate internal errors, decreasing the efficiency of value range analysis. Another technique that is capable of deducing return values is symbolic execution [23, 25, 33]. It also suffers from low efficiency. Symbolic execution aims to explore more execution paths, leading to its high overhead on solving constraints. However, solving constraints is not a must for predicting return values. Our experiments in Section 5.2 show this low efficiency.

To address the above problems, we propose *Context-sensitive Backtrace Prediction (CBP)*, which predicts the return values of functions backwards. CBP is based on three observations. (i) We can convert the nested invocations to the intraprocedural problem by analyzing the invoked functions in front of the callers and replacing the invocations with their return values. (ii) Value range analysis extracts constraints with constraint derivation rules to build the constraint graph describing the variables' range constraints. Similarly, we summarize three types of path constraints of the returned variable, which can decrease the false positives by excluding unreasonable return values. (iii) Most returned values can be predicted by backtracking some statements from return statements instead of analyzing the whole function from front to back. CBP contains three stages: order of function analysis decision, path constraints extraction, and backtrace prediction.

Order of Function Analysis Decision. CBP analyzes the invoked functions ahead of the caller to replace the invocations with their return values. This stage deduces the analysis sequence to ensure the invoked functions are analyzed before the callers. Specifically, CBP constructs the global call graph and removes the nodes that own back edges, i.e., functions that invoke themselves directly or indirectly, on the call graph. Then, CBP calculates the topological sort of the call graph to get the analysis order. According to this order, the following two stages analyze each function.

Path Constraints Extraction. CBP generates the execution paths of the function by traversing the CFG and collects the path constraints of each execution path. Specifically, after analyzing abundant code, we summarize three types of path constraints that appear frequently. (i) *conditions of if statements* constraint. Suppose there is an *if* statement $if\ (cond)\ \{statement1\}\ else\ \{statement2\}$ and an execution path P which returns the variable R. We also assume that *cond* contains the variable R. If *statement1* is in P, then R should satisfy the condition *cond*. Otherwise, R should satisfy *not cond*. We use C_i to represent the condition that R should satisfy, i.e., *cond* or *not cond*. For the execution path P that contains n *if* statements, the constraint is defined as Equation 1:

$$R\ satisfies\ (C_1 \wedge C_2 \wedge ... \wedge C_n) \tag{1}$$

For example, in Listing 3, the execution path containing lines 3, 4, 5, and 6 has two *if* statements (lines 4 and 5). To reach the line 6, both the two return statements in lines 4 and 5 will not be executed. Thus, `recvd` should satisfy the negation of two conditions, i.e., `recvd != -1` and `recvd != 0`. In this way, CBP will exclude the values -1 and 0 from the return values of the current execution path.

(ii) *subscript* constraint, which is common in the functions that perform searches or queries. CBP detects if the returned value R originates from the subscript of an array with size S. If so, the constraint is defined as Equation 2:

$$R \in [0,\ S) \tag{2}$$

(iii) *loop counter* constraint. Sometimes, the returned variable depends on the induction variable i of a loop. We try to calculate the returned variable's value by estimating the value of i. As we know, calculating the lower and upper bounds of the loop is still an open problem [32, 38, 39]. We estimate the value of i from the loop and use i to calculate the value of the returned variable. In particular, we observe that the value of a loop's induction variable is limited by the initialized value ($Loop_{init}$) and the exit condition ($Loop_{cond}$) of this loop. Thus, we calculate the value of i (represented by V_i) using $Loop_{init}$ and $Loop_{cond}$ by $getInterval\,(Loop_{init},\ Loop_{cond})$, where *getInterval* calculates the interval consisting of two parameters. Further, for the returned variable R, *loop counter* constraint is defined as Equation 3:

$$R \in f\,(V_i) \tag{3}$$

where f represents the calculations from i to R. For instance, in Listing 3, the execution path labeled with arrows returns `i` (line 21), which directly stems from an induction variable in the loop. CBP can infer that the return value of this execution path is within [0, `size - 1`].

Algorithm 1: Backtrace Prediction

Input: *Path*: Execution path;
C: Constraints from path constraints extraction
Output: *R*: Return values

```
1  R ← ∅;
2  ReturnVar ← getReturnedVariable(Path);
3  do
4      if isPredictableObject(ReturnVar) then
5          R ← getValue(ReturnVar);
6          break;
7      end
8      if findReachDef(ReturnVar) then
9          ReturnVar ← getReachDef(ReturnVar);
10     else
11         break;
12     end
13 while True;
14 R ← applyConstraints(R, C);
15 return R;
```

Backtrace Prediction. Given the execution paths with their path constraints of the returned values, CBP predicts the return values backwards in this stage. The analysis of each execution path is shown in Algorithm 1. CBP first gets the returned variable of the execution path (line 2) and checks whether its a *predictable object* (line 4). The predictable objects contain numeric literals, logic expressions representing 0 and 1, invocations whose values can be concluded from the return values of previously predicted functions, and the variables stem from the above objects. The values that predictable objects represent are apparent and can be obtained by CBP directly (line 5). If the returned variable is not predictable, CBP backward searches the reaching definition[3] until it finds a predictable object (line 8). CBP excludes the values contradicting with the constraints obtained from path constraints extraction (line 14). The left values are the return values of the execution path.

Example. In this part, we use `ebcdic_gets()` in Listing 3 as an example to show the process of CBP. First, AURC extracts the execution paths of `ebcdic_gets()`. We focus on the path marked with arrows while predicting the returned `ret`. Then, AURC collects the path constraints on the execution path. Two *conditions of if statements* constraints exist in lines 14 (`ret <= 0`) and 21 (`ret < 0`). After taking intersection, AURC gets the range `ret < 0` by path constraints. Further, AURC starts searching reaching definition of `ret` iteratively and discovers it in line 13 in the first round. Since `ret` is assigned with a predictable object, AURC queries the return values of `ebcdic_read()` in the previously predicted functions. The return values of `ebcdic_read()` contains -2, -1, 0 and >0. Considering the range `ret < 0` from path constraints, AURC discards 0 and >0. Thus, the final prediction result of this path is -2 and -1.

```
1  /* Code details have been simplified */
2  int SocketRecv(int sockFd, char* buf, int sz){
3      int recvd = (int)recv(sockFd, buf, sz, 0);
4      if (recvd == -1) { return -1; }
5      else if (recvd == 0) { return -5; }
6      return recvd;
7  }
8  int ebcdic_gets(BIO *bp, ...) {
9  →  int i, ret = 0;
10 →  BIO *next = BIO_next(bp);
11     if (next == NULL) return 0;
12 →  for (i = 0; i < size - 1; ++i) {
13 →      ret = ebcdic_read(bp, &buf[i], 1);
14 →      if (ret <= 0)
15 →          break;
16         else if (buf[i] == '\n') {
17             ++i;
18             break;
19         }
20     }
21 →  return (ret < 0 && i == 0) ? ret : i;
22 }
```

Listing 3: Code Example of CBP

3.3 Analysis of Caller

Return checks can implicitly reflect the callers' understanding of callees. For example, in Listing 4, `__get_cur_name_and_parent()` deems the return values of `gen_unique_name()` contain two types (<0 and >=0) by using `ret < 0` to perform the return check. Based on this observation, AURC collects all return checks of the invocations and uses them to construct the *Cut-off Point Set (CoPS)*. CoPS is a collection containing conditions of if statements. By analyzing the symbols and operands of the comparison within conditions, AURC can deduce the correspondence between ranges of return values and different execution statuses of the callee from the caller's angle. The element within CoPS is in the format *(callee, symbol, value, location)*. The items *callee*, *symbol*, and *value* stem from the comparison within the conditions of if statements. They record the callee, the symbol of comparison, and the operand of a comparison. They work together for the deduction above. The item *location* records where the checked invocation happens and provides position information when the return check is defective and reported. For example, AURC will convert the return check in line 6 of Listing 4 to *(gen_unique_name, <, 0, __get_cur_name_and_parent:5)* and save it in CoPS. Conditions like `if(!api())` and `if(api())` will be converted to `if(api() == 0)` and `if(api() != 0)` to facilitate the collection.

```
1  int __get_cur_name_and_parent(...) {
2      ret = is_inode_existent(sctx, ino, gen);
3      if (ret < 0) goto out;
4      if (!ret) {
5          ret = gen_unique_name(sctx, ...);
6          if (ret < 0) goto out;
7          ret = 1;
8          goto out_cache;
9      }
10 out_cache: ...
11 out:
12     return ret;
13 }
```

Listing 4: Example of Separated Checks

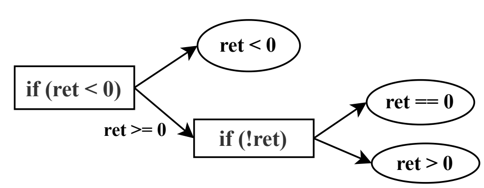

Figure 3: Example of Range Deduction Tree

The diversity of the code style hinders the construction of CoPS. Listing 4 shows as an example. The caller may separately check the invocation. Both lines 3 and 4 check the invocation in line 2. If AURC only considers one of the checks while deducing the ranges, the false positive will be high. AURC constructs the CoPS based on the Data Dependency Graph (DDG) to address the above problem to ensure the completeness of captured checks. DDG is a graph that describes the data dependency relationships. The nodes of DDG represent the statements. The edge between two nodes means the variable in the end node originates from the start node.

[3] Reaching definition is commonly used in compiler theory. If a variable is defined in statement A and used in statement B, and no other assignment of this variable between A and B, A is the reaching definition of B [22].

Specifically, AURC traverses every node containing invocations in DDG. If the node contains a direct comparison like `if (gen_unique_name() < 0)`, AURC can convert and save this in CoPS directly. Otherwise, if the node assigns the invocation to another variable, AURC further traverses every node that depends on this node to collect all comparisons related to this variable. The invocations and comparisons will be saved in CoPS too. This way, since both lines 3 and 4 depend on line 2, AURC can collect them as the return checks for `is_inode_existent()` in line 2.

The separated checks also hinder concluding the ranges of return values from the return checks. For example, `ret < 0` in line 3 separates the range of return values into <0 and >=0, whereas `!ret` in line 4 separates the range into 0 and !0. Since these two checks check the invocation in line 2, they separate the range of return values into <0, 0, and >0. The range >0 is an implicit range since it can not be concluded by any single check of the invocation. We observed that the key to capturing this implicit range is to be aware that the precondition of reaching line 4 is the condition in line 3, i.e., `ret < 0`, is not satisfied, which means `ret >= 0`. Based on this observation, we proposed *Range Deduction Tree*, or RDT for short, to conclude the ranges of return values from the return checks. Specifically, for all return checks of an invocation, AURC first constructs the RDT based on the relationships of these checks in CFG. If one check A is the parent or ancestor node of another check B in CFG, they keep the same relationship in RDT. Also, the edges in RDT label the preconditions of going along these edges. Starting from the root node of RDT, AURC concludes every check with the preconditions in edges to deduce the final ranges. Figure 3 shows an example of deducing the return values' ranges of the invocation in line 2 by RDT. With the help of the preconditions on edges, AURC concludes that the ranges include <0, 0, and >0.

3.4 Analysis of Document

Documents, as one of the AURs, describe return values of APIs in natural language. Extracting return values from documents for comparison is nontrivial for two reasons. First, documents lack strict writing norms. Sentences related to return values are hard to be filtered out since they interweave with other sentences. Second, return value-related sentences are human-oriented and contain phrases that the human can easily understand but not for automatic comparison like "*return the number of characters written*".

To address the first issue, previous work [46, 48, 57] proposed methods that heavily depend on human observation. People look through many pages and conclude some heuristic rules for future extraction, which is laborious. It is also powerless in coping with codebase migration. Advance [41] employs sentiment analysis to address the above problems, finding that desired sentences usually contain strong emotions. Nevertheless, this finding is not universal. For example, documents describe return values of APIs as neutral. Although sentence structures change with codebase migration, which breaks the heuristic rules, the meanings of sentences remain similar. Thus, AURC employs an embedding-based way to identify the desired sentences. The pre-trained model in NLP can convert the sentences in natural language to vector embeddings. The classification based on these embeddings eliminates the dependence on specific rules. Specifically, AURC leverages BERT [26] for classification. We fine-tuned the classifier with the manually labeled dataset that contains sentences from documents. We also design the experiment to show its ability to cope with codebase migration. After testing, the classifier achieves 95.5% accuracy and 94.3% recall on average (see Section 5).

Table 2: Part of Mapping Table

Word	**Range**
nonzero	(-, 0) $\bigcup$ (0, +)
zero	0
length,size,amount,number,index	(0, +)
negative	(-, 0)

The analysis of callers and callees presents the deduced return values in number or range formats. To enable the cross-checking between AURs, we also convert the return values expressed by documents to numbers or ranges. Specially, we design a mapping table (as shown in Table 2) that maps return values described in natural language to numbers. Given a selected sentence, AURC first collects the numbers within it. Then, AURC inspects the nouns within the sentence to find the words in the mapping table. This way, the information hidden in the documents is transformed into comparable forms for further cross-checking between AURs.

3.5 Defects Detection

In the above step, AURC extracts the usage from the three AURs and converts them into numerical form to enable direct comparison. AURC then cross-checks these three AURs to find inconsistencies, i.e., potential defects in the code or documentation. It is nontrivial to determine which AUR is correct when inconsistencies are found. Some previous studies [44, 57] have focused on detecting inconsistencies, such as majority voting, but lacked methods to conclude the defective one. These approaches are limited to one or two AURs in their analysis. Unlike them, AURC utilizes all three AURs and has a more reasonable method to summarize the defects. We summarize four rules for resolving inconsistencies through extensive research and analysis of practical cases (focusing on fixing inconsistencies when they occur), communicating with senior maintainers of several widely used libraries, as shown in Table 3.

Rule 1: *The caller has bugs if it is inconsistent with the callee and the document.* From the perspective of API developers, their responsibility is to develop the callees and describe the usage in documents. If the documents and callees

Table 3: Rules of Correctness Inference

Consistency Check			Modified Subject
Caller	Callee	Document	
✗	✓	✓	Caller
✓	✓	✗	Document
✗	✓	/	Caller
Others			Manual Check

are consistent, the inconsistency is because the caller violates the document usage described while invoking callees.

Rule 2: *The document has bugs if it is inconsistent with the callee and the caller.* If the caller and the callee are consistent, then the code is executed without defects. At this point, updating the documentation will neither impact the existing code nor leave inconsistencies unresolved. Furthermore, due to the rapid evolution of code, it is common to fix outdated documents.

Rule 3: *When the document does not exist, the caller has bugs if it is inconsistent with the callee.* If the description of the callee does not exist in the document, the callee's source code is the only reliable source for providing the usage. Thus, the caller should follow the callee while inconsistency happens.

Rule 4: *If the callee is inconsistent with the document and the caller or all AURs are inconsistent, further manual check is needed.* In both cases, automatic analysis is helpless. Inconsistencies will be collected for manual checking.

The stability of our correctness inference module is evaluated in Section 5.3. During the 298 code or document patches that are accepted by maintainers, 294 of them conform to the correctness inference module, which shows its practical effects.

4 Implementation

4.1 Code Analysis

AURC conducts code analysis based on LLVM infrastructure [18]. Adopting LLVM for code analysis is a common choice [27, 37, 40, 51, 55]. It provides rich interfaces to meet various analysis requirements and reduce development costs. LLVM reads in bitcode files which are closer to what will be executed. It could reduce the chance of compiler bugs that change program semantics. We leverage wllvm [21] to convert the source code to bitcode files. During the generation of bitcode files, the compiler's preprocessor will expand the macros and convert the enumerations to numbers, which eases the analysis of bitcodes. In total, AURC contains 2,500 lines of C++ code for code analysis.

Analysis of callees. During the analysis of callees, CBP generates the execution paths for the backward analysis. We achieve this by traversing the basic blocks along the CFG of the function with the help of interfaces `getEntryBlock`, `getTerminator`, and `getSuccessor`. One concern is the loop statement which owns the backward edge in the CFG. We unroll the loops by treating them as branch statements. This is a widely used method in practice [27, 37, 40, 52, 53]. Besides, CBP identifies path constraints to constrain the range of return values with the help of `ICmpInst`, `GetElementPtrInst`, and `LoopInfo` classes. Currently, our implementation of *loop counter* constraints only supports loops that define the initialized value and the exit condition by numbers.

Analysis of callers. To address the concern that the caller performs return checks of the same invocation in multiple separated conditions of if statements, AURC leverages the DDG to aggregate these checks. LLVM provides the interface `users` to express the data dependency relationships in the DDG. AURC also performs RDT deduction to infer the hidden checked ranges of the return values. To construct the RDT, for each check C obtained above by the DDG, we first find the `BranchInst` that uses it as the condition. By identifying the jump targets of the `BranchInst`, we further collect the checks executed under satisfying or not satisfying the check C. The former performs checks within the range of satisfying C. The latter is with the prerequisite that C is not true.

4.2 Text Analysis

We introduce the technologies adopted in text analysis in this subsection. First, we write scripts to extract the sentences for later classification according to the document formats. Note that these scripts can be reused. Also, we utilized the "bert-base-case" pre-trained tokenizer and model provided by HuggingFace [6] as the basis of our classifier. Specifically, we selected the model pre-trained on dataset "ft-sst3" for our downstream task. We also fine-tuned the model with sentences from the documents of OpenSSL [4], libwebsockets [2], and libzip [3] with the learning rate $2e^{-5}$ and 2 training epochs.

Moreover, for the functions that return macros and enumerations, the documents may describe their return values in the format of macros and enumeration values instead of the numbers they represent. This hinders the comparison between code and documents. To solve this, AURC contains a tree-sitter-based [1] script as a complement to the mapping table. In particular, this script searches for the macro and enumeration definitions in the source code. It further appends the values of macros and enumerations and the numbers they represent to the mapping table. In this way, the mapping table is able to convert the macro and enumeration values in the sentences to numbers.

5 Evaluation

In this section, we evaluate the effectiveness of AURC. First, we tested the overall performance of AURC and the effectiveness of its individual components, such as CBP, correctness inference, and pre-trained model based classifier. Then, we compared it with state-of-the-arts before presenting the exciting findings. All our experiments were conducted on a 64-bits server running Ubuntu 20.04 with 8 processors (Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz), 3TB hard drive, 128GB memory, and 2 GPUs (RTX 3090) with CUDA 11.

5.1 Effectiveness

To evaluate the effectiveness of AURC, we chose ten popular and widely used codebases in the real world. They belong to different areas, including TLS/SSL protocol and cryptography (OpenSSL [4] and GnuTLS [13]), HTTP server and client (libwebsockets [2] and httpd [14]), SNMP protocol support (net-snmp [20]), version control (libgit2 [16]), file parser (libxml2 [17] and libzip [3]), data transport (curl [11]), and audio player (mpg123 [19]). The diversity of codebases demonstrates that AURC can work out of the box on various programs/libraries. During the evaluation process, the callees, the callers, and the documents are from the same codebase. Moreover, AURC is fully automatic. We provide the extraction scripts for popular document formats, including Groff format and Doxygen format. The code analysis on callees and callers is also automatic.

Table 4 shows the results of the evaluation. The average running time of ten codebases is 103 seconds. Even for a huge library like OpenSSL, which owns 695,242 lines of code, AURC can finish the analysis in 4 minutes. Moreover, 80.5% of all detected inconsistencies belong to the first three inference rules, which avoid the human-involved analysis. We submitted the reported documents and code bugs to the maintainers to help them refine the documents' reliability and the codebases' robustness. So far, 76 of document patches and 222 of code patches have been accepted by maintainers. We publish the accepted patches in [9] .

False Positives. Up to now, AURC has reported 857 inconsistencies. We determine the true and false positives by manual analysis. First, we check whether AURC extracts the wrong information from AURs and causes the reported bugs. Second, we check whether the mishandled return values are possible to be returned in the reported position. We treat the reported case as a true positive if it satisfies the above two conditions. Otherwise, it is a false positive. After the manual analysis, 753 of which are code or document defects. The overall false positive rate for AURC is 12.1%, which is much lower than previous studies such as 65% of Crix [40], 63.5% of IPPO [37], and 32.6% of APEx [31][4], according to the statistics these studies provide. We found that it appeared false positive for the following main reasons. (i) Nonexistent execution path (40%). AURC generates the execution paths by traversing the CFG and deduces the return values on the execution path. However, some paths are virtually nonexistent, so the corresponding return values do not exist. These nonexistent return values lead to unnecessary constraints on return checks and further cause false positives. (ii) The mapping table fails to convert return value-related sentences to numbers (22%). To convert the sentences within documents to comparable numbers for further cross-checking, AURC performs the mapping by the mapping table. However, some words do not exist in the mapping table. This omission leads to incomplete information extraction from documents and the misunderstanding that documents are defective. A potential solution to address this limitation is complementing the mapping table during the application of AURC. (iii) Separation between API and its description (13%). The document is prepared for humans and loosely structured. Thus, different APIs, with their descriptions, may mix thoroughly, hindering the association between the descriptions and their corresponding APIs.

False Negatives. To evaluate the false negative rate of AURC, we constructed a dataset containing 450 defects. They equally distribute in documents, callers, and callees. For documents, we randomly selected 150 pages and modified the words describing return values. These pages do not overlap with the sentences for fine-tuning the pre-trained model. We also randomly selected 150 functions and changed their return statements to simulate the faults of callees. For callers, we randomly chose 150 functions containing return checks and modified the symbols of checks. After testing, AURC omitted 20 document defects and 21 code defects. The overall false negative rate is 9.1%. Moreover, we analyzed the results of AURC and found that it appeared false negative for the following reasons. (i) Return value-related sentences lack clear subjects (48.8%). As discussed in the last subsection, the document is prepared for humans and loosely structured. Thus, different APIs, with their descriptions, may mix thoroughly, hindering the association between the descriptions and their corresponding APIs. "*All other functions return 1 on success, 0 on error*" from OpenSSL is an example. Its unclear subject makes AURC fail to gather the descriptions of some APIs. (ii) Indirect calls hinder the prediction of return values (34.1%). The issue of indirect calls is still an open problem and is closer to the dataflow analysis scope instead of our core idea. (iii) Other reasons. There are some other reasons leading to false negatives. For example, the return values stem from the structure member of the parameter. The return values depend on an invocation to an extern function that the library does not contain its source code. Under these situations, the return values can not be statically predicted.

5.2 Comparison with the State-of-the-Art

Comparison with other detectors. To evaluate the effectiveness of AURC, we selected three state-of-the-art tools [30, 31, 41] that also detect incorrect return checks and experimented with how many defects they can find among all defects that AURC found. Note that the ten codebases we selected are also the codebases that prior work performs well. For example, according to the statistics from the previous studies, Advance found the second-most and third-most bugs on libxml2 and OpenSSL, respectively. EPEx found the most and second most bugs on OpenSSL and GnuTLS, respectively. APEx found

[4]The false positive rates of IPPO and Crix are directly provided by their authors. The false positive rate of APEx is calculated by the found bugs the authors provide.

Table 4: Effectiveness of AURC.

Codebase	Inconsistency Detection				Inconsistency Type				Running Time (s)			
	Report	Code/True	Doc/True	Acc	Rule 1	Rule 2	Rule 3	Rule 4	Classification	CBP	CoPS	Detection
OpenSSL	534	424/403	110/83	0.910	178	67	135	106	22.80	222	71	0.55
libzip	2	2/2	0/0	1	0	0	2	0	1.47	4	1	0.003
libwebsockets	8	0/0	8/8	1	0	8	0	0	1.75	34	8	0.047
GnuTLS	35	22/22	13/8	0.857	0	8	20	2	3.29	63	21	0.19
curl	2	2/2	0/0	1	0	0	2	0	12.65	46	11	0.064
mpg123	7	5/5	2/2	1	0	1	5	1	1.71	13	2	0.029
httpd	20	0/0	20/16	0.800	0	12	0	4	8.13	172	32	0.077
libgit2	129	46/37	83/73	0.852	5	46	31	28	14.02	57	16	0.155
libxml2	106	60/51	46/29	0.754	41	29	5	5	26.36	69	13	0.12
net-snmp	14	9/7	5/5	0.857	5	4	2	1	4.37	58	18	0.087
Average	–	–	–	–	–	–	–	–	9.65	74	19	0.135
All	857	570/529	287/224	0.879	229	175	202	147	–	–	–	–

second most and third most bugs on OpenSSL and GnuTLS, respectively. Table 5 shows the results, and the experiment steps are discussed in the Appendix.

Table 5: Comparison with Other Approaches. Since APEx, EPEx, and Advance cannot detect the defects of documents, the listing results are code bugs. N^* means Advance can find N bugs at most, see Appendix A.

Codebase	**AURC**	**APEx**	**EPEx**	**Advance**
OpenSSL	403	0	80	16
libzip	2	0	0	0*
libwebsockets	0	0	0	0*
GnuTLS	22	0	0	0*
curl	2	0	0	0*
mpg123	5	0	0	0*
httpd	0	0	0	0*
libgit2	37	timeout	0	5*
libxml2	51	0	10	9
net-snmp	7	0	2	0*
All	529	0	92	30*

Advance [41] detects code defects based on the usage extracted from documents. Take OpenSSL as an example. While AURC found 403 code bugs of OpenSSL, Advance only extracted 37 bug-related sentences. Two reasons lead to this. First, documents lack the description of many APIs. Second, sentiment analysis is not the silver bullet to extracting security-critical sentences from documents since they may be in neutral sentiment. Moreover, 21 of extracted sentences provide incorrect knowledge, which indicates it is untenable to assume the documents are always correct.

APEx [31] found no code bugs. APEx first infers the error specifications from the callers based on the return checks' statistical features, which include the number of subsequent paths and statements. However, when return checks are too less to infer, or the statistical features deviate from the inference rules, APEx fails to get the error specifications, which hinders the following bug detection. After the above inference, APEx leverages majority voting to dig out the callers that check the return values incorrectly. However, our analysis in Section 5.4 shows that majority voting does not apply to 58.5% of code bugs that AURC found.

EPEx [30] leverages provided error specifications to detect error-handling bugs. We first ran EPEx with the error specifications predefined by the authors, but it found no bugs that AURC reported because the error specifications cover few APIs. We then manually define the error specifications of the callees of the incorrect checks AURC reported, which costs about 2 hours. EPEx found 92 code bugs. Two reasons limit EPEx in detecting bugs. First, EPEx heavily depends on predefined knowledge. One entails manually defining the error specification of the functions, which is pretty labor-consuming since it is normal for codebases to own thousands of functions. Second, EPEx adopts majority voting to detect bugs. Once finding a potential bug, EPEx will ignore this bug if it is consistent with the most frequent usage. For example, EPEx found two code bugs in mpg123 but later filtered out findings because of majority voting. Majority voting is untenable, as discussed in APEx.

Comparison with symbolic execution engines. We compare CBP with two existing symbolic execution engines focusing on the ability to conclude return values. One is KLEE [25], which is widely used and famous for generating inputs to thoroughly explore the execution paths. The other is APEx [31], an under-constrained symbolic engine aiming to reveal error specifications based on heuristic rules. In particular, we randomly extract 200 functions with integer return types from ten codebases. The selected functions make up the test suite. For KLEE, we leverage scripts to generate the invocation of the target function and deliver symbolic arguments by `klee_make_symbolic`. The process accords to [15]. The setup of APEx follows the instructions of [8]. As presented in Table 6, the results show that CBP is 5 times and 1000 times faster than APEx and KLEE, respectively. Moreover, CBP outperforms KLEE and APEx in terms of prediction accuracy.

Table 6: The Ability of Predicting Return Values

Name	Accuracy	Time (s)
CBP(AURC)	93%	26
APEx	35%	156
KLEE	51%	26703

KLEE is very time-consuming compared with APEx and AURC. KLEE aims to generate inputs to execute as many paths as possible. Thus, it spends much time tracing call chains and solving constraints to produce the inputs that satisfy the constraints. The inaccuracy of KLEE is because the unsolvable constraints hinder the generation of the inputs for the corresponding paths. APEx predicts the return values based on the checks in the callers. If the callers do not check all possible return values of the callee or contain errors, APEx will fail to find the correct return values of the callee. The results show that CBP, which is specially designed for predicting return values, can better cope with this task compared with KLEE and APEx.

5.3 Evaluation of Indivudial Components

Performance of CBP. To evaluate the performance of CBP, we manually analyzed 300 functions of ten codebases and found that AURC mistakenly predicted only 11 functions. The overall accuracy is 96.3%. Specifically, 9 functions are because CBP searches the wrong reaching definition or cannot find the reaching definition of the returned variable, and another two stem from the virtually non-existent execution paths. They reflect the inherent limitation of static analysis. Also, we counted the return values that AURC cannot deduce while testing it on the ten codebases. CBP can predict 90.8% of all return values in total. Regarding the return values that AURC failed to deduce, 43.9% of them are due to the return values stemming from the arithmetic calculation. 33.4% of them are because the return values are affected by the global variables or parameters, leading to failure to search for the reaching definition. Moreover, indirect call causes 13.2% of failures, and access to the pointers and fields of structures leads to 9.5% of failed cases. These failures are mainly because of the inherent limitations of static analysis.

To quantitatively represent the role of backward analysis in CBP, for each function, we define:

$$Cov = \frac{LC(return\ statement) - LC(CBP\ finishes)}{LC(return\ statement)}$$

where $LC(N)$ represents the Lines of Code from the function entry to the statement N except the comments and blank lines. "$LC(return\ statement) - LC(CBP\ finishes)$" represents how many lines the backward analysis needs to scan to predict the return values, while " $LC(return\ statement)$ " represents how many lines the forward analysis needs to scan. We calculated Cov of ten codebases. The average values is 12%, which shows that 88% of code does not need to be analyzed. In this way, CBP can effectively skip unimportant statements compared with the forward analysis.

As discussed in Section 3.2, CBP can overcome the problem of path explosion caused by nested invocations by replacing the invocations with their return values. We counted the difference in the number of paths due to the replacement. In particular, we define:

$$PathRate = \frac{NumberOfPaths(replacement)}{NumberOfPaths(no\ replacement)}$$

where $NumberOfPaths(replacement)$ represents the number of execution paths for analysis with nested invocation replacement, $NumberOfPaths(no\ replacement)$ represents no replacement. During the evaluation, the maximum nesting depth is limited to three. The average value of $PathRate$ on ten codebases is 0.06%, which means that CBP can save the analysis of 99.94% paths by nested invocation replacement. Since it is common for a function to own a call chain longer than three, $PathRate$ is smaller than 0.06% in practice.

Performance of correctness inference. We evaluate correctness inference with the patches accepted by codebase maintainers. During the 298 patches, 294 of them conform to the rules of correctness inference. The other 4 patches are inconsistencies between the callers and the callees, with no existing document. According to rule 3, the caller should follow the callee. However, in these cases, the callees are rarely-used internal APIs, owning only one invocation in the whole codebase. Thus, maintainers decide to patch them unusually, i.e., ignoring the other callers that depend on the callees and modifying the callees directly.

Performance of model-based classifier. We also evaluated the performance of the pre-trained model-based classifier. We randomly divided ten codebases under testing into three groups to construct the datasets: Group1 contains OpenSSL, mpg123, and httpd; Group2 consists of GnuTLS, libgit2, and libwebsockets; Group3 includes libzip, net-snmp, curl, and libxml2. Each group is composed of 1,000 sentences describing return values and 1,000 irrelevant sentences. It cost 172 minutes to label the sentences. Manual labeling is an one-time effort and does not need to be performed for each new codebase. We also divided each group into training, testing, and validation sets in the ratio of 8:1:1.

Table 7: Performance of classifier.

	Group1		**Group2**		**Group3**	
	Acc	*Recall*	*Acc*	*Recall*	*Acc*	*Recall*
Group1	99.5%	99.2%	89.5%	82.3%	95.2%	91.8%
Group2	90.8%	98.8%	99.9%	99.8%	94.8%	94.8%
Group3	97.4%	96.1%	92.5%	86.0%	99.9%	100%

The main goal of our evaluation is to evaluate the model's ability to cope with codebase migration, i.e., the model trained on one codebase also works on another codebase. This ability makes our method superior to those based on heuristic rules. Thus, we designed a cross-checking experiment. We trained the model on one dataset and tested its performance on another two datasets. Table 7 shows the results. The three groups in the first column show the dataset used for training in the corresponding row. The groups in the first row show the dataset used for testing. For example, 89.5% in the fourth column of the third row represents the accuracy while using Group1 for training and Group2 for testing. The average accuracy and recall are 95.5% and 94.3%, respectively. The results show the classifier has decent performance even when the training datasets are different from the testing datasets, showing the generalizability of our approach. In practice, we can use the model fine-tuned on a diverse dataset to achieve higher accuracy and recall.

5.4 Findings

Bug types. We found plenty of bugs from the ten codebases, although they experienced thorough tests by previous work. To find out the reason, we analyzed the bugs' distribution. The column "Inconsistency Type" in Table 4 shows the characteristics of the bugs grouped by the correctness inference rules. 229 conform to rule 1 while 175, 202, and 147 conform to rule 2, rule 3, and rule 4, respectively. Besides, we find that the callees of 204 bugs have no documents while the other 91 callees have defective documents. Moreover, 104 bugs do not have enough cases to perform majority voting, and 311 bugs do not conform to majority voting. Thus, the majority voting is not applicated to 415 bugs. The distribution accounts for why previous work which depends on majority voting and documents cannot find these bugs. Moreover, we further break down the found bugs focusing on whether they come from different root causes. We assume that two bugs share the same root cause if the callers of these two bugs incorrectly check the same callee in the same way. After manual analysis, 184 of these bugs don't share the same root cause, which shows AURC's ability to detect unique issues.

Security impacts. To evaluate the security impacts of findings, we adopted the Common Weakness Enumeration [10] as our standard and manually analyzed 100 bugs to evaluate the security impacts of AURC's findings. We found the bugs AURC reported conform to a wide range of CWE's categories. First, all code bugs conform to CWE-253: Incorrect Check of Function Return Value, which reveals the practical value of the issue AURC concentrates on. Besides, we further found that 27% of bugs conform to one of the following categories: CWE-1270: Generation of Incorrect Security Tokens (5%), CWE-122: Heap-based Buffer Overflow (1%), CWE-330: Use of Insufficiently Random Values (3%), CWE-226: Sensitive Information in Resource Not Removed Before Reuse (1%), CWE-295: Improper Certificate Validation (1%), CWE-393: Return of Wrong Status Code (5%), CWE-703: Improper Check or Handling of Exceptional Conditions (11%). We present five case studies in Appendix to show the security impacts.

6 Discussion

Lessons from Incorrect Checks. After studying numerous cases of incorrect return checks, we summarize four rules from the aspects of API developers and users to mitigate the occurrence. *(i) Use a uniform error specification.* We found many errors in OpenSSL compared to other codebases in our experiment because it is designed with two different error specifications; OpenSSL should adopt a uniform error specification to reduce error return checking. *(ii) Make all APIs follow the error specification.* After defining the error specification, the codebase should ensure all APIs follow it. Otherwise, API users may check for functions that deviate from the error specification in a way that conforms to it. *(iii) Return enumerated values to indicate errors.* We found curl [11] implements an elegant mechanism to indicate errors by returning enumerated values. The enumeration limits the API users to perform return checks within the range of this structure. *(iv) Code and documents should be updated simultaneously.* Documents should be carefully maintained and updated promptly as an essential guide to API usage.

Port to Missing Resource Release. The proposed CBP can also be ported to check for missing resource releases. To find the missing resource releases, one needs to collect the functions that allocate the resources. The function `malloc()` is a primitive function to allocate resources. However, mature software customizes the resource allocation functions to fit specific situations. For example, one self-defined allocation function may invoke `malloc()` and then return the allocated pointer. In this situation, CBP can conclude the source of the returned variable backwards and infer the current function is a resource allocation function if the returned variable stem from another resource allocation function.

7 Conclusion

In this paper, we present AURC to detect code and document defects based on cross-checking of AURs. Leveraging the classifiers, CBP, and CoPS collection, AURC collects usage from three AURs. Running on the ten famous open-source codebases, AURC successfully detected 529 new bugs and 224 new document defects. Maintainers have accepted 222 code patches and 76 document patches, proving that AURC refines both the codebases' code robustness and document reliability.

Acknowledgments

We want to thank our shepherd and reviewers for their insightful comments which highly improve our paper. The authors are supported in part by NSFC (U1836211, 92270204), Beijing Natural Science Foundation (No.M22004), Youth Innovation Promotion Association CAS, Beijing Academy of Artificial Intelligence (BAAI) and a research grant from Huawei.

References

[1] Tree-sitter. https://tree-sitter.github.io/tree-sitter/, 2020.

[2] libwebsockets. https://github.com/warmcat/libwebsockets, 2021.

[3] libzip. https://github.com/nih-at/libzip, 2021.

[4] Openssl. https://github.com/openssl/openssl, 2021.

[5] The php interpreter. https://github.com/php/php-src, 2021.

[6] Transformers. https://huggingface.co/transformers, 2021.

[7] Advance datasets. https://github.com/lvtao-sec/Advance, 2022.

[8] Apex implementation. https://github.com/yujokang/APEx, 2022.

[9] Aurc online. https://github.com/PeiweiHu/AURC, 2022.

[10] Common weakness enumeration. https://cwe.mitre.org/, 2022.

[11] curl. https://github.com/curl/curl, 2022.

[12] Epex implementation. https://github.com/yujokang/EPEx, 2022.

[13] Gnutls. https://www.gnutls.org/, 2022.

[14] httpd. https://github.com/apache/httpd, 2022.

[15] Klee tutorials. http://klee.github.io/tutorials/testing-regex/, 2022.

[16] libgit2. https://github.com/libgit2/libgit2, 2022.

[17] libxml2. https://github.com/GNOME/libxml2, 2022.

[18] Llvm. https://llvm.org/, 2022.

[19] mpg123. http://mpg123.org/, 2022.

[20] net-snmp. https://github.com/net-snmp/net-snmp, 2022.

[21] wllvm. https://github.com/travitch/whole-program-llvm, 2022.

[22] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. Compilers: principles, techniques, & tools. Pearson Education India, 2007.

[23] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. ACM Computing Surveys (CSUR), 51(3):1–39, 2018.

[24] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pages 242–253, 2018.

[25] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In OSDI, volume 8, pages 209–224, 2008.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[27] Lirong Fu, Shouling Ji, Kangjie Lu, Peiyu Liu, Xuhong Zhang, Yuxuan Duan, Zihui Zhang, Wenzhi Chen, and Yanjun Wu. Cpscan: Detecting bugs caused by code pruning in iot kernels. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 794–810, 2021.

[28] Alberto Goffi, Alessandra Gorla, Michael D Ernst, and Mauro Pezzè. Automatic generation of oracles for exceptional behaviors. In Proceedings of the 25th international symposium on software testing and analysis, pages 213–224, 2016.

[29] William H. Harrison. Compiler analysis of the value ranges for variables. IEEE Transactions on software engineering, (3):243–250, 1977.

[30] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In 25th {USENIX} Security Symposium ({USENIX} Security 16), pages 345–362, 2016.

[31] Yuan Kang, Baishakhi Ray, and Suman Jana. Apex: Automated inference of error specifications for c apis. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pages 472–482, 2016.

[32] Yuriy Kashnikov, Pablo de Oliveira Castro, Emmanuel Oseret, and William Jalby. Evaluating architecture and compiler design through static loop analysis. In 2013 International Conference on High Performance Computing & Simulation (HPCS), pages 535–544. IEEE, 2013.

[33] James C King. Symbolic execution and program testing. Communications of the ACM, 19(7):385–394, 1976.

[34] Chi Li, Min Zhou, Zuxing Gu, Ming Gu, and Hongyu Zhang. Ares: Inferring error specifications through static analysis. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 1174–1177. IEEE, 2019.

[35] Zhenmin Li and Yuanyuan Zhou. Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code. ACM SIGSOFT Software Engineering Notes, 30(5):306–315, 2005.

[36] Ziyang Li, Aravind Machiry, Binghong Chen, Mayur Naik, Ke Wang, and Le Song. Arbitrar: User-guided api misuse detection. In 2021 IEEE Symposium on Security and Privacy (SP), pages 1400–1415. IEEE, 2021.

[37] Dinghao Liu, Qiushi Wu, Shouling Ji, Kangjie Lu, Zhenguang Liu, Jianhai Chen, and Qinming He. Detecting missed security operations through differential checking of object-based similar paths. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 1627–1644, 2021.

[38] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In 2009 International Symposium on Code Generation and Optimization, pages 136–146. IEEE, 2009.

[39] Paul Lokuciejewski and Peter Marwedel. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. In 2009 21st Euromicro Conference on Real-Time Systems, pages 35–44. IEEE, 2009.

[40] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting missing-check bugs via semantic-and context-aware criticalness and constraints inferences. In 28th {USENIX} Security Symposium ({USENIX} Security 19), pages 1769–1786, 2019.

[41] Tao Lv, Ruishi Li, Yi Yang, Kai Chen, Xiaojing Liao, XiaoFeng Wang, Peiwei Hu, and Luyi Xing. Rtfm! automatic assumption discovery and verification derivation from library document for api misuse detection. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 1837–1852, 2020.

[42] Rahul Pandita, Kunal Taneja, Laurie Williams, and Teresa Tung. Icon: Inferring temporal constraints from natural language api descriptions. In 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 378–388. IEEE, 2016.

[43] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. Api-misuse detection driven by fine-grained api-constraint knowledge graph. In 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 461–472. IEEE, 2020.

[44] Cindy Rubio-González and Ben Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 73–80, 2010.

[45] Lingyun Situ, Linzhang Wang, Yang Liu, Bing Mao, and Xuandong Li. Vanguard: Detecting missing checks for prognosing potential vulnerabilities. In Proceedings of the Tenth Asia-Pacific Symposium on Internetware, pages 1–10, 2018.

[46] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /* icomment: Bugs or bad comments?*. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles, pages 145–158, 2007.

[47] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specification and detecting violations. In USENIX Security Symposium, pages 379–394, 2008.

[48] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. acomment: mining annotations from comments and code to detect interrupt related concurrency bugs. In 2011 33rd International Conference on Software Engineering (ICSE), pages 11–20. IEEE, 2011.

[49] Yuchi Tian and Baishakhi Ray. Automatically diagnosing and repairing error handling bugs in c. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pages 752–762, 2017.

[50] Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC), pages 53–64. IEEE, 2019.

[51] Qiushi Wu, Aditya Pakki, Navid Emamdoost, Stephen McCamant, and Kangjie Lu. Understanding and detecting disordered error handling with precise function pairing. In 30th {USENIX} Security Symposium ({USENIX} Security 21), 2021.

[52] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In 2009 IEEE/IFIP International Conference on Dependable Systems & Networks, pages 359–368. IEEE, 2009.

[53] Meng Xu, Chenxiong Qian, Kangjie Lu, Michael Backes, and Taesoo Kim. Precise and scalable detection of double-fetch bugs in os kernels. In 2018 IEEE Symposium on Security and Privacy (SP), pages 661–678. IEEE, 2018.

[54] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pages 499–510, 2013.

[55] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. Apisan: Sanitizing {API} usages through semantic cross-checking. In 25th {USENIX} Security Symposium ({USENIX} Security 16), pages 363–378, 2016.

[56] Hao Zhong and Zhendong Su. Detecting api documentation errors. In Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, pages 803–816, 2013.

[57] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. Analyzing apis documentation and code to detect directive defects. In 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pages 27–37. IEEE, 2017.

Appendix

A - Experiment Steps

We compare AURC with three bug detectors, including APEx, EPEx, and Advance, to evaluate the effectiveness of AURC, as presented in Section 5.2. In this section, we explain the details of implementing these experiments. In particular, we define the callees of all incorrect return checks that AURC reported as F.

APEx. The authors provide the implementation [8] of APEx. To use APEx, one needs to define a function list containing the names of functions for analysis and the exit functions. We collect the names of functions in F and combine them with the exit functions that APEx provided. Other steps strictly follow the guides of implementation.

EPEx. The authors provide EPEx's implementation [12]. EPEx detects error-handling bugs based on predefined error specifications. We first ran EPEx with the error specifications predefined by the authors, but it found no bugs that AURC reported because the error specifications cover few APIs. Thus, we manually construct the error specifications of functions in F according to the format EPEx defines. Note that by this step we already indicate the error specifications of incorrectly checked APIs, which benefits EPEx a lot. This is impossible if one evaluates EPEx on a new codebase. Other steps strictly follow the guides of implementation.

Advance. The authors of Advance provide datasets [7] that contain two of our codebases (OpenSSL and libxml2) instead of executable tools. Thus, for these two codebases, we check whether the documents in provided datasets correctly describe the return values of functions in F. If yes, we treat the corresponding incorrect return checks as successfully detected. This is because Advance leverages the description in documents as the oracles to detect bugs. More specifically, Advance detects defects according to the Integration Assumptions (IAs) in documents. The callees in F that have no documents or defective documents must not be detected. We define T to represent all bugs that AURC reported and N to represent bugs whose callees have no documents or defective documents. We can ensure that Advance can find at most $(T - N)$ bugs. Thus, we use $(T - N)$ as a conservative way to represent the bugs that Advance can detect on left eight codebases. Moreover, we label these results with the mark "$*$".

B - Case Studies

Case 1 - Heap-based buffer overflow (CWE-122). AURC found a potential heap buffer overflow bug in OpenSSL, as shown in Listing 5. `EC_POINT_bn2point` (line 2) decodes a curve point from the given `BIGNUM` format and is used in elliptic curve cryptography. The successful execution of it ensures the strength of the crypto. `EC_POINT_bn2point` invokes `BN_bn2binpad` (line 8) to convert the object of

BIGNUM to big-endian form but omits to check the negative return values, which indicate the execution is defective and the content in buf is unexpected. After BN_bn2binpad returns negative values, EC_POINT_bn2point continues the execution and transfers the buf to BN_bin2bn (line 20) along with the call chain EC_POINT_oct2point, ossl_ec_GF2m_simple_oct2point, and BN_bin2bn. BN_bin2bn accesses the buf, which contains random contents, in the loop until meeting the nonzero element (line 23). The len also fails to prevent breaking the bound of buf since it is not set to the length of buf. The heap buffer overflow happens.

```
1  /* crypto/ec/ec_deprecated.c */
2  EC_POINT *EC_POINT_bn2point(...) {
3      ...
4      if ((buf = OPENSSL_malloc(buf_len)) ==
       NULL) {
5          ECerr(...);
6          return NULL;
7      }
8  →  if (!BN_bn2binpad(bn, buf, buf_len)) {
9          OPENSSL_free(buf);
10         return NULL;
11     }
12     ...
13 →  if (!EC_POINT_oct2point(..., buf, ...)) {
14         ...
15     }
16     OPENSSL_free(buf);
17     return ret;
18 }
19 /* crypto/bn/bn_lib.c */
20 BIGNUM **BN_bin2bn(unsigned char *s, ...) {
21     ...
22     /* Skip leading zero's. */
23 →  for ( ; len > 0 && *s == 0; s++, len--)
24         continue;
25     ...
26 }
```

Listing 5: Example of Case 1

Case 2 - Sensitive information in resource not removed before reuse (CWE-226). AURC found the sensitive information leakage caused by incorrect return checks in OpenSSL, as shown in Listing 6. The function cipher_init (line 2) invokes EVP_EncryptInit_ex (line 4) to set up the context for encryption. In particular, the parameter key is the symmetric key, which is critical to be secret to ensure the effectiveness of encryption. However, cipher_init omits to check the negative return values of EVP_CIPHER_CTX_set_key_length (line 9) and continues execution while ctx->key_len equals to default value zero. After encryption, krb5kdf_reset (line 16) invokes OPENSSL_clear_free (line 18) to reset the symmetric key according to the key length ctx->key_len. Since it keeps the default value zero, ctx->key is not cleaned, causing the leakage of the symmetric key.

```
1  /* providers/implementations/kdfs/krb5kdf.c */
2  static int cipher_init(...) {
3      ...
4      ret = EVP_EncryptInit_ex(..., key, NULL);
5      if (!ret)
6          goto out;
7      klen = EVP_CIPHER_CTX_get_key_length(ctx);
8      if (key_len != (size_t)klen) {
9  →      ret = EVP_CIPHER_CTX_set_key_length(ctx, key_len);
10         if (!ret)
11             goto out;
12     }
13     ...
14 }
15 /* providers/implementations/kdfs/krb5kdf.c */
16 static void krb5kdf_reset(void *vctx) {
17     ...
18 →  OPENSSL_clear_free(ctx->key,ctx->key_len);
19     ...
20 }
```

Listing 6: Example of Case 2

Case 3 - Use of insufficiently random values (CWE-330). The randomness of the seed is the basis of reliable crypto. AURC found the function BN_generate_dsa_nonce (line 2), which is intended for generating a random number within the specified range for DSA and ECDSA, invokes another random number generator RAND_priv_bytes_ex incorrectly in OpenSSL, as shown in Listing 7. RAND_priv_bytes_ex (line 4) returns negative values to indicate that the execution is defective and the content within random_bytes keeps the unchanged default value instead of the random number. BN_generate_dsa_nonce fails to catch negative return values and treats random_bytes as a random number for the following generation (line 11). The use of insufficiently random seed breaks the reliability of the subsequent crypto and gives the attackers a chance to guess the secret key.

```
1  /* crypto/bn/bn_rand.c */
2  int BN_generate_dsa_nonce(...) {
3      ...
4  →  if (!RAND_priv_bytes_ex(... random_bytes))
5          goto err;
6
7      if (!EVP_DigestInit_ex(...)
8          || !EVP_DigestUpdate(...)
9          || !EVP_DigestUpdate(...)
10         || !EVP_DigestUpdate(...)
11 →      || !EVP_DigestUpdate(... random_bytes)
12         || !EVP_DigestFinal_ex(...)
13         goto err;
14     ...
15 }
```

Listing 7: Example of Case 3

Case 4 - Generation of Incorrect Security Tokens (CWE-1270). Besides, we found an incorrect return check of OBJ_obj2txt() in CMS_SignerInfo_sign() in OpenSSL. It is worth noting that OBJ_obj2txt() has no document and the majority of invocations are defective, which means both document-based and majority voting-based approaches can not detect it. AURC discovered it since we do not limit ourselves to documents and callers but also leverage the callees. Listing 8 shows the definition of CMS_SignerInfo_sign(). CMS_SignerInfo_sign() fails to handle the negative return value of OBJ_obj2txt() and continues using the invalid data in md_name to perform signature generation, which is defective. To detect this, AURC leveraged CBP and found that the return values of OBJ_obj2txt() contain -1 to indicate errors. However, RDT deduces the ranges of return values of the return check in line 3 are 0 and !0, which confuses the positive and negative numbers. The signature, as the basis of authentication, plays an important role in crypto. Detecting the generation of the insecure signature has practical significance.

```
1  int CMS_SignerInfo_sign(CMS_SignerInfo *si) {
2      char md_name[OSSL_MAX_NAME_SIZE];
3      if (!OBJ_obj2txt(md_name, ...))
4          return 0;
5      EVP_MD_CTX_reset(mctx);
6      if (EVP_DigestSignInit_ex(mctx, md_name,
       ...) <= 0)
7          goto err;
8      if (EVP_DigestSignUpdate(mctx, ...) <= 0)
9          goto err;
10     if (EVP_DigestSignFinal(mctx, ...) <= 0)
11         goto err;
12 }
```

Listing 8: Example of Case 4

Case 5 - Latent document error. During the evaluation, we found a long-hidden bug that has existed for over 20 years in OpenSSL. BIO_free() is a frequently used release function for BIO structure. With the help of path constraints in CBP, the predicted return values contain a negative range. However, the document states it *return 1 for success and 0 for failure.*, and we submitted the patch of BIO_free(), and the maintainers accept it. It is worth noting that the implementation of BIO_free() that returns negative values has existed in OpenSSL since 1999, and the inconsistent document description was added in 2000, which has been a mistake for over 20 years, showing a lack of community focus on document reliability.

Table 8: Version of Codebases

Codebase	Version/Commit Id
OpenSSL	3.0.0/1ef526
libzip	1.8.0/547d98
libwebsockets	4.3.0/c19dc9
GnuTLS	3.7.6/dbfbaa
curl	7.85.0/2481db
mpg123	1.30.2/7ca057
httpd	2.4.54/8ea5f4
libgit2	1.5.0/9286e5
libxml2	2.10.2/e2bae1
net-snmp	5.9.3/10dd27