Inference of Error Specifications and Bug Detection Using Structural Similarities

Niels Dossche and Bart Coppens, *Ghent University*

https://www.usenix.org/conference/usenixsecurity24/presentation/dossche

This paper is included in the Proceedings of the
33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the
33rd USENIX Security Symposium
is sponsored by USENIX.

Inference of Error Specifications and Bug Detection Using Structural Similarities

Niels Dossche
Ghent University

Bart Coppens
Ghent University

Abstract

Error-handling code is a crucial part of software to ensure stability and security. Failing to handle errors correctly can lead to security vulnerabilities such as DoS, privilege escalation, and data corruption. We propose a novel approach to automatically infer error specifications for system software without a priori domain knowledge, while still achieving a high recall and precision. The key insight behind our approach is that we can identify error-handling paths automatically based on structural similarities between error-handling code. We use the inferred error specification to detect three kinds of bugs: missing error checks, incorrect error checks, and error propagation bugs. Our technique uses a combination of path-sensitive, flow-sensitive and both intra-procedural and inter-procedural data-flow analysis to achieve high accuracy and great scalability. We implemented our technique in a tool called ESSS to demonstrate the effectiveness and efficiency of our approach on 7 well-tested, widely-used open-source software projects: OpenSSL, OpenSSH, PHP, zlib, libpng, freetype2, and libwebp. Our tool reported 827 potential bugs in total for all 7 projects combined. We manually categorised these 827 issues into 279 false positives and 541 true positives. Out of these 541 true positives, we sent bug reports and corresponding patches for 46 of them. All the patches were accepted and applied.

1 Introduction

Error-handling code is a crucial part of software to ensure stability and security. Developers often make mistakes in writing correct error-handling code because these paths are not always well-tested [10, 16, 35]. This can lead to serious security vulnerabilities and semantic bugs. This problem is not limited to systems software; it applies to a wide variety of applications, ranging from low-level operating system kernels to more high-level applications. Two of the top security issues of the OWASP top ten relate to missing or incorrect checks [1]. For security-critical software like Linux, about 47% of security vulnerabilities are related to missing or incorrect error checks or API usage [24, 43].

In this paper, we focus on the following specific kinds of error checking mistakes: missing checks, incorrect checks, and incorrect error value propagation bugs. As a simple example of a missing check, consider the case of memory allocations. A memory allocation may fail when there is no more memory available. Failure to handle such an error will cause a program to crash and results in a denial of service. As for incorrect checks, these occur when a check does not cover all the possible error return values. An example of an incorrect check can be found in OpenSSL: the function `RAND_bytes_ex` returns a value ≤ 0 if an error occurred [29]. We found that the prime number generator for FIPS-186-4 approved digital signature algorithms in OpenSSL incorrectly checked the return value for a value < 0 instead. Failing to handle such an error appropriately can cause a weak cryptographic secret to be used, which compromises confidentiality and integrity. Documentation for functions can be available, but often only exists for public API functions. Furthermore, even if such documentation *is* available, it often contains mistakes [34].

Error-handling mistakes are not always straightforward to find. With return values propagating over multiple nested function calls, it becomes difficult for programmers to track which values can be returned when an error occurs, even more so when the error value propagates from a deeply nested call. For example, it might not be clear that a function allocates memory in a deep path. Furthermore, a function call may have many callees that the programmers might not be aware of. These challenges make finding these kinds of mistakes hard.

Two of the most popular approaches for automatically detecting bugs in error-handling code are cross-checking and error specifications. As for cross-checking, the underlying assumption is that *most* code is correct, and that deviations from similar code patterns are bugs [6]. For example, if *most* accesses to a certain pointer are preceded by a NULL check, but there is *a minority of accesses* without a NULL check, the latter could be assumed to be bugs [6]. This works relatively well in practice, and it is an intuitive approach, which has

been successfully used in many bug-finding analysis techniques [6, 14, 20, 24, 25, 27, 41, 45–47]. However, if we were to apply such techniques on error checking code, we will only be able to find bugs for the error-returning functions that are used most often. The second approach relies on error specifications. An error specification states for each function whether it returns an error and, if it does, which return values are indicative of an error. These error specifications can then be used as an input for static analyses to find violations against these specifications [8, 13]. Unfortunately, creating error specifications manually is an error-prone and time-consuming process. Especially when considering functions with deeply nested calls, it becomes difficult to create a precise specification manually. That is why recent research proposes automatic techniques to infer error specifications from programs with as little programmer interaction as possible [2, 5, 17, 19, 40, 41]. The drawback of these approaches is that their specification's recall is low.

We aim to automatically find bugs in error-handling code, even when there are few places in the code base where incorrectly-handled functions occur, while still achieving a high recall. We also want our technique to be practical: it should not be too slow nor use too much memory. To achieve this goal, we introduce a novel technique to infer error specifications and to find violations against them. Our key insight is that error-handling code within a function exhibits *structural similarities* when compared to non-error-handling code in the same function. We leverage this characteristic to automatically build an initial error specification for the functions in the analysed program. This information is then propagated inter-procedurally through the program's call graph. The resulting error specification is then used to detect bugs in the program; or in the case of a library, the specification can be used to detect bugs in library consumers.

We implemented this technique in a tool called ESSS. Although we focus on complex code bases programmed in C, our technique is sufficiently generic to apply to a diverse range of programming languages. We evaluate our tool's capability to infer error specifications and to find bugs against those specifications using 7 well-tested and widely-used open-source software projects: OpenSSL, PHP, OpenSSH, zlib, libpng, freetype2, and libwebp. Our tool is able to find new, previously undiscovered bugs in those code bases, all while being *orders of magnitude faster* and requiring *orders of magnitude less memory* than state-of-the-art techniques to find missing error checks.

In this paper, we present the following contributions:

- **Novel technique to accurately infer error specifications:** We present a novel analysis technique that can automatically infer error specifications in complex codebases, without requiring the users to provide any additional input beyond the program's code. This approach can accurately and in a scalable way infer error specifications and leverage that information to find bugs.
- **An open-source tool to infer error specifications and detect violations thereof:** We have released ESSS as an open-source tool. This will allow developers of system software to find error-checking and error-propagation bugs in their code with our tool and possibly validate their error specification with the error specification inferred by our tool.
- **An evaluation of our technique:** We evaluate ESSS with regards to scalability (both in execution time and memory consumption); with regards to the accuracy of the error specifications; and with regards to the effectiveness in bug detection.
- **Bug fixes in various projects:** As part of our evaluation of ESSS, we have reported new, previously unknown bugs in OpenSSL, OpenSSH, and PHP and sent patches to fix them. All of our 46 submitted patches have been applied by the maintainers of these projects. One issue was assigned a CVE.

The rest of the paper is structured as follows. We present a high-level overview of our technique in Section 2. The design of our specification inference algorithm is described in Section 3, and the design of our bug detection in Section 4. We describe implementation details in Section 5. This is followed by the evaluation in Section 6 and a discussion in Section 7. We present related work in Section 8, and end with our conclusion in Section 9.

2 Overview

The main goal of our inference technique is to perform the error specification inference with a high recall, but without relying on any user input. We have two main motivations for this goal. First, we want to create a technique that is as autonomous and accessible as possible, such that anyone can use it in a plug-and-play manner. Second, relying on human input can limit the recall and precision of the error specification because for large software projects, it is likely that the programmer who enters the specification is unaware of the details of the error-returning functions.

To show the problem with existing approaches, consider Listing 1, which shows a function adapted from OpenSSL. As with many functions in OpenSSL, it returns 0 on failure and 1 on success. However, how can we deduce this? A common way to do so, is by noting that the name of `ERR_raise` (to humans) clearly indicates its involvement in error handling. This error handling occurs twice, once at line 5 for one function call and once at line 16 for two function calls. This leads us to deduce that the three called functions return 0 on error. Similarly, the check on line 9 involves comparing against the special value NULL; which again points to it being error-handling code, and leads us to conclude that `BN_CTX_new_ex` returns NULL on error. We can now deduce

```
1  int ossl_rsa_sp800_56b_check_keypair(
2      const RSA *rsa, int nbits) {
3    int ret = 0;
4    if (!rsa_check_public_exponent(rsa->e)) {
5      ERR_raise(ERR_LIB_RSA,                          A
           RSA_R_PUB_EXPONENT_OUT_OF_RANGE);
6      return 0;
7    }
8    BN_CTX *ctx = BN_CTX_new_ex(rsa->libctx);
9    if (ctx == NULL)
10     return 0;                                       B
11   if (!BN_mul(r, rsa->p, rsa->q, ctx))
12     goto err;
13   ret = rsa_check_prime_factor(rsa->p, rsa->e,
         nbits, ctx)
14         && rsa_check_crt_components(rsa, ctx);
15   if (ret != 1)
16     ERR_raise(ERR_LIB_RSA,                          C
           RSA_R_INVALID_KEYPAIR);
17 err:
18   BN_CTX_free(ctx);                                 D
19   return ret;
20 }
```

Listing 1: Motivating example roughly based on an error-returning function from OpenSSL [28].

more confidently that the shown function returns 0 on error. We now know the error return values of the function calls, and also the error return value of the shown function. The two characteristics used for reasoning about the possible error values were the *identifiers* and values with a *special meaning* by convention (e.g. NULL). These characteristics have been used many times in prior work as a basis for automatically inferring error-handling code [19, 22, 24, 25, 31, 33, 39].

However, such approaches depend on ad hoc characteristics: they depend on specific names of identifiers, which can vary significantly across projects; and they depend on values with a special meaning. So either we need to start from ad-hoc pattern matching on names and values, or we depend on developers to provide this information for us. We consider neither approach satisfactory to reach our goals. To put it another way, if we remove the identifiers from the above code, and thus can no longer see that the identifier `ERR_raise` is error-related, can we still derive error specifications correctly and with a sufficiently high precision? Our novel approach answers this question affirmatively.

Figure 1 provides an overview of our approach's pipeline. The first and crucial step is to build an initial error specification based on *path similarity matching*. In Section 2.1, we motivate the similarity matching, and give a summary of how this works. The resulting initial error specification is then iteratively propagated with different strategies, which we describe in Section 2.2. Section 2.3 shows how the final error specification can be used to detect different kinds of bugs.

2.1 Path Similarity Matching for Error Specification Inference

Our key insight is that we can match structurally similar paths, within the same function, and consider those as potential error-handling paths. The error-handling code is more similar to each other than all other code in the function. Consider again Listing 1. The branches at lines 4 and 15 are similar because they return the same value and involve a subset of the same calls, and the branches at lines 11 and 15 are similar too. All these return 0, and only the success path returns 1. Once we know that 0 indicates an error, we conclude that line 10 also returns an error.

Our analysis applies this reasoning to all functions in a program. After doing so, it obtains zero or more potential error check values for each callee. Some of these error check values may be contradictory. This happens when we would consider two branches to be structurally similar, but at least one of them was not related to error handling, or due to imprecisions of the analysis. We solve this with a simple majority vote. This results in a mapping of functions to their error values that we call the initial error specification.

Note that many functions are still missing from this error specification, which is why we call it an initial specification. For example, functions that do not contain any conditional paths will certainly not contribute to this initial error specification. If such a function uses an API that returns an error, then the error values for that callee will not be considered in the error specification. Furthermore, if the similarity matching fails for a function then it will not contribute towards the specification of its callees. In Section 3 we describe in detail how we perform this similarity matching.

2.2 Error Specification Expansion

The initial specification is incomplete because the similarity matching only has a limited call chain coverage and because some functions do not have many error-handling paths. For example, a function that has a single error-handling path may have no other similar code to match against, hence this error-handling path will not be detected. To solve these issues, the next component completes the initial specification by propagating information about functions and about error values.

As for propagating information about functions, consider again OpenSSL's error-raising function `ERR_raise`. Initially, the analysis does not know whether the program contains any error-raising functions, nor if there are any calls to such error-raising functions. However, the similarity matching algorithm *detects error-handling paths*. As such, it can count which function calls occur frequently on error-handling paths, and which do not. Our technique creates an association between functions and error-handling paths using two metrics from association analysis: support and confidence. The support measures the number of times a function appears, and

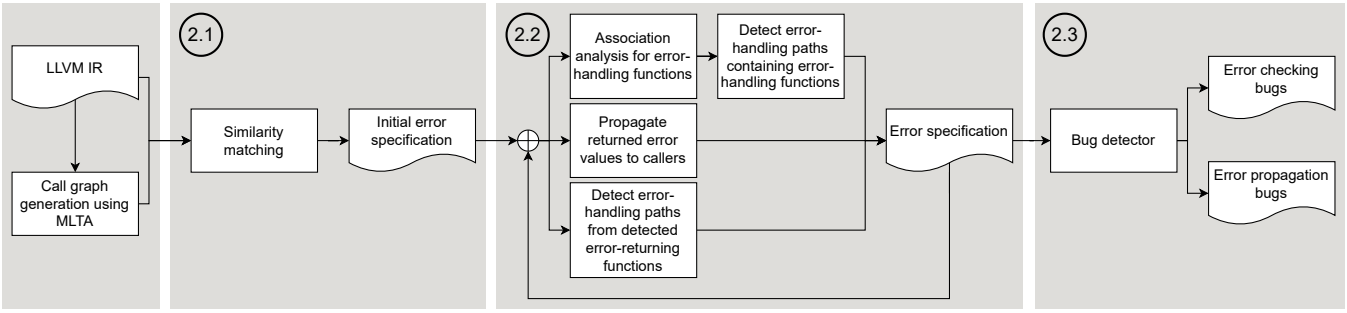

Figure 1: Overview of our inference and bug detection pipeline

confidence is the ratio of how many times it is used in error-handling paths to the total number of uses. If the support and confidence are high enough, our analysis labels the callees of such function calls as error-raising functions. This can then proceed iteratively: If a path contains a call to such an error-raising function, we can infer that such a path is also an error-handling path. This enables the discovery of more error-handling paths, even if they were previously undiscovered by the matching algorithm. Finding more error-handling paths increases the coverage of error checks in the program, which will increase the recall of the error specification.

As for propagating information about error values, we consider two related kinds of propagation. The first one is propagating returned error values to callers. For example, if we have a function F that returns a value ≤ 0 on error, and there is a wrapper function for F that returns the result as-is, then the error values are the same for the wrapper. Similarly if the wrapper function would handle the return value 0, and propagates the other return values, then the specification would be < 0. Our analysis does exactly this kind of error propagation. The second propagation of error values involves detecting error checks on calls based on the knowledge of error return values. For example, assume we inferred that `BN_CTX_new_ex` returns NULL on error. If there is a NULL check in another function G on the return value of `BN_CTX_new_ex`, then we know one of the error paths of G. Combined with other propagation techniques we can infer more information about the error return values of G.

After performing all the propagations iteratively, we obtain the final error specification that we use to detect bugs.

2.3 Bug Detection

We leverage the final error specification to find bugs against error checks and error propagation. Not all error specification violations are equally important. To reduce false positives, and reduce the number of reports that programmers do not find as important, we use an outlier-based filtering step. One approach would be to compute the ratio of incorrect checks to the total number of checks. If this ratio is below a predefined threshold, the tool reports a bug. This approach cannot find bugs for checks with a low number of occurrences. We instead use the lower bound of the Wilson score confidence interval. Essentially, this approach balances the proportions of positive and negative trials by keeping in mind small sample sizes [42]. Hence, it is less sensitive to the aforementioned problem. We describe this in Section 4.

3 Error Specification Inference

In this section we explain in more detail the challenges associated with path similarity matching for error specification inference, and our solutions to them. First, we discuss how we slice paths. Then, we discuss how we summarise those slices, and how these slices are matched. Then, we discuss how we narrow the generated error value sets to increase the precision of our subsequent bug finding analysis. Finally, we describe how we unify the potentially contradictory error value sets for a function.

3.1 Path Slicing

We want to identify potential error-handling paths by using structural similarities among different paths. It is infeasible to consider all the possible paths within a function because of path explosion. We hence limit the length and number of the paths by constructing slices of the function.

Algorithm 1 shows our slicing algorithm. Upon encountering a conditional jump, it creates slices for all jump directions (lines 17–22). The slices start at the first instruction within the branch target and end at either the last instruction within the branch target or at a conditional check, whichever comes first. A single if statement's conditional expression can contain multiple subexpressions (e.g. by using "and" conditions), each corresponding to a conditional jump. We keep track which conditional jump belongs to which slice (lines 10, 21), and will use that information in Algorithm 2. Upon encountering a conditional jump, we check whether it corresponds to the same expression that started the slice. If so, we continue

Algorithm 1: Create path slices

```
   output : jumps, comparisons, path_slices
1  fn expand_path(slice S, block B, conditional_jump J) do
2      S.append(B)
3      if B has single successor B_next then
4          expand_path(S, B_next, J)
5      else if B ends in conditional jump J' then
6          if J and J' are part of the same conditional expression then
7              foreach successor B_next of B do
8                  S' = clone S
9                  path_slices.append(S')
10                 jumps[S'] = J
11                 comparisons[S'] = expression(J')
12                 expand_path(S', B_next, J)
13 path_slices = []
14 forall function F do
15     foreach conditional_jump J in F do
16         comparison C = expression(J)
17         foreach successor_block B of J do
18             slice S = []
19             expand_path(S, B, J)
20             path_slices.append(S)
21             jumps[S] = J
22             comparisons[S] = C
```

Algorithm 2: Generate initial error specification

```
   input  : jumps, comparisons, path_slices
   output : The initial specification initial_spec
   // Abstract Representation Using Summaries § 3.2.1
1  slices = {}
2  foreach slice S in path_slices do
3      summaries[S] = []
4      foreach basic_block B in S do
5          summaries[S].extend(summarise(B))
6      slices[summaries[S]] = S
7      resolve_values(summaries[S])
   // Similarity Matching § 3.2.2
8  best_match = {}
9  foreach unique slice pair (S1, S2) in (summaries, summaries) do
10     if jumps[slices[S1]] = jumps[slices[S2]] then
11         continue // ignore the same conditional jump
12     if len(LCS(S1, S2)) != min(len(S1), len(S2)) then
13         continue
14     foreach slice S in [S1, S2] do
15         comparison C = comparisons[slices[S]]
16         best = best_match[C]
17         if no best or len(best) < len(S) or (len(best) = len(S) and
             #branches(best) > #branches(S)) then
18             best_match[C] = S
   // Narrowing § 3.3 and unification § 3.4
19 foreach (comparison C, summary S) in best_match do
20     function F = function_involved_in(C)
21     set V = checked_values_of(C)
22     error_set E = V ∩ return_value_set(F)
23     counts[F][E]++
24 foreach (_, set_counts) in counts do
25     foreach unique slice pair (S1, S2) in (set_counts, set_counts) do
26         if S1 ⊂ S2 then
27             set_counts[S2] += set_counts[S1]
28         else if S2 ⊂ S1 then
29             set_counts[S1] += set_counts[S2]
30 initial_spec = {}
31 foreach (function F, error_set E) in counts do
32     if confidence(E) ≥ 0.5 then
33         initial_spec[F] ∪= E
```

extending the current slice, otherwise we stop (line 6). This differs from existing approaches that start slices at conditional checks and end at the return instruction [24, 25, 31]. Those approaches result in overly long slices.

3.2 Similarity Matching

Algorithm 2 shows how we deduce the initial specification. It consists of four parts. The first two parts are for matching similar code which we will discuss in this section. First, it abstractly represents code to ease comparing between similar code. Then, that representation is used to match similarities.

3.2.1 Abstract Representation Using Summaries

Some analyses detect similar code using Natural Language Processing (NLP) techniques, such as bag of words [3, 45, 46]. This works reasonably well, but it can be easily influenced by the syntax of the language instead of structure and semantics of the code. Similarity matching techniques that only associate pairs of conditions and error-raising functions are too restricting [14, 41]. This is because they use called functions as the main characteristic, but not all error-handling paths call functions. Rather, we propose a light-weight similarity matching approach that captures the code's structure and semantics.

Our approach is loosely inspired by Bai et al.'s function summaries [4]. The idea is that summaries serve as an abstract representation of an ordered list of interesting instructions. Bai et al. used the idea to create a list of locking function calls per function. We instead create summaries per slice containing instructions that are related to error handling: stores, returns, calls, and conditional branch instructions. For stores and returns we also incorporate their used values. We first create a summary per basic block, and then combine those into path summaries (lines 2–5). Our slices do not contain unconditional jumps. Instead, as a consequence of slice construction, they contain the instructions at the jump's target. This allows matching longer sequences (e.g. with code such as `goto out`), which improves the specification's precision.

Constants and pointers play an important role in our summaries. In particular, return values are important because they are often used to propagate error values. Developers often store the return value in a variable and use that variable later in the return statement. In LLVM's Intermediate Representation (IR), this will result in some store instructions, followed by a load and a return instruction at the return sites. To accurately summarise the return instruction, we use a path-sensitive and

flow-sensitive backwards data-flow analysis to try to resolve the values to a constant that can be returned if the slice is executed (line 7). We use LLVM's basic alias analysis to find potential store targets for corresponding loads. The scope of our data-flow analysis is limited to the slice the summary is from, with one small exception. If there is only one unique path towards the slice, then we also take into account the unique predecessors of this slice. This allows resolving more path-sensitive values because they may contain store instructions to pointers or variables that will be used in the slice.

Listing 1 contains many slices. We focus on four of them annotated with A–D. Slice A starts just after the first conditional check on line 4, contains a call to `ERR_raise`, and ends at the return instruction on line 6, for which the summary contains its return value 0. The other slices are at lines 10 (B), 16–19 (C), 18–19 (D, because of the `goto` at line 12).

3.2.2 Matching Summaries

The problem of matching our path summaries is related to the problem of matching the longest common subsequence (LCS) between two datasets. The second part of Algorithm 2 shows how we perform a pair-wise loop over all the summaries to compute the length of the longest common subsequence. We break ties by favouring paths with fewer branching points. We mark the matches as error-handling slices.

We define a match between the operations if they have the same type and the same resolved values. A matching operation adds a score of 1 and a mismatch adds a score of 0. If a value is not resolved, then we will match the raw IR value instead, i.e. without taking into account the path condition. This may miss some matches that would be equivalent in execution on certain paths.

To illustrate this summary matching, consider again the example in Listing 1. Note that we take into account that `ret` has the value 0 for these four slices. Slices A and D are of length 2, B is of length 1, and C is of length 3. We can clearly see that slice D is a subsequence of slice C and so we mark both as error-handling slices. We also see that A is a subsequence of C, and B is a subsequence of all others, and so we mark A and B too as error-handling slices. As shown here, matching subsequences has two advantages. First, error-handling code tends to increase in length deeper down in functions because when errors occur later in execution, the code needs to clean up more resources. Second, the matching instructions are not always adjacent, such as with the `BN_CTX_free` separating the return and `ERR_raise` call. There are other slices as well, but they are not error-handling slices and are therefore left out of this example.

3.3 Narrowing Error Value Sets

While other tools often represent error value sets in a way that is limited to simple predicates like ≤ 0, < 0, $\neq 0$, etc. [5, 19], we chose a more precise representation. We represent the possible error values for a function as the union of disjoint integer intervals. This still allows for representing the simple predicates, while also enabling us to precisely model the error value set. Representing the error value set in a more versatile and precise way helps to reduce the number of false positives.

An example where our representation helps, is PHP's `zend_get_property_info` function. For this function, the values 0 and `UINTPTR_MAX` have an error-related meaning. If we were to rely on simple predicates, then it is not possible to represent such sets precisely. A simple predicate will vastly overapproximate the error values and cause false positives.

However, even when using the precise error value sets, it can lead to false positive bug detections. Consider for example the function `open` from libc. If all callers of `open` use a check condition < 0, then the error specification inferred for open will be $(-\infty, -1]$. If we now add another caller of `open` with a check condition $== -1$, then the error specification will remain the same, but our bug detection would flag that check as incorrect. This happens because our tool expected a check of the form < 0 instead. There is clearly a lack of precision in this specification.

If we were to look at `open`'s *implementation*, we observe that only values in $[-1, +\infty)$ can be returned. Upon intersecting that with $(-\infty, -1]$, derived from the inference, we get the desired set $[-1, -1]$. We call this narrowing, and we use it to improve the precision of the specifications. To implement this, we perform an inter-procedural, depth-limited value set analysis on the return value of each function. The depth refers to how many edges are followed in the call graph. If the value set could not be determined, or the depth limit was reached before it could be determined, then the set will be $(-\infty, +\infty)$. We take the intersection of this value set with the set inferred from the error specification. This either results in a more precise set or the same set.

One additional insight we use to narrow the error value set is that error values are usually nominal values instead of ordinal values. For example, it does not make sense to perform arithmetic with two error values, e.g. adding `EINVAL` and `EPERM` is meaningless. We can thus further improve precision by discarding values that are (transitively) used in arithmetic instructions. Although this assumption is unsound, we have not observed that this introduced false positives or negatives.

Narrowing is not limited to libc, but is applied to every module. For libc, we compile the musl C library to LLVM IR and link it statically with our programs under test. This approach works even if the program does not support static linking because the call graph is created at analysis time.

Lines 19–23 of Algorithm 2 compute which error sets are candidates for consideration (lines 19–23). Every comparison of a return value will involve a function F, gathered using **function_involved_in**. The error value set corresponding to the comparison is stored in V. The narrowing as described above happens at line 22, where we take the intersection of the

checked values V with the values that can actually be returned by F. Finally, we count how many times each candidate error set occurs for each function.

3.4 Unifying Results Into Error Value Sets

As a function F can have many call sites, the checks derived from them can potentially be conflicting. Some error value sets of F might even be *subsets* of one another, e.g. one condition might use $\neq 0$ while the other uses < 0. There are three possible causes for this. First, it is possible that both checks are correct. For example, functions might return 0 on success and -1 on error (e.g. `stat`), making both conditions valid. The second major cause are simply bugs: the code base in which we are trying to find bugs is the same from which we derive the error specification. A third minor cause is possible mistakes by the similarity matching algorithm due to matching code that is not error-related. To prevent the correctness being influenced by these issues, we need to unify the results such that there is only one error value set per function. We previously counted how many times a value set occurs per function (line 23). We *unify* subsets by first summing the tally of the subset to the superset (lines 24–29). For example, if < 0 has a tally of x and $\neq 0$ has a tally of y, the result after unification is a tally of $x + y$ for $\neq 0$. The tally of < 0 remains unchanged. Then, we *reject* a set if its confidence is below 50% because we assume that most code is correct and we want to avoid conflicts, leaving only a single set per function.

4 Bug Detection

In this section we describe how we use our error specification to find bugs in error checks and error propagation. As for the bugs in error checks, we first describe the main approach to detect incorrect and missing check bugs. Then we describe how we filter false positives using a simple heuristic. Finally, we describe how we detect error propagation bugs.

4.1 Detecting Missing and Incorrect Checks

Our main goal is to use the inferred error specifications to detect missing error checks and incorrect error checks.

The first step in our bug detection analysis is to determine which error values are checked for each call to an error-returning function. Note that a return does not need to be checked directly; it can also be stored in a variable and be checked later. It is thus crucial to track the checked values per call instead of per branch condition. Note that a function call may have multiple callees. Indirect calls are particularly interesting because programmers do not always realise what the possible call targets are, making it easy to forget about a specific error value.

The second step is to use the set of checked values to detect bugs, and filter away likely false positives. There can be functions in the program of which the return value is (almost) never checked, even though they can return an error. It is possible to filter such cases using an outlier-based analysis. Typically, outlier-based analyses count how many times a function is checked (correctly) and how many times it is not. In particular, for error-handling code, we consider an error-returning function to be checked correctly when the set of checked error values is a non-strict superset of the set of possible error values returned by that function. We additionally need to normalise the counts for indirect calls with many potential call targets as a single call could otherwise increase the counts of many functions. Note that missing checks are a special case of incorrect checks: the checked error values for a missing check is the empty set.

One approach is to check whether the ratio of correct checks to the total number is above a certain threshold. If it is, then a bug is reported, otherwise it is filtered. A significant downside of this approach is that cases with a low total count will almost never be reported as a bug because the ratio will be too low. This approach risks filtering away too many true bugs. Even worse, a program often contains API functions that are used only a few times. If, for example, the checks on an infrequently-used function are incorrect at *every call site*, most outlier-based filtering would always filter such cases away, leading to significant false negatives. We want to be able to detect bugs even for functions that are used few times, even if they are always checked incorrectly, while still filtering away cases programmers do not care about.

Informally speaking, we want a filtering technique that does *not* filter away checks for functions that do not have a lot of occurrences. It might be counter-intuitive, but it allows finding rare cases like the one described above, where there is a single incorrect check and no correct ones. Nevertheless, we *do* want to filter away cases where the number of incorrect occurrences becomes sufficiently higher in proportion to the number of correct occurrences. Hence, rather than using traditional regular majority voting, we use the lower bound of the Wilson score confidence interval. This takes into account not only the relative difference between positive and negative occurrences, but also their absolute difference [26, 42]. Importantly, this allows us to differentiate between a code pattern that has 0 positive occurrences and 1 negative occurrence, and a code pattern with 0 positive occurrences and 100 negative occurrences at the *same* threshold. This is impossible with regular outlier-based techniques.

4.2 Heuristics

A recurring issue in prior work is that the techniques incorrectly assign an error specification to some functions that do not per se return errors. Typical examples include comparator-related functions like `strcmp` [5, 47]. We encountered the same issue during evaluation. To solve this, we observe that these kinds of functions are pure (i.e. functions without side-

```
1  zend_result zend_update_static_property(...) {
2  -  bool retval =
       zend_update_static_property_ex(...);
3  +  zend_result retval =
       zend_update_static_property_ex(...);
4    return retval;
5  }
```

Listing 2: An error propagation bug in PHP [32] found by our analysis tool with the diff showing the fix.

effects). The hypothetical failure of such functions is only dependent on the input that was already known. Based on this observation, our heuristic then discards pure functions from the set of error-returning functions. LLVM already tracks which functions are pure using function attributes, so we already have the necessary information. This got rid of all such cases from the reported issues, but of course also results in a slight decrease in the absolute number of true positives.

4.3 Error Propagation Checkers

The error specifications can furthermore not only be used to detect missing and incorrect checks, but can also be leveraged to detect bugs against error propagation. We implemented such an additional checker to detect accidental signedness conversions during error propagation. We illustrate this with the example in Listing 2, which shows such a bug we found with our tool in PHP. In this example, it is important to note that `zend_result` is a signed enum type where 0 indicates success and -1 indicates failure. The function `zend_update_static_property_ex` returns a `zend_result`, but its return value is instead assigned to a `bool`. On the platform we target, `bool` is an unsigned type. The consequence is that if a value of -1 were to be returned, the assignment to the `bool` converts the value to 1. Since `zend_result` is a wider type than `bool` the function `zend_update_static_property` would then return 1 on failure, instead of -1.

To find such issues, we leverage type information in addition to return values. To detect this particular issue, we create four bins for each return type: ≤ 0, $== 0$, ≥ 0, and a fallback bin. We count how many times the error values belong to a bin. If there is a bin with a high enough support and confidence, then we assume that bin indicates the sign of the error values. Once we know the sign associated with the type, we report a bug when an assignment happens to a type that has a different associated sign than the type of the original value.

5 Implementation

In this section, we discuss the implementation of our technique, which we call *ESSS* (Error Specification through Structural Similarities): a tool to infer error specifications from programs and find violations thereof. Our static analysis is based on the LLVM 14 compiler framework [18], extended with Multi-Level Type Analysis (MLTA) [23] to determine indirect call targets, which increases the precision of the interprocedural CFG and call graph[1]. In theory, our implementation is therefore compatible with every language that compiles to LLVM IR. Although we implemented multithreading for almost all the steps in our analysis pass, it was not necessary to utilise this functionality during the evaluation because the tool was already fast enough for our purposes. Only the LLVM bitcode module loading uses parallel processing.

6 Evaluation

In this section, we empirically evaluate our tool's ability to automatically infer error specifications and to detect bugs. Our goal is to answer the following four research questions:

- How well does our technique scale regarding run time and memory usage?
- How well does the error specification inference algorithm work? We compare our approach to the current state of the art for error specification inference: EESI [5].
- How effective is our tool at finding new bugs?
- What is the false negative rate, and how does that compare with the state-of-the-art tools CodeQL [7] and APISan [47]?

First, we describe our experimental setup. We then answer each of our research questions separately, and we end with threats to validity.

6.1 Datasets and Experimental Setup

To evaluate the error specification inference and the effectiveness of detecting new bugs, we use the following 7 open-source software projects: OpenSSL (commit 8d927e55), OpenSSH (commit 36c6c3ef), PHP (commit abc41c2e), zlib (commit 12b345c4), libpng (commit e519af8b), freetype2 (commit bd6208b7), and libwebp (commit 233960a0). The commits are from the development branch at the time of starting the evaluation. We used the default configuration that Debian uses while compiling these packages. Unless mentioned otherwise, we ran our experiments on an Intel Core i7-5930K with 48GiB of DDR4 RAM at 2666MHz.

To evaluate the effectiveness regarding false negatives, we use the APIMU4C dataset [9]. This dataset contains various API misuse bugs, including error check bugs, from past versions of OpenSSL, curl, and httpd.

[1] We extended MLTA with some small bugfixes, but these are out of scope for discussion in this paper. Since we have open-sourced our code, this includes a version of MLTA with our modifications.

Project	EESI		ESSS		SLOC
	Time	Memory	Time	Memory	
OpenSSL	176.79	20.46	3.22	0.75	542K
OpenSSH	81.70	4.02	1.44	0.22	120K
PHP	-	-	14.50	1.85	1.46M
zlib	4.09	0.25	0.38	0.05	30K
libpng	21.44	4.50	0.49	0.09	63K
freetype2	91.66	10.56	0.94	0.19	141K
libwebp	12.65	2.20	0.40	0.14	75K

Table 1: Run-time performance and memory usage comparison. Time in seconds, memory in GiB.

As our approach infers error specifications, we evaluate the accuracy of the inferred specifications. As EESI consistently outperforms other error-specification-based tools such as APEx [17] and Ares [19], we only compare our tool to EESI. To evaluate false negatives, we compare ourselves with tools that do not depend on error specifications: APISan and CodeQL. For CodeQL, we use the queries MissingNullTest, ReturnValueIgnored, InconsistentNullnessTesting, InconsistentCheckReturnNull, MissingNegativityTest.

6.2 Scalability

A static analysis tool that aids developers, should be *usable* by developers. To maximize the usability of ESSS, its run-time performance and memory usage should allow developers to run it on their own machine.

First, we compare the scalability against EESI. Table 1 compares the run time and peak memory usage of EESI with our solution, ESSS. We ran both of them ten times and averaged the results for both the run time and the memory usage. EESI ran out of memory on PHP, even on a machine with 128GiB of memory. The table rows for EESI will hence contain a dash, indicating the absence of results. Note that the measurements on EESI only include the time and memory needed for the error specification inference, whereas the measurements for ESSS include both error specification inference and bug finding. Despite this, the data clearly show that our tool is consistently faster and uses less memory. Even for large projects like PHP, ESSS can analyse tens of thousands of functions in under 15 seconds, while EESI is not able to finish its analysis.

Next, we compare ESSS's scalability on the APIMU4C dataset against CodeQL and APISan in Table 4. We had to evaluate APISan on an AMD Threadripper 2990WX with 64GiB of DDR4 at 2934MHz, as the 48GiB of memory in our normal benchmarking machine was insufficient for APISan's database creation step. For CodeQL, we report query time, excluding query compilation and database creation. For APISan, we report checker time, excluding database creation. It is clear that even when those times are not included for CodeQL and APISan, ESSS significantly outperforms them.

6.3 Error Specification Inference

To evaluate the effectiveness of our technique regarding the generated error specifications, we use three metrics for evaluating the specifications: recall, precision, and F1-score. We express all three metrics as percentages. Recall measures how many of the expected specifications were found by the tool. Precision measures how many of the inferred specifications were correct. We value precision and recall as equally important and thus additionally chose the F1-metric to combine both metrics into one metric. We again compare ourselves against EESI.

Because it is infeasible to manually analyse every generated error specification, we randomly sampled the results to determine the precision, recall and F1-score. Recall is about our expectations of the generated output, so we randomly selected 750 of all non-void functions from each project. We only kept the functions that returned an error using manual review. This resulted in 159 functions for OpenSSL, 151 for OpenSSH, 312 for PHP, 70 for zlib, 76 for libpng, 154 for freetype2, and 219 for libwebp. Precision is about how accurate the specifications are, so we sampled twice: once to evaluate EESI and once for our tool. The sample size used to compute the precision is the same as for computing the recall. However, for some projects both EESI and ESSS did not generate enough specifications to obtain the same sample size. In those cases, we included every generated specification for computing the precision of that tool on that project.

It is essential to precisely define when we consider an error specification for a function to be correct. For example, `BIO_accept` from OpenSSL returns -1 on error. EESI infers the error specification as the simple predicate < 0, while our tool inferred $[-1,-1]$. If this result is checked with $== -1$, then EESI would report a false positive bug because not all error values are covered, while our tool would (correctly) not report a bug. This leads us to the following two requirements to consider an error specification correct. First, the value set must include every error value, while not including any success value. Second, simple predicates are only correct when they cannot cause false positive reports in the code base, e.g. < 0 is correct because there are no checks of the form $== -1$.

Table 2 shows the results. ESSS consistently infers more error specifications than EESI, which consequently results in a higher recall. The main limitation of EESI is that its propagation is limited to the initial domain knowledge. The quality and completeness of the initial knowledge pose a hard limit on the recall and precision of the technique, whereas our tool does not need domain knowledge, and automatically infers everything. Furthermore, Table 2 shows that our propagation is more precise, as we consistently get a higher precision than EESI, although that difference is less dramatic than the difference in recall. The number of disjoint specifications refers to error value sets that are the union of two or more disjoint intervals that cannot be represented correctly by EESI. These

Project	EESI				ESSS				
	# Specs	Precision	Recall	F1-score	# Specs	# Disjoint specs	Precision	Recall	F1-score
OpenSSL	4655	83.54%	62.89%	71.76%	7853	50	92.54%	96.86%	94.60%
OpenSSH	550	75.50%	50.99%	60.87%	1655	56	91.39%	87.42%	89.01%
PHP	-	-	-	-	4417	87	83.06%	80.13%	81.57%
zlib	33	84.85%	44.29%	58.20%	42	2	95.24%	58.47%	72.53%
libpng	32	84.38%	39.62%	53.92%	57	0	96.15%	75.47%	84.57%
freetype2	20	30.00%	5.19%	8.85%	426	12	84.97%	72.08%	77.99%
libwebp	135	59.26%	47.49%	52.73%	219	4	89.50%	75.80%	82.80%

Table 2: Comparison of the number of specifications, precision, recall and F1-score between EESI and our tool ESSS.

are a minority of the cases for all analysed projects. The main benefit is that this avoids false positives like `BIO_accept`.

The numbers obtained here for the precision and recall metrics of EESI are lower than in the original EESI paper, because EESI's authors limited themselves to only public-facing API functions, and excluded internal functions. However, bugs do not exclusively occur in the handling of public-facing API functions, and thus we consider *all* functions. The recall is generally quite high for ESSS, except for zlib, where the recall for both ESSS and EESI is poor. The low recall is caused by zlib not having a specific error-logging or typical error-raising function. Furthermore, zlib usually does not do much clean-up, so there is not much code to match.

We set the required threshold to associate error-raising functions with error-handling paths at 95%. It is interesting to look at which functions are inferred to be typical error-raising functions with a confidence $\geq 95\%$. For OpenSSL, this set includes the `ERR_new`, `ERR_set_debug`, and `ERR_set_error` functions, which the `ERR_raise` macro we showed in Section 2 uses. We also see functions whose name contain keywords that are typically error-related (e.g. free, fatal, end).

6.4 Effectiveness in New Bug Detection

We evaluated our tool ESSS at a bug-finding threshold of 72.5%. This threshold is low and could result in many false positive cases. However, we chose our threshold such that the number of bug reports was still manageable to evaluate by hand, and also such that we can evaluate the effect of increasing the threshold on the false positive rate.

Table 3 gives the number of detected bugs by ESSS. The table specifies how many bugs were reported for each category. We break down the bugs into three kinds of bugs: missing checks (MCs), incorrect checks (ICs) and propagation bugs (PBs). Furthermore, we classify the reported bugs for each kind into three categories: probable true positives, probable false positives, and unknown cases. We denote these with 'probable' to stress that we had no access to a ground truth and therefore categorised them ourselves with manual effort. Unknown cases happen where we were unconfident to determine to which category they belong. The false positive rate

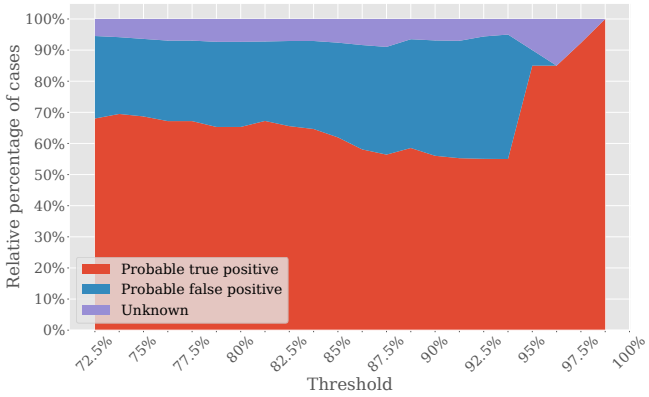

Figure 2: PHP bug relative category breakdown.

does not take into account the unknown cases.

The false positive rates for the missing checks are in general higher than those for the incorrect checks. We think this is because, for the incorrect checks, we know these checks already occur in a context where the function can fail, whereas for the missing checks we do not know this as our tool is context-insensitive. The total number of true positives is 541 and the total number of false positives is 279. Thus the total averaged false positive rate, over all projects, is 34.02%. Depending on if we count the unknown cases conservatively as false positives or count them as true positives, our false positive lies in the range of 31.42% to 39.08%. We found during our evaluation that many false positives are easily pruned in a matter of seconds because many originate from the same error-returning function. For example, once we learned about the particular false positive patterns for OpenSSL, we were able to process each such false positive case in < 10 seconds.

PHP was the only project where propagation bugs occurred. We believe this is because the other projects are (partially) libraries, which typically have a standardised error-propagation style. Propagation bugs are likely more common in non-library code. The propagation bug from Listing 2 was confirmed and fixed. There was one other true positive bug, and

Project	Reports	MC	IC	PB	Unknown	FPR MC	FPR IC	FPR total
OpenSSL	366	238	84	0	44	46.22%	39.29%	44.41%
OpenSSH	61	49	11	0	1	16.33%	54.55%	23.33%
PHP	355	303	29	3	20	26.40%	44.83%	28.01%
zlib	4	4	0	0	0	25.00%	-	25.00%
libpng	1	1	0	0	0	0.00%	-	0.00%
freetype2	25	20	4	0	1	0.00%	50.00%	8.33%
libwebp	15	13	0	0	2	18.18%	-	18.18%

Table 3: Breakdown of the statistics of detected bugs by ESSS. FPR = False Positive Rate, IC = Incorrect Checks, MC = Missing Checks, PB = Propagation Bugs.

the remaining bug was a false positive.

As the results of ESSS depend on the chosen threshold, we evaluated the impact of the threshold on the false positive rate. Figure 2 shows the results for PHP. We chose this project because it had the highest number of reports after filtering unknown cases, and is therefore excellent candidate to study more in depth. We see a steady increase in the false positive rate followed by a substantial decrease. This is caused by the `zend_hash_add` function. It has a score of 94.20% and causes 30 of the 75 false positive missing checks. All of these false positives are filtered when the threshold reaches 95%. Again, note that while this single function has a large impact on the false positive rate, a developer could relatively easily filter such results manually, as all 30 originate from the same error-returning function. Finally, although these graphs show that higher thresholds result in a lower false positive rate, they also lower the number of reported bugs, thus increasing the number of false negatives.

For OpenSSL, functions like `ASN1_item_ex_i2d` cause many false positives because they can be used in two ways: either as a validation function, or as a function that writes the data to the buffer. Once the programmer learns about this pattern, it is easy for them to recognise and ignore these cases.

We submitted 16 fixes for OpenSSL, 1 fix for OpenSSH, and 29 fixes for PHP; one was assigned a CVE. We only fixed bugs that seemed harmful and easy enough to fix ourselves. All these fixes were accepted. Appendix Table 6 contains a breakdown of the reported bugs and their impact.

It is also worth noting that OpenSSL is regularly checked with the Coverity static analysis tool, which can also find some kinds of error-checking mistakes. OpenSSL's release policy even states that all open Coverity issues must be handled before releasing a new version [30]. Despite that, our tool managed to find incorrect error-checking code in OpenSSL.

6.5 False Negative Evaluation

We manually inspected the APIMU4C bugs for the three projects, and removed those that were not related to error handling bugs. We again evaluated at a threshold of 72.5% for ESSS. Table 4 lists the results of our experiment. Our tool

Project	CodeQL			APISan			ESSS			Total
	#	M	T	#	M	T	#	M	T	#
OpenSSL	10	6.86	144.34s	5	25.24	3h 32m 53s	16	0.26	3.43s	42
curl	2	1.47	33.33s	0	1.46	2m 58s	1	0.13	1.27s	18
httpd	0	3.20	57.03s	2	2.15	13m 39s	2	0.23	1.28s	12

Table 4: Comparison between different tools of the number of found cases (#), the peak memory usage in GiB (M), and time spent analysing (T). Evaluated on APIMU4C.

ESSS consistently uses orders of magnitude less memory and time, yet still finds more cases overall, indicating our tool has a lower false negative rate while having better performance. Most of the false negatives for ESSS are caused by return values that escape to fields; filtering these would require extensive escape and inter-procedural data-flow analysis. CodeQL finds two issues (missing checks on malloc) for curl whereas ESSS only finds one (missing check for an internal function). ESSS did not find the malloc issues, despite having inferred the specification for malloc, because their return values escape to a field. CodeQL on the other hand is overly eager in reporting these kinds of issues; it will always report missing checks even if the value is checked at the field's use-site.

We also tried to run FICS [3] on these programs. It was only able to find a single bug in OpenSSL. No bugs were reported for the other programs.

APIMU4C also contains single-file tests, each consisting of around 100 lines of C. These isolated cases test for common C library function misuses. However, due to the tiny amount of code in these tests, any self-learning system without domain knowledge does not have enough code to actually learn from this. As such, we did not consider these single-file tests and preferred to evaluate on real-world code.

We also checked whether CodeQL and APISan could detect the bugs from Table 6, i.e. those we reported and patched. Table 5 lists the results. We found that they were unable to find most bugs ESSS could find. We ran these benchmarks the same way as before, except we ran OpenSSL with a single thread because APISan kept deadlocking. We did not encounter this issue with APIMU4C because that OpenSSL version is older. To estimate what the time with multithreading could be, we ran PHP both with multithreading enabled and disabled. With multithreading disabled, we obtained a time of around 13h 7m 56s. If we expect the same speed-up for OpenSSL as for PHP then the run time should be 1h 11m 53s.

We also tried to run FICS on these programs. However, it crashed on both our versions of PHP and OpenSSL. For OpenSSH it did not find our reported bug, after running for 1h 35m.

6.6 Threats to Validity

There was no ground truth data set available for the error specifications of the evaluated applications. We had to manually

Project	CodeQL			APISan			ESSS			Total
	#	M	T	#	M	T	#	M	T	#
OpenSSL	1	6.70	136.93s	2	56.99	15h 41m 42s	16	0.75	3.22s	42
OpenSSH	0	1.22	46.24s	0	5.04	16m 55s	1	0.22	1.44s	18
PHP	0	8.30	450.87s	0	15.38	1h 9s	29	1.85	14.50s	12

Table 5: Comparison between different tools of the fixed bugs from Table 6. Same legend as Table 4.

create such a data set to compute the recall and precision of the inferred error specifications. Similarly, there is no ground truth for the *new* bugs our tool detected, because they were unknown at the time we found them. All 46 cases where patches were accepted by the maintainers are, of course, truly confirmed bugs. Since we evaluated our tool on a limited set of projects, it is possible that our technique does not generalise well to code bases with different error propagation.

In our comparison to EESI, the threat to validity from EESI's own evaluation also applies here: the quality of the error specification depends upon the initial domain knowledge we gave to EESI. Not all projects that we analysed were also analysed by the authors of EESI, further compounding that risk. We want to emphasise that this threat only applies to EESI, as our own tool does not depend on domain knowledge: we are immune to this threat. The details about the domain knowledge given to EESI are described in Appendix C.

7 Discussion

We now discuss some broader aspects of our work. We describe how error inference and bug detection can be split between libraries and their consumers. We also discuss future work and describe the ethical considerations.

We discuss the generality of our approach in Appendix A.

7.1 Splitting Inference and Bug Detection

So far, we applied our technique to detect bugs in the same software from which we inferred error specifications. However, the inference and the bug detection components can operate separately. Hence, it is *also* possible to detect bugs in consumers of libraries. In that case, one would first create an error specification for the library, and then use that as input with a library consumer. We tested this approach with OpenSSL as a library and PHP as a consumer. We used this to find two extra true positive bugs in PHP. The specification of libraries can be further refined by combining results from multiple library consumers. However, library consumers typically only use the public API of a library, so although this may improve the precision of the public API specification, this will decrease the precision of the internal API's specification.

7.2 Future Work

We currently create the summaries in an intra-procedural way. Operations from called functions are not included in the summary of the caller. Instead, they are represented with a call instruction. We believe that "inlining" summaries of callees may improve the precision of the summaries, at a small cost of reducing analysis run-time performance.

Not all applications use the return value of a function to check for errors. Some may use an additional global error state (e.g. the exception mechanism in PHP). Our tool is unaware of this, and will report a false positive when the global state is checked instead of the return value. Another propagation method is when a function uses a pointer, passed as an argument, to write an error code to. The similarity matching already takes into account store instructions to pointers, but our bug detection is limited to checking the return values of functions. Our current implementation will therefore miss bugs in code that use an error propagation mechanism other than using return values. We leave these cases for future work.

For OpenSSL we discovered that some functions serve a dual purpose in Section 6.4. The first call may do validation and the second call may do the actual work. This means that a false positive report will occur on the second call, despite there being a check on the first call. We leave learning this relation for future work. This would greatly reduce the number of false positives for OpenSSL.

7.3 Ethical Considerations

There are two ethical considerations: our tool reports potential security bugs, and we interact with maintainers. First, our tool does not generate exploits. Second, we clearly label findings as from an experimental tool, and only submit patches for probable true positives.

8 Related Work

This section discusses the related work, primarily focusing on static analyses to detect bugs in error-handling or error-checking code. We have divided the related work into four different categories: techniques that try to match conventional patterns, techniques that depend on error specifications, similarity-based techniques, and other static techniques. It is also often possible to extend static techniques with a dynamic analysis such as fuzzing [11, 15, 16]. However, we consider them out of scope as our focus lies on static analysis.

8.1 Conventional Patterns

For some software, such as the Linux kernel, error return values for many functions follow a standard convention (e.g. `ENOMEM`, `EINVAL`, etc.). These error codes have been used to detect error-handling code and data mine rules from the

involved functions [22, 24, 25, 31, 33, 37, 39]. However, even though most kernel functions in the public API follow this convention, not all internal kernel functions do. There are kernel functions that return a boolean value to indicate success or failure, or even use custom error values. Furthermore, error-handling code is not standardised for user applications. These analysis tools will therefore be unable to find error-handling code that does not follow a specific convention.

Other tools use approaches from data mining to automatically deduce these kinds of patterns. PeX uses association analysis to determine the relation between permission checks and actions [48]. Nguyen et al. mine preconditions of APIs based on strong type information [27]. EH-Miner uses cross-checking on checked functions and their conditional actions to determine likely error-returning functions [14]. More generic bug-finding tools include PR-Miner to find common patterns using association analysis in large software [20]. However, these tools either have scalability issues, do not specifically detect error checks, or rely only on cross-checking, which makes them unsuitable for functions with few uses.

8.2 Error-specifications

DeFreez et al. proposed EESI, a technique that uses initial domain knowledge provided by the programmer to build an error specification [5]. EESI propagates this initial domain knowledge inter-procedurally through the functions in the program [5]. The downside of this technique is that the precision and recall of the generated error specification are limited by the precision and completeness of the initial domain knowledge. For example, the authors often include the fact `malloc` returns a NULL pointer in their initial domain knowledge as the only function that returns an error. Hence, every other error value that does not transitively originate from a memory allocation failure will not be detected by the tool and thus not be included in the error specification. This reduces the recall of the error specification. Furthermore, their technique can only represent error values as simple predicates such as < 0, > 0, etc. IMChecker uses a domain-specific language to specify common API usage and uses symbolic execution to find violations thereof [8].

Ares is another static analysis tool that uses heuristics specific to C code to identify error-handling blocks, which allows the tool to deduce error-handling specifications [19]. The authors note that some error-related keywords, such as `goto err`, are indicative of error-handling blocks. This is a common practice in C-style error-handling code, but it is relatively ad hoc and does not generalise well.

APEx uses path-length and path-complexity heuristics to determine likely error-handling paths [17]. As this approach does not take into account much of the semantics or code structure, it does not perform as well as our approach. Furthermore, the usage of symbolic execution impedes scalability. Generated specifications are used in tools like EPEx that find bugs using an input error specification generated by a tool or by hand [13]. ErrDoc improves upon EPEx's bug detection and also implements ways to automatically fix four types of error-handling bugs [38].

Weimer et al. use association analysis on pairs of actions (e.g. initialisation and clean-up) to detect likely error-raising functions and then use that information to deduce error specifications [41]. This limited association analysis limits the applicability and usefulness of their approach.

8.3 Similarity-based Techniques

Saha et al. have developed Hector to statically find resource-release omission faults in error-handling code [36]. This is a specific type of bug where an error-handling path is lacking one or more resource-release operations. Examples include missing unlock calls, missing memory frees, etc. One of their key ideas is that basic blocks nearby other blocks containing resource-release operations might also need those operations. IPPO statically finds bugs in security-critical paths, including error paths, using differential checking of similar paths [21]. Some approaches use NLP-like techniques, sometimes even combined with outlier-based analysis to discover missing check bugs like Chucky [46] or try to be a bug-generic analyser like FICS [3]. Yamaguchi et al. extend upon the idea of Chucky to find taint-style vulnerabilities such as Heartbleed [45]. APISan uses semantic beliefs to detect API misuses. These beliefs are computed using symbolic execution to collect constraints on call instructions. These constraints are then compared using an outlier-based analysis [47].

8.4 Other Static Techniques

NDI finds inconsistencies in error-handling code with a focus on unobservable propagation inconsistencies [49]. It detects inconsistencies regarding resource initialisation, NULL or invalid pointer dereferences, and resource releases. Their key insight is that a caller of an error-returning function must be able to observe whether an error occurred. They detect whether different error-handling paths are distinguishable for the caller. If there is a non-observable inconsistency, then the caller will not be able to resolve that inconsistency, which means there might be a bug.

Resource acquisition and release functions often appear together as pairs in C code. Resources are first acquired, and when execution fails they are typically released in the reverse order. DiEH uses this observation to automatically mine patterns of such pairings in Linux, and based on this information, it can detect incorrect error-handling clean-up code [44].

9 Conclusion

We presented a scalable and efficient static analyser called ESSS to find bugs against error-checking code and error-

propagation code. Based on matching the longest common subsequence of path slices, we infer precise error specifications without a priori knowledge about the analysed software. We evaluated our tool on real-world widely-used software, and we have shown that our approach scales well in run time and memory usage. This scalability makes it an excellent choice to integrate into both CI/CD pipelines and into IDEs. By analysing OpenSSL, OpenSSH, and PHP, we sent patches for 46 bugs, one with a CVE. All patches were applied.

Availability

Our tool ESSS has been open sourced and is available at https://github.com/csl-ugent/esss.

Acknowledgements

The authors thank the anonymous reviewers, the shepherd, Thomas Faingnaert, and Bjorn De Sutter for their valuable feedback. The authors also thank the maintainers of the evaluated software projects for confirming the issues and reviewing the patches. This research was partly funded by the Cybersecurity Initiative Flanders (CIF) from the Flemish Government and by the Fund for Scientific Research - Flanders (FWO), Grant No. 1SF7923N.

Conflicts of Interests

Halfway through the evaluation process of this paper, the first author became involved in PHP's maintenance. This is not a paid role and has no financial gains. We made sure that all pull requests were reviewed by other maintainers, and that the CVE bug report was reviewed by the PHP security team (which at the time did not yet include the first author).

References

[1] Owasp Top Ten. https://owasp.org/www-project-top-ten/.

[2] Mithun Acharya and Tao Xie. Mining API error-handling specifications from source code. In *Fundamental Approaches to Software Engineering: 12th International Conference, FASE 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 12*, pages 370–384. Springer, 2009.

[3] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In *USENIX Security Symposium*, pages 2025–2040, 2021.

[4] Jia-Ju Bai, Tuo Li, and Shi-Min Hu. DLOS: Effective Static Detection of Deadlocks in OS Kernels. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 367–382, 2022.

[5] Daniel DeFreez, Haaken Martinson Baldwin, Cindy Rubio-González, and Aditya V Thakur. Effective Error-Specification Inference via Domain-Knowledge Expansion. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 466–476, 2019.

[6] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. *ACM SIGOPS Operating Systems Review*, 35(5):57–72, 2001.

[7] GitHub. CodeQL: the libraries and queries that power security researchers around the world, as well as code scanning in GitHub Advanced Security. https://github.com/github/codeql.

[8] Zuxing Gu, Jiecheng Wu, Chi Li, Min Zhou, Yu Jiang, Ming Gu, and Jiaguang Sun. Vetting API Usages in C Programs with IMChecker. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 91–94. IEEE, 2019.

[9] Zuxing Gu, Jiecheng Wu, Jiaxiang Liu, Min Zhou, and Ming Gu. An Empirical Study on API-Misuse Bugs in Open-Source C Programs. In *2019 IEEE 43rd annual computer software and applications conference (COMPSAC)*, volume 1, pages 11–20. IEEE, 2019.

[10] Haryadi S Gunawi, Cindy Rubio-González, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Ben Liblit. EIO: Error Handling is Occasionally Correct. In *FAST*, volume 8, pages 1–16, 2008.

[11] Yang Hu, Wenxi Wang, Casen Hunger, Riley Wood, Sarfraz Khurshid, and Mohit Tiwari. ACHyb: A Hybrid Analysis Approach to Detect Kernel Access Control Vulnerabilities. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 316–327, 2021.

[12] Immunant. immunant/c2rust: Migrate C code to Rust. https://github.com/immunant/c2rust/tree/master.

[13] Suman Jana, Yuan Jochen Kang, Samuel Roth, and Baishakhi Ray. Automatically detecting error handling bugs using error specifications. In *USENIX Security Symposium*, pages 345–362, 2016.

[14] Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, Ji Wang, Xiaodong Liu, and Yunhuai Liu. Detecting Error-Handling Bugs without Error Specification Input. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 213–225. IEEE, 2019.

[15] Zu-Ming Jiang, Jia-Ju Bai, Julia Lawall, and Shi-Min Hu. Fuzzing Error Handling Code in Device Drivers Based on Software Fault Injection. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 128–138. IEEE, 2019.

[16] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing Error Handling Code using Context-Sensitive Software Fault Injection. In *the 29th USENIX Security Symposium (Security'20)*, 2020.

[17] 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.

[18] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[19] 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.

[20] 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.

[21] 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.

[22] Yongzhi Liu, Xiarun Chen, Zhou Yang, and Weiping Wen. Automatically constructing peer slices via semantic and context-aware security checks in the linux kernel. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 108–113. IEEE, 2021.

[23] Kangjie Lu and Hong Hu. Where Does It Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1867–1881, 2019.

[24] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Automatically Identifying Security Checks for Detecting Kernel Semantic Bugs. In *Computer Security–ESORICS 2019: 24th European Symposium on Research in Computer Security, Luxembourg, September 23–27, 2019, Proceedings, Part II 24*, pages 3–25. Springer, 2019.

[25] Kangjie Lu, Aditya Pakki, and Qiushi Wu. Detecting Missing-Check Bugs via Semantic- and Context-Aware Criticalness and Constraints Inferences. In *Proceedings of the 28th USENIX Conference on Security Symposium*, 2019.

[26] Evan Miller. How not to sort by average rating. https://www.evanmiller.org/how-not-to-sort-by-average-rating.html, Feb 2009.

[27] Hoan Anh Nguyen, Robert Dyer, Tien N Nguyen, and Hridesh Rajan. Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–177, 2014.

[28] OpenSSL. openssl/crypto/rsa/rsa_sp800_56b_check.c at 8d927e55 · openssl/openssl. https://github.com/openssl/openssl/blob/8d927e55b751ba1af6c08cd4e37d565a43c56157/crypto/rsa/rsa_sp800_56b_check.c#L368-L437.

[29] OpenSSL Foundation Inc. RAND_bytes_ex. https://www.openssl.org/docs/man3.0/man3/RAND_bytes_ex.html.

[30] OpenSSL Foundation Inc. Release strategy. https://www.openssl.org/policies/releasestrat.html.

[31] Aditya Pakki and Kangjie Lu. Exaggerated Error Handling Hurts! An In-Depth Study and Context-Aware Detection. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1203–1218, 2020.

[32] php src. php-src/zend_api.c at 7c3b92fc · php/php-src. https://github.com/php/php-src/blob/7c3b92fc913e7606cbc33c68eeddff36256c33f7/Zend/zend_API.c#L4783-L4789.

[33] Cindy Rubio-González, Haryadi S Gunawi, Ben Liblit, Remzi H Arpaci-Dusseau, and Andrea C Arpaci-Dusseau. Error propagation analysis for file systems. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–280, 2009.

[34] 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.

[35] Cindy Rubio-González and Ben Liblit. Defective Error/-Pointer Interactions in the Linux Kernel. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 111–121, 2011.

[36] Suman Saha, Jean-Pierre Lozi, Gaël Thomas, Julia L Lawall, and Gilles Muller. Hector: Detecting Resource-Release Omission Faults in error-handling code for systems software. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12. IEEE, 2013.

[37] Qintao Shen, Hongyu Sun, Guozhu Meng, Kai Chen, and Yuqing Zhang. Detecting API Missing-Check Bugs Through Complete Cross Checking of Erroneous Returns. In *International Conference on Information Security and Cryptology*, pages 391–407. Springer, 2022.

[38] 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.

[39] Wenwen Wang, Kangjie Lu, and Pen-Chung Yew. Check It Again: Detecting Lacking-Recheck Bugs in OS Kernels. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1899–1913, 2018.

[40] Westley Weimer and George C Necula. Finding and Preventing Run-Time Error Handling Mistakes. In *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, 2004.

[41] Westley Weimer and George C Necula. Mining Temporal Specifications for Error Detection. In *Tools and Algorithms for the Construction and Analysis of Systems: 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005. Proceedings 11*, pages 461–476. Springer, 2005.

[42] Edwin B Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

[43] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *The 2020 Annual Network and Distributed System Security Symposium (NDSS'20)*, 2020.

[44] 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)*, pages 2041–2058, 2021.

[45] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic Inference of Search Patterns for Taint-Style Vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812. IEEE, 2015.

[46] 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.

[47] Insu Yun, Changwoo Min, Xujie Si, Yeongjin Jang, Taesoo Kim, and Mayur Naik. APISan: Sanitizing API Usages through Semantic Cross-Checking. In *Usenix Security Symposium*, pages 363–378, 2016.

[48] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M Azab, and Ruowen Wang. PeX: A Permission Check Analysis Framework for Linux Kernel. In *28th USENIX Security Symposium*, 2019.

[49] Qingyang Zhou, Qiushi Wu, Dinghao Liu, Shouling Ji, and Kangjie Lu. Non-Distinguishable Inconsistencies as a Deterministic Oracle for Detecting Security Bugs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3253–3267, 2022.

A Generality

Although we focused on C systems code bases, our approach is not fundamentally constrained to such code bases. In fact, because our implementation is based on LLVM, we can apply it to any language that can compile to LLVM IR. This means we can deal with C++, and we can even use our approach on Rust. Of course, both languages not only allow users to return error values in a manner similar to C, but each also has a more language-specific idiomatic way of returning errors. We discuss how our approach deals with language-specific error handling for both Rust and C++.

A.1 Rust

Rust implements monad-like structures for handling errors: `Option<T>` and `Result<T, E>`. Idiomatic Rust code typically wraps error and success return values in these structures.

We therefore do not expect to find bugs with our analysis technique in idiomatic Rust code. To test this hypothesis we experimented with two popular Rust libraries: serde (commit 2ba40672) and webrender (commit 5f0de38b). Most importantly, our tool indeed correctly infers the error specifications for these structures (e.g. `Option<T>::unwrap()` uses a discriminator of 0 to indicate errors). However, our tool reported no bugs in either of these libraries. This indicates that Rust's approach for a type-system-based error-propagation mechanism likely prevents numerous bugs at compile time. However, it can still be useful to use ESSS to detect bugs in less idiomatic code, e.g. code that interacts across different programming languages.

We experimented with the output of C2Rust [12], which is a C to Rust transpiler. C2Rust is generally used to help programmers in transitioning a C code base to Rust. However, it does not transition the transpiled code base to safe Rust: this is left to the developers. It hence preserves the unsafety and semantics of the original C code. This results in Rust code that still uses the C-style error representation and error handling. We created some toy examples of C code that we transpiled to Rust using C2Rust. We found that the error specification inference produced the same results for both the original C code as the transpiled Rust code.

This leads us to another potential application domain of our technique. Because the inferred error specification for non-idiomatic Rust code still contains the error-returning functions and their error values, this information could help transpilation tools to automatically generate more idiomatic Rust code. Our tool derives what functions return an error and what their error values are. It would be possible to automatically generate enum values to use in `Result<T, E>` for example.

A.2 C++

Similarly, we tested some small C++ snippets and found a similar result for the `std::optional<T>` type. Furthermore, exceptions are also discovered by our similarity matching technique. However, since the implementation of our analysis tool only takes into account return values, it cannot detect missing exception handling. That being said, this is more a limitation of our implementation, and not one of our analysis technique per se. Extending our implementation to support these kinds of constructs is left for future work. We also confirmed that C-style return-code error values are also detected in C++ code. Finding bugs with the error specification also worked correctly.

A.3 Implementation details

Our current implementation has limited support for structures as a return value. This happens for example for Rust's `Result<T, E>` and `Option<T>` types, and for C++'s `std::optional<T>` type. We implemented basic support for structures by keeping track of which members have the role of representing error values. We handle this by checking if a conditional check uses the `extract` instruction from LLVM. This instruction extracts a member from a compound type by index. We keep track of that index and use it as a tag for the entries in the error specification. Larger structs may not necessarily use these specific instructions as the front end may choose to compile them to a pointer argument, although we did not encounter such cases.

B Fixed Bugs

Table 6 contains an overview of the fixed bugs per project, the type of bug, and the bugs' impact.

C EESI Domain Knowledge

For OpenSSL and zlib, we gave the same domain knowledge as EESI's authors used. Since both zlib and libpng do not contain typical error-raising functions, we set their error-only input to the default `__errno_location`, and set the only known specification to `malloc == 0`. For freetype2, we also gave EESI the previous two inputs, and added the `FT_Throw` function as an error-raising function. For libwebp, we use the default `__errno_location` and `VP8SetError`, `VP8LSetError`, and `WebPEncodingSetError`.

Project	File	Description	Impact
OpenSSL	crypto/asn1/asn_pack.c	Incorrect `ASN1_item_i2d()` check	Malfunction: corrupt data
	crypto/asn1/asn1_parse.c	Incorrect `BIO_dup_state()` check	Crash
	crypto/asn1/asn1_parse.c	Incorrect `BIO_set_indent()` check	Malfunction: incorrect output
	crypto/bio/bss_acpt.c	Incorrect `BIO_set_accept_name()` check	Malfunction: parameter not set
	crypto/bn/bn_rsa_fips186_4.c	Incorrect error branch in `ossl_bn_rsa_fips186_4_derive_prime()`	Weak randomness in case of random number generator failure
	crypto/cms/cms_ec.c	Incorrect `CMS_SharedInfo_encode()` check	Malfunction: corrupt data
	crypto/cms/cms_lib.c	Incorrect `BIO_set_md()` check	Malfunction: digest not set
	crypto/evp/ctrl_params_translate.c	Incorrect `default_check()` check	Malfunction: parameter not set
	crypto/evp/evp_lib.c	Incorrect `EVP_CIPHER_param_to_asn1()` check	Malfunction: parameter not set
	crypto/evp/evp_pbe.c	Incorrect `EVP_get_digestbynid()` check	Malfunction: fallback path not taken resulting in crashes
	crypto/evp/p5_crpt2.c	Incorrect `EVP_CIPHER_asn1_to_param()` check	Malfunction: parameter not set
	crypto/pkcs7/pk7_doit.c	Incorrect `EVP_CIPHER_param_to_asn1()` check	Malfunction: parameter not set
	crypto/ffc/ffc_params_generate.c	Incorrect `RAND_bytes_ex()` check	Weak randomness in case of random number generator failure
	crypto/ocsp/ocsp_ext.c	Incorrect `X509V3_add1_i2d()` check	Malfunction: nonce not added
	crypto/x509/v3_prn.c	Incorrect `method->i2r()` check	Malfunction: incorrect output
	providers/.../rsa_sig.c	Incorrect `RSA_public_decrypt()` check	Malfunction: incorrect error reporting
OpenSSH	addr.c	Incorrect `getnameinfo()` check	Crash
PHP	ext/curl/multi.c	Missing `zend_fcall_info_init()` error check	Crash
	ext/ftp/php_ftp.c	Missing `ftp_quit()` propagation check	Malfunction: incorrect error reporting
	ext/phar/util.c	Missing `EVP_MD_CTX_create()` check	Crash
	ext/phar/util.c	Missing `EVP_VerifyInit()` check	Crash
	ext/phar/util.c	Missing `EVP_VerifyUpdate()` check	Malfunction: corrupt data
	ext/.../mbfilter_7bit.c	Missing `mbfl_filt_conv_illegal_output()` check	Malfunction: corrupt data
	ext/.../mbfilter_iso2022jp_mobile.c	Missing `mbfilter_unicode2sjis_emoji_kddi()` check	Malfunction: corrupt data
	ext/.../mbfilter_sjis_mobile.c * 3	Missing `mbfl_filt_conv_illegal_output()` check	Malfunction: corrupt data
	ext/.../mbfilter_sjis_mobile.c	Missing `filter->output_function()` check	Malfunction: corrupt data
	ext/.../mbfilter_sjis_mobile.c	Missing `mbfilter_unicode2sjis_emoji_docomo()` check	Malfunction: corrupt data
	ext/.../mbfilter_sjis_mobile.c	Missing `mbfilter_unicode2sjis_emoji_kddi()` check	Malfunction: corrupt data
	ext/.../mbfilter_sjis_mobile.c	Missing `mbfilter_unicode2sjis_emoji_sb()` check	Malfunction: corrupt data
	ext/openssl/openssl.c	Missing `PEM_write_bio_PKCS7()` check	Malfunction: corrupt data
	ext/openssl/openssl.c	Missing `PEM_write_bio_CMS()` check	Malfunction: corrupt data
	ext/openssl/openssl.c	Missing `i2d_PKCS12_bio()` check	Malfunction: corrupt data
	ext/openssl/xp_ssl.c	Incorrect `SSL_CTX_set0_tmp_dh_pkey()` check	Malfunction: parameter not set leading to crashes
	ext/openssl/xp_ssl.c	Incorrect `SSL_CTX_set_tmp_dh()` check	Malfunction: parameter not set leading to crashes
	ext/openssl/xp_ssl.c	Missing `php_openssl_set_server_dh_param()` check	Crash
	ext/pdo_odbc/odbc_driver.c	Incorrect `SQLAllocHandle()` check	Malfunction: spurious failures
	ext/pdo_odbc/odbc_driver.c	Missing `SQLAllocHandle()` check	Crash
	ext/session/mod_files.c * 2	Incorrect `ps_files_cleanup_dir()` propagation check	Crash
	ext/soap/php_http.c	Missing check in random number generation (CVE-2023-3247)	Weak randomness used and stack information leak
	ext/standard/browscap.c	Incorrect `pcre2_match()` check	Crash
	ext/tidy/tidy.c	Missing `tidyLoadConfig()` check	Malfunction: parameters not set
	ext/xmlwriter/php_xmlwriter.c	Missing `xmlTextWriterEndElement()` check	Malfunction: corrupt data
	Zend/zend_API.c	Incorrect `zend_update_static_property_ex()` propagation check	Malfunction and eventually a crash

Table 6: List of new bugs discovered by ESSS for which we created patches. All patches were accepted. Entries with * N indicate N similar bugs in the same file.