Automatically Detecting Error Handling Bugs using Error Specifications

Suman Jana[1], Yuan Kang[1], Samuel Roth[2], and Baishakhi Ray[3]

[1]Columbia University
[2]Ohio Northern University
[3]University of Virginia

Abstract

Incorrect error handling in security-sensitive code often leads to severe security vulnerabilities. Implementing correct error handling is repetitive and tedious especially in languages like C that do not support any exception handling primitives. This makes it very easy for the developers to unwittingly introduce error handling bugs. Moreover, error handling bugs are hard to detect and locate using existing bug-finding techniques because many of these bugs do not display any obviously erroneous behaviors (*e.g.,* crash and assertion failure) but cause subtle inaccuracies.

In this paper, we design, implement, and evaluate EPEx, a tool that uses error specifications to identify and symbolically explore different error paths and reports bugs when any errors are handled incorrectly along these paths. The key insights behind our approach are: (i) real-world programs often handle errors only in a limited number of ways and (ii) most functions have simple and consistent error specifications. This allows us to create a simple oracle that can detect a large class of error handling bugs across a wide range of programs. We evaluated EPEx on 867,000 lines of C Code from four different open-source SSL/TLS libraries (OpenSSL, GnuTLS, mbedTLS, and wolfSSL) and 5 different applications that use SSL/TLS API (Apache httpd, cURL, Wget, LYNX, and Mutt). EPEx discovered 102 new error handling bugs across these programs—at least 53 of which lead to security flaws that break the security guarantees of SSL/TLS. EPEx has a low false positive rate (28 out of 130 reported bugs) as well as a low false negative rate (20 out of 960 reported correct error handling cases).

1 Introduction

Error handling is an important aspect of software development. Errors can occur during a program's execution due to various reasons including network packet loss, malformed input, memory allocation failure, *etc.* Handling these errors correctly is crucial for developing secure and robust software. For example, in case of a recoverable error, a developer must ensure that the affected program invokes the appropriate error recovery code. In contrast, if an error is critical, the program must display an appropriate error message and fail in a safe and secure manner. Error handling mistakes not only cause incorrect results, but also often lead to security vulnerabilities with disastrous consequences (*e.g.,* `CVE-2014-0092`, `CVE-2015-0208`, `CVE-2015-0288`, `CVE-2015-0285`, and `CVE--2015-0292`). More worryingly, an attacker can often remotely exploit error handling vulnerabilities by sending malformed input, triggering resource allocation failures through denial-of-service attacks *etc.*

To understand how incorrect error handling can lead to severe security vulnerabilities in security-sensitive code, consider the bug in GnuTLS (versions before 3.2.12), a popular Secure Sockets Layer (SSL) and Transport Layer Security (TLS) library used for communicating securely over the Internet, that caused `CVE-2014-0092`. Listing 1 shows the relevant part of the affected X.509 certificate verification code. The function `_gnutls_verify_certificate2` called another function `check_if_ca` to check whether the issuer of the input certificate is a valid Certificate Authority (CA). `check_if_ca` returns < 0 to indicate an error (lines 4 and 5 of Listing 1). However, as line 16 shows, `_gnutls_verify_certificate2`, the caller function, only handles the case where the return value is 0 and ignores the cases where `check_if_ca` returns negative numbers as errors. This missing error check makes all applications using GnuTLS incorrectly classify an invalid certificate issuer as valid. This bug completely breaks the security guarantees of all SSL/TLS connections setup using GnuTLS and makes them vulnerable to man-in-the-middle attacks. In summary, *Listing 1:* GnuTLS error handling bug (CVE-2014-0092). The lines marked in color gray show an error path and the red lines highlight the source of error handling bug.

```
int check_if_ca (...)
{ ...
  result = ...;
  if (result < 0) {
   goto cleanup;
  }
  ...
  result = 0;

cleanup:
  return result;
}

int _gnutls_verify_certificate2 (...)
{ ...
  if (check_if_ca (...)  == 0)  {
     result = 0;
     goto cleanup;
  }
  ...
  result = 1;

cleanup:
  return result;
```

this bug renders all protections provided by GnuTLS useless.

Developers often introduce error handling bugs unwittingly, as adding error checking code is repetitive and cumbersome (especially in languages like C that do not provide any exception handling primitives). Moreover, a large number of errors in real systems cannot be handled correctly at their source due to data encapsulation and, therefore, must be propagated back to the relevant module. For example, if a protocol implementation receives a malformed packet that cannot be parsed correctly, the parsing error must be appropriately transformed and propagated to the module implementing the protocol state machine in order to ignore the packet and recover gracefully. Implementing correct error propagation in real-world software is non-trivial due to their complex and intertwined code structure.

Automated detection of error handling bugs can help developers significantly improve the security and robustness of critical software. However, there are three major challenges that must be tackled in order to build such a tool: (i) **error path exploration.** Error handling code is only triggered in corner cases that rarely occur during regular execution. This severely limits the ability of dynamic analysis/testing to explore error paths. Moreover, error handling code is usually buried deep inside a program and, therefore, is hard to reach using off-the-shelf symbolic execution tools due to path explosion; (ii) **lack of an error oracle.** Error handling bugs often result in silent incorrect behavior like producing wrong output, causing memory corruption, *etc.* as shown in the previous example. Therefore, accurately separating incorrect error handling behavior from the correct ones is a hard problem; and (iii) **localizing error handling bugs.** Finally, the effects of error handling bugs are usually manifested far away from their actual sources. Accurately identifying the origin of these bugs is another significant problem.

Our contributions. In this paper, we address all these three problems as discussed below. We design, implement, and evaluate EPEX, a novel algorithm that can automatically detect error-handling bugs in sequential C code.

Identification and scalable exploration of error paths. As low-level languages like C do not provide any exception handling primitives, the developers are free to use any arbitrary mechanism of their choice for communicating errors. However, we observe that real-world C programs follow simple error protocols for conveying error information across different modules. For example, distinct and non-overlapping integer values are used throughout a C program to indicate erroneous or error-free execution. Integer values like 0 or 1 are often used to communicate error-free execution and negative integers typically indicate errors. Moreover, functions that have related functionality tend to return similar error values. For example, most big number functions in OpenSSL return 0 on error. These observations allow us to create simple error specifications for a given C program, indicating the range of values that a function can return on error. Given such specifications as input, our algorithm performs under-constrained symbolic execution at the corresponding call-sites to symbolically explore only those paths that can return error values and ignores the rest of the paths. Such path filtering minimizes the path exploration problem often plaguing off-the-shelf symbolic execution tools.

Design of an error oracle. We observe that when an error occurs, most C programs usually handle the scenario in one of the following simple ways: (i) propagate an appropriate error value (according to the corresponding error protocol) upstream, (ii) stop the program execution and exit with an error code, or (iii) display/log a relevant error message. We leverage this behavior to create a simple program-independent error oracle. In particular, our algorithm checks whether errors are handled following any of the above three methods along each identified error path; if not, we mark it as a potential bug.

Accurate bug localization. Our error oracle also helps us accurately localize the error handling bugs as it allows our algorithm to detect the bugs at their source. As a side-effect, we can precisely identify buggy error handling code and thus drastically cut down developers' effort in fixing these bugs.

Implementation and large-scale evaluation. Using our algorithm, we design and implement a tool, EPEX, and evaluate it. EPEX's analysis is highly parallelizable and scales well in practice. EPEX can be used to find error-handling bugs in any C program as long as the above mentioned assumptions hold true. We evaluated EPEX on a total of 867,000 lines of C code [56] from 4 different open-source SSL/TLS libraries (OpenSSL, GnuTLS, mbedTLS, and wolfSSL) and 5 different applications using SSL/TLS APIs (cURL, Wget, Apache httpd, mutt, and LYNX). EPEX discovered 102 new error handling bugs across these programs—at least 53 of which lead to critical security vulnerabilities that break the security guarantees of SSL/TLS. We also found that EPEX has both low false positive (28 out of 130 reported bugs) and false negative rates (20 out of 960 reported correct error handling cases). Thus, EPEX has a 78% precision and 83% recall on our tested programs. Several of our tested programs (*e.g.,* PolarSSL, cURL, and Apache httpd) have been regularly checked with state-of-the-art static analysis tools like Coverity, Fortify, etc. The fact that none of these bugs were detected by these tools also demonstrates that EPEX can detect bugs that the state-of-the-art bug finding tools miss.

The rest of this paper is organized as follows. We present a brief overview of error handling conventions in C programs in Section 2. We describe our platform- and language-independent technique for detecting error handling bugs in Section 3. The details of implementing our algorithm in Clang and the results are presented in Sections 4 and 5 respectively. We survey the related work in Section 6 and present several directions for future work in Section 7. Section 8 concludes our paper.

2 Error handling in C programs

C does not support exception handling primitives like `try-catch`. In C, a fallible function, which may fail due to different errors, *e.g.,* memory allocation failure or network error, usually communicates errors to the caller function either through return values or by modifying arguments that are passed by reference. While there are no restrictions on the data types/values that can be used to communicate errors, C programmers, in general, create an informal, program-specific error protocol and follow it in all fallible functions of a program to communicate errors. An error protocol consists of a range of error-indicating and non-error-indicating values for different data types. For example, a program may use an error protocol where any negative integer value indicates an error and 0 indicates an error-free execution. Similarly, an error protocol may also use a `NULL` pointer or a boolean value of `false` to indicate errors. The existence of such error handling protocols makes it easier for us to create error specifications for different functions of a program.

		Error Range	Non-Error Range
Libraries	OpenSSL	$e \leq 0$	$e = 1$
	GnuTLS	$-403 \leq e \leq -1$	$e = 0$
	mbedTLS	$e < 0$	$e = 0$
	wolfSSL	$-213 \leq e \leq -1$	$e \in \{0,1\}$
Applications	httpd [51]	$1 \leq e \leq 720000$	$e = 0$
	curl	$1 \leq e \leq 91$	$e = 0$
	lynx	$-29999 \leq e \leq -1$	$e \geq 0$
	mutt	$e = -1$	$e \geq 0$
	wget	$e = -1$	$e = 0$
e represents the return values of fallible functions			

Table 1: Error protocols of the tested libraries/applications

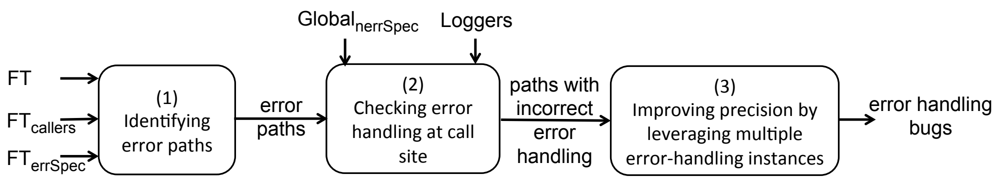
For example, consider the C programs that we studied in this work. Table 1 shows their error protocols. Fallible functions in OpenSSL usually return 0 or a negative integer to indicate errors and 1 to indicate error-free execution. In contrast, GnuTLS uses negative integers between -1 and -403 to indicate errors and 0 to indicate error-free execution. In spite of the variety of protocols, in all the cases, error-indicating and non-error-indicating ranges for fallible functions do not overlap, to avoid ambiguities.

3 Methodology

In this section, we introduce the details of EPEX (**E**rror **P**ath **Ex**plorer), a tool for automatically detecting different types of error handling bugs in sequential C programs. Our key intuition is that if an error is returned by a function in a program path, that error must be handled correctly along that path according to the program's error convention. Given a function under test, say FT, EPEX identifies possible *error paths*—the paths along which FT returns error values, and ensures that the error values are handled correctly along the error paths at the call site; if not, EPEX reports bugs due to missing error-handling.

3.1 Overview

An overview of EPEX's workflow for an individual API function is presented in Figure 1. EPEX takes five inputs: the signature of the fallible function under test (FT), the caller functions of FT ($FT_{callers}$), a specification defining a range of error values that FT can return ($\text{FT}_{errSpec}$), a range of return values that are used to indicate error-free execution according to the test program's error protocol ($\text{Global}_{nerrSpec}$), and a set of error logging functions used by the program (Loggers). The list of fallible functions,



Global$_{nerrSpec}$
Loggers
FT
FT$_{callers}$
FT$_{errSpec}$
(1) Identifying error paths
error paths
(2) Checking error handling at call site
paths with incorrect error handling
(3) Improving precision by leveraging multiple error-handling instances
error handling bugs

Figure 1: EPEx workflow

their error specifications, and list of error logging functions are created manually, while their caller functions are automatically identified by EPEx. EPEx then works in three steps.

In Step-I, by performing under-constrained symbolic execution at $FT_{callers}$, EPEx identifies the error paths along which FT returns an error value, based on FT$_{errSpec}$. For example, in Listing 1, `check_if_ca`'s error specification says the function will return ≤ 0 on error. Hence, Step-I symbolically executes `_gnutls_verify_certificate2` function and marks the path along the `if` branch in the `check_if_ca` function as an error path (marked in color gray).

Next, in Step-II, EPEx checks if the call site of FT handles the error values correctly. In particular, EPEx checks that if FT returns an error, the error value is handled by the caller in one of the following ways: it (i) pushed the error upstream by returning a correct error value from the caller function, (ii) stopped the program execution with a non-zero error code, or (iii) logged the error by calling a program-specific logging function. If none of these actions take place in an error path, EPEx reports an error handling bug. For instance, in case of Listing 1, the error path returns < 0 at the call site, `_gnutls_verify_certificate2` (line 16). However, the error value is not handled at the call site; in fact it is reset to 1 (line 21), which is a non-error value as per Global$_{nerrSpec}$. Thus, in this case, an error path will return a non-error value. EPEx reports such cases as the potential error-handling bugs (marked in red).

Finally, in Step-III, EPEx checks how error handling code is implemented in other call sites of FT. For example, if all other FT call sites ignore an error value, EPEx does not report a bug even if the error value is not handled properly at the call site under investigation. As FT$_{errSpec}$ may be buggy or symbolic execution engines may be imprecise, this step helps EPEx reduce false positives. The final output of EPEx is a set of program error paths—FT signature, call-site location, and error paths in the caller functions along with EPEx's diagnosis of correct and buggy error handling. We present the detailed algorithm in the rest of this section.

Algorithm 1: EPEx workflow

```
EPEx (FT, FT$_{errSpec}$, FT$_{callers}$, Global$_{nerrSpec}$, Loggers)
Input : function FT, error spec FT$_{errSpec}$, callers of FT
        FT$_{callers}$, global non-error spec Global$_{nerrSpec}$,
        error logging functions Loggers
Output: Bugs
________________________________________________
Bugs ⇐ $\phi$
shouldHandle ⇐ False
for each caller $c \in FT_{callers}$ do
    for each argument $a \in c.inputArguments$ do
        $a.isSymbolic \Leftarrow True$
    end
    for each path $p \in c.Paths$ do
        isErrPath ⇐ False
        errPts ⇐ $\phi$
        for each program point $s \in p$ do
            /* Step-I : identifying error paths */
            if $s$ calls FT then
                FT$_{ret}$ ⇐ symbolic return value of FT
                isErrPath ⇐ chkIfErrPath(FT$_{ret}$,
                  FT$_{errSpec}$)
                if isErrPath = True then
                    errPts ⇐ errPts ∪ $s.location$
                end
            end
            if isErrPath = True then
                /* Step-II : checking error Handling */
                isHandled ⇐ chkErrHandling($s$,
                  Global$_{nerrSpec}$, Loggers)
                if (isHandled = unhandled) or
                  (isHandled = maybe_handled) then
                    Bugs ⇐ Bugs ∪ ⟨errPts, isHandled⟩
                end
                if (isHandled = handled) or (isHandled =
                  maybe_handled) then
                    /* Resetting an error path */
                    isErrPath ⇐ False
                    errPts ⇐ $\phi$
                end
                /* Example requires error handling */
                if (isHandled = handled) then
                    shouldHandle ⇐ True
                end
            end
        end
    end
end
/* Step-III : Leveraging multiple error-handling
  instances */
if shouldHandle then
    return Bugs
else
    return $\phi$
end
```

3.2 Step-I: identifying error paths

We define *error paths* to be the program paths in which a function call fails and returns error values. To identify the error paths for a function, EPEX first has to know the error values that a function can return; EPEX takes such information as input (see Section 2 for details). Then the program paths along which the function returns the error values are identified as error paths. The call sites of the failed function are treated as error points (errPts in Algorithm 1). For example, in Listing 1, the program path containing an if-branch (highlighted gray) is an error path; line 16 of `_gnutls_verify_certificate2` is an error point along that error path. Note that an error path can have one or more error points. Given a function under test, say FT, and its caller functions FT_{caller}, the goal of Step-I is to explore all possible error paths going through FT and mark the corresponding error points.

Algorithm 2: Step-I: Identifying error paths

```
1  chkIfErrPath ($FT_{ret}$, $FT_{errSpec}$)
   Input  : $FT_{ret}$, $FT_{errSpec}$
   Output: isErrPath, $FT_{ret}$
2  ----------------------------------------
3  if $FT_{ret} \wedge FT_{errSpec}$ is satisfiable then
4      /* Error path is possible */
5      if $FT_{ret} \wedge \neg FT_{errSpec}$ is satisfiable then
6          /* Force the error path, if needed */
7          $FT_{ret} \Leftarrow FT_{ret} \wedge FT_{errSpec}$
8      return True
9  else
10     /* Error path is impossible */
11     return False
12 end
```

Exploring error paths. First, EPEX performs under-constrained symbolic execution at each caller function in $FT_{callers}$, and monitors each method call to check if FT is called. If EPEX finds such a call, then right after returning from symbolically executing FT, EPEX checks if the path conditions involving the symbolic return value (FT_{ret}) satisfy its error specifications ($FT_{errSpec}$), as mentioned in the input spec (see Algorithm 2), *i.e.* if $FT_{ret} \wedge FT_{errSpec}$ is satisfiable. This helps EPEX identify two main cases:

- **Error path possible.** If $FT_{ret} \wedge FT_{errSpec}$ is satisfiable, the error path is possible. But while continuing to analyze the error path, EPEX must make sure the constraints make the error path inevitable, so it checks if $FT_{ret} \wedge \neg FT_{errSpec}$ is satisfiable, and if so, sets FT_{ret} to $FT_{ret} \wedge FT_{errSpec}$, so that the constraints force the error path to be taken.

- **Error path impossible.** When $FT_{ret} \wedge FT_{errSpec}$ is unsatisfiable, EPEX considers it as not an error path and stops tracking it any further.

Algorithm 2 illustrates this process. If a path is considered to be an error path, EPEX notes the corresponding call-site locations in the source code as error points and continues tracking the path in Step-II. In Listing 1, the buggy path has `check_if_ca` return a negative value, which means that it is certainly an error path, and the algorithm returns True, without having to further restrict the constraints.

3.3 Step-II: checking error handling

If a path is marked as an error path in Step-I ($isErrPath == True$ in Algorithm 1), this step checks whether the error is handled properly along the error path in the caller function. As the symbolic execution engine explores different error paths, we propagate the error path state (*e.g., isErrPath*, FT, and *errPts*) to any new path forked from conditional branches. We let the rest of the symbolic execution proceed normally unless one of the following happens (see Algorithm 3):

At return point. If EPEX encounters a return statement along an error path, it checks whether the error value is pushed upstream. To do that, Step-II takes program-wide specifications for non-error values ($Global_{nerrSpec}$) as input and checks the constraints on the returned variable of FT_{caller} against $Global_{nerrSpec}$ to determine whether FT_{caller} is returning an error value along the error path. If the returned variable can only contain a non-error value, EPEX marks the corresponding path to be *unhandled*; if it may have a non-error value or an error value, EPEX marks it as *maybe_handled*; and if it cannot have any non-error values, EPEX marks the path as *handled*.

Although both *maybe_handled* and *unhandled* indicate potential bugs, we differentiate between them because in Line 27 of Algorithm 1, we no longer count the path as an error path in the case of *maybe_handled*, since we have already found where the error could be handled; the same error value does not have to be checked repeatedly.

At exit point. A call to libc function `exit` (or other related function like `_exit`) ends the execution of a path. In such a case, EPEX checks the constraints on the symbolic argument to the exit function along an error path: if the symbolic argument can have only error or non-error indicating value, EPEX marks the path as *handled* or *unhandled* respectively. If the argument may have both error and non-error indicating values, EPEX marks the path as *maybe_handled*.

At logging point. The global specifications also support providing the names of the program-specific error logging functions (Loggers). In most C programs, errors are logged through special logging or alerting functions.

If an error path calls an error logging function, EPEx marks that path as *handled*.

In Listing 1, `_gnutls_verify_certificate2` sets `result` to the non-error value, 1, in the error path before returning it, so the algorithm classifies the error as *unhandled*.

Algorithm 3: Step-II: Checking error handling

```
1  chkErrHandling ($s$, Global$_{nerrSpec}$, Loggers)
   Input  : program point $s$, global non-error spec
            Global$_{nerrSpec}$, error logging functions Loggers
   Output: isHandled
2  ———————————————————————————————
3  if ($s$ is a top-level ret statement) or ($s$ is a call to "exit")
   then
4      $tval$ ← symbolic return value/exit argument
5      if ($tval$ ∧ Global$_{nerrSpec}$ is satisfiable) and ($tval$ ∧ ¬
       Global$_{nerrSpec}$ is unsatisfiable) then
6          return unhandled
7      else if ($tval$ ∧ Global$_{nerrSpec}$ is unsatisfiable) and
       ($tval$ ∧ ¬ Global$_{nerrSpec}$ is satisfiable) then
8          return handled
9      else if ($tval$ ∧ Global$_{nerrSpec}$ is satisfiable) and ($tval$
       ∧ ¬ Global$_{nerrSpec}$ is satisfiable) then
10         return maybe_handled
11 else if $s$ ∈ Loggers then
12     return handled
13 return not_checked
```

3.4 Step-III: leveraging multiple error-handling instances

As program documentation may be buggy or symbolic execution engines may be imprecise, EPEx compares the analysis results across multiple callers of the function under test (FT) to minimize false positives. Lines $34-45$ in Algorithm 1 present this step. If EPEx finds that all the callers of FT return *unhandled* or *maybe_handled*, EPEx ignores the corresponding bugs and does not report them. However, if at least one caller sets *isHandled* to *handled*, all the buggy paths marked from Step-II (line 25 in Algorithm 1) will be reported as bugs. The underlying idea behind this step is inspired by the seminal work of Engler *et al.* [16] where deviant behaviors were shown to indicate bugs.

For example, function `gnutls_x509_trust_list_add_trust_file` adds each Certificate Authority (CA) mentioned in the input file to the list of trusted CAs. In an error-free execution, it returns the number of added CAs. It returns a negative number in case of a parsing error. However, in all the 5 instances in GnuTLS where `gnutls_x509_trust_list_add_trust_file` is called, step II indicates that error values are not handled correctly. In such cases, Step III assumes that the error values can be safely ignored and does not report any bugs. With manual analysis we confirmed that as trusted certificate authorities can be loaded from multiple sources, such errors can indeed be ignored safely.

4 Implementation

EPEx is implemented using the Clang static analysis framework [42] (part of the LLVM project) and its underlying symbolic execution engine. The Clang analyzer core symbolically explores all feasible paths along the control flow graph of an input program and provides a platform for custom, third-party *checkers* to monitor different paths, inspect the constraints of different symbolic values along those paths, and add additional constraints if necessary. A typical checker often looks for violations of different invariants along a path (*e.g.,* division by zero). In case of a violation, the checker reports bugs. We implement EPEx as a checker inside the Clang analyzer. The rest of this section describes how EPEx is implemented as a Clang checker in detail.

Error specifications. EPEx takes a text file containing the per-function error specifications (FT$_{errSpec}$), global non-error specification (Global$_{nerrSpec}$), and global error specification (Global$_{errSpec}$) as input. Listing 2 shows a sample input file. FT$_{errSpec}$ contains five parameters: ⟨function name, number of arguments, return type, lower bound of error value, upper bound of error value⟩. The first three parameters define a function signature. The number of arguments and return type (*e.g.,* integer, boolean, *etc.*) help to disambiguate functions with identical names. The last two optional parameters represent a range of error values that the function can return. For example, error specification for function `RAND_bytes` is: ⟨ RAND_bytes, 2, int, $>= -1$,$<= 0$⟩ (see line 2 in Listing 2). This shows RAND_bytes takes 2 input arguments and returns an integer value. The fourth and fifth parameters indicate that error values range from -1 to 0. Similarly, if a function `foo` takes four arguments and it has a boolean return type where `False` indicates an error, its error spec will be ⟨ foo, 4, bool, =False ⟩. We also support specifications for functions returning `NULL` pointers to indicate errors.

Since most functions in a project follow the same global error convention, (*e.g.,* most OpenSSL functions return 0 to indicate error), error specifications can be simplified by providing a global lower and upper bound of errors. If the per-function error spec does not contain any error range, *i.e.* the fourth and fifth parameters are empty, the checker uses Global$_{errSpec}$. Otherwise, the function-specific bounds override Global$_{errSpec}$. For instance, OpenSSL functions `RAND_bytes` and `RAND_pseudo_bytes` specify custom lower and upper bounds $-1 <=$ and $<=$

Listing 2: Sample error specifications for OpenSSL functions

```
1  /* Per-func error spec */
2  RAND_bytes, 2, int, >=-1, <=0
3  RAND_pseudo_bytes, 2, int, >=-1, <=0
4
5  /* Per-func spec with empty error ranges
6     (global error ranges will be used)*/
7  ASN1_INTEGER_set_int64, 2, int
8  ASN1_INTEGER_set, 1, int
9
10 /* Global error spec */
11 __GLOBAL_ERROR_BOUND__, int, =0, NA
12
13 /* Global non-error spec */
14 __GLOBAL_NOERR_VAL__, int, =1
15 __GLOBAL_NOERR_VAL__,  ptr, !=NULL
```

0 respectively, and hence global error bounds are not valid for them. Finally, Global$_{nerrSpec}$ contains non-error bounds/values for functions with different return types (see lines 14 and 15 of Listing 2). For example, $\langle$ __GLOBAL_NOERR_VAL__, int, $= 1\rangle$ indicates that any function with integer return type returns 1 to indicate an error-free execution. Similarly, $\langle$__GLOBAL_NOERR_VAL__, ptr, $! =$NULL$\rangle$ indicates an error-free execution for the functions returning pointers will result in the return pointer to be non-null. Such Global$_{nerrSpec}$ specifications are used to ensure that the return value of the caller function of FT is pushing the errors upstream correctly.

The error specifications for all the functions under test were created manually. Since, a majority of these functions either follow the per-project global error convention or, at least, functions inside same modules have the same error ranges, *e.g.,* all the big number routines in OpenSSL return 0 on error, the overhead of manual spec generation is not very significant. In fact, it took only one man-day to generate error specs for all 256 functions that we have examined. Table 2 shows the number of specified functions for each library, and the number of unique specifications. Except for WolfSSL, where we used more individualized specifications, each library contained no more than 10 unique error specifications, so that we were able to generate 256 specifications out of only 38 unique constraints.

Table 2: Error specification counts

Library	**Functions**	**Unique Specifications**
OpenSSL	109	9
GnuTLS	58	3
mbedTLS	46	10
wolfSSL	43	16
Total	256	38

As the same set of library functions are used by multiple applications, the same error specifications can be reused for all such applications. In fact, for our test applications, we focused on only OpenSSL API functions. We also found that fallible functions that return booleans or pointers, irrespective of the library they belong to, mostly indicate errors by returning `false` and `NULL` respectively. For functions returning integer error codes, we found that the error codes were almost always represented by a macro or enumerated type that is defined in a header file and therefore was very easy to find. Functions that use the same enumerated type/macro tend to follow the same error protocol. We show some sample error specifications for OpenSSL API functions in Listing 2.

Note that once the error specifications for a set of API functions are created manually, testing new applications using the same API is very easy; the user only needs to add application-specific non-error values (*i.e.* the values indicating error-free execution) for each new application.

Identifying error paths. For identifying error paths, as mentioned in Step-I of Section 3, EPEx checker uses the built-in callback method `checkPostCall`. `checkPostCall` is called once the analyzer core finishes tracking each function body. We overrode `checkPostCall` so that it looks for the functions mentioned in the error spec, *i.e.* checks whether the current function's name, number of arguments, and return type match the specification. In case of a match, Algorithm 2 is called to check whether the function's return value satisfies the lower and upper bounds of error values as given in its error spec; if so, the current path is marked as error path.

Checking error handling. We implemented Step-II by extending the `checkPreStmt` callback method for checking the program state before `return` statements and the `checkPreCall` callback for checking the program state before calling exit functions or any program-specific error logging functions as specified in the input spec. Inside `checkPreStmt` callback, EPEx checks whether the symbolic return value satisfies the global non-error spec (see Algorithm 3). A similar check is performed for exit functions inside `checkPreCall`.

To check the satisfiability conditions of Algorithm 2 and Algorithm 3, we use Clang's built-in constraint solver.

Outputs. EPEx can be run on any single source file (say, foo.c) using the command `clang -cc1 -analyze -analyzer-checker=EPEx foo.c`. For running it on large projects like OpenSSL, mbedTLS, etc. we used Clang's `scan-build` utility such that EPEx can be run as part of the regular build process. Scan-build overrides different environment variables (*e.g.,* CC, CPP) to force the build system (e.g., `make`) to use a special compiler script that compiles the input source file and invokes EPEx on it. We pass the `-analyze-header` option to the clang analyzer core to ensure that the functions defined in any included header files are also analyzed.

The output of EPEx contains one line for each analyzed error path, as shown in Table 3. Each line in the output has four components: name of the caller function, call-site of FT, candidate error-handling location (*i.e.* return instruction, exit call or error logging), and EPEx's diagnosis about whether the error has been handled correctly or not. As each output line represents an error path and multiple error paths may pass through the same call-site, a call-site for a given FT might be repeated in the output. For example, lines 1 and 2 in Table 3 have the same call-site (ssl_lib.c:1836) but their error handling locations are different (ssl_lib.c:1899 and ssl_lib.c:1905 respectively). Note that we implement Step-III as a separate script and execute it on the output of EPEx before producing the final bug report.

Table 3: Sample EPEx output for OpenSSL function under test: RAND_pseudo_bytes.

Caller Function	Call-site	Error Handling Location	Diagnosis
SSL_CTX_new	ssl_lib.c:1836	ssl_lib.c:1899	handled
SSL_CTX_new	ssl_lib.c:1836	ssl_lib.c:1905	unhandled
dtls1_send_newsession_ticket	d1_srvr.c:1683	d1_srvr.c:1736	maybe_handled

5 Results

5.1 Study subjects

To check whether errors are handled correctly in different functions of popular SSL/TLS libraries as well as applications using them, we ran EPEx on four libraries: OpenSSL, GnuTLS, mbedTLS (formerly known as PolarSSL), and wolfSSL (formerly known as cyaSSL), and five applications that use OpenSSL: cURL, `mod_ssl` of the Apache HTTP server, Lynx, Mutt, and Wget (see Table 4). For the libraries, we primarily focused on the source files implementing core functionality (*e.g.,* src, lib sub-directories, *etc.*) as opposed to the test files, as detecting bugs in the test code may not be a high priority. All the applications but the HTTP server were small enough to run EPEx on the entire program, although it eventually only produced results for the source files that used the OpenSSL library. For Httpd we only checked `mod_ssl`. The second column of Table 4 shows the modules investigated by EPEx for each tested library.

For each library, we generate a call graph using a tool named `GNU cflow` [1]. From the call graph, we choose top functions that are frequently called by other functions within the same library. Note that here we did not distinguish between internal library functions and library functions exposed to the outer world as APIs. We further filtered out functions based on their return types—functions returning integer, boolean, and pointers are chosen because Clang's symbolic analysis engine can currently only handle these types. In addition, we only selected those functions that can fail and return at least one error value. For the applications, we tested all the OpenSSL APIs that the applications are using. We found such APIs by simply using `grep`. Further, we only chose those APIs for which documentations are available, and the APIs that could return errors as integers, booleans or pointers. The third column of Table 4 shows the number of functions tested for a studied program.

Table 4: Study subjects

Projects	Modules	#Functions tested	#Call sites	#Error paths
OpenSSL v1.0.1p	ssl, crypto	46	507	3171
GnuTLS v3.3.17.1	src, lib	50	877	3507
mbedTLS v1.3.11	library	37	505	1621
wolfSSL v3.6.0	wolfcrypt, src	20	138	418
curl v7.47.0	all	17	49	2012
httpd v2.4.18	mod_ssl	14	86	4368
lynx v2.8.8	all	3	23	494
mutt v1.4.2.3	all	3	9	5
wget v1.17.1	all	5	13	2409
Total		195	2207	18005

5.2 General findings

We evaluated EPEx on 195 unique program-API function pairs from 2207 call-sites, and covered 18005 error paths (see Table 4). EPEx found 102 new error-handling bugs from 4 SSL/TLS libraries and 5 applications: 48 bugs in OpenSSL, 23 bugs in GnuTLS, 19 bugs in mbedTLS, and 0 bugs in wolfSSL, 2 in cURL, 7 in httpd, 1 in Lynx, 2 in Mutt, and 0 in Wget (see Table 5). We evaluate EPEx's performance after completion of Step-II and Step-III separately. Since we are using recent versions of real code, and finding all potential bugs in such code is an extremely difficult problem, we do not have a ground truth for bugs against which to compare the reports. Also, EPEx is not designed to detect all types of error handling bugs. For this paper, we define a bug to be any error path whose output behavior is identical to that of a non-error path, *e.g.,* no logging takes place and the same values as in the non-error paths are propagated upwards through all channels. Thus, for counting false positives and negatives, we do not consider bugs due to incomplete handling, for example, where failures are only logged, but the required cleanup is missing. Table 5 presents the detailed result. After Step-II, EPEx reported 154 bugs in the library code and 29 bugs in the application code. After a manual investigation, we found 61 of them to be false positives. Step-III reduced this false positive to 28 out of 130 reported bugs (106 in library and 24 in application code). Thus, overall, EPEx

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

detected bugs with 84% precision in the library code and 50% precision in the application code with an overall precision of 78%.

In general, measuring false negatives for static analysis tools is non-trivial as it is hard to be confident about the number of bugs present in the code at any given point of time. However, for the sake of completeness, we checked false negatives by randomly selecting 100 cases at the end of Step-II, where EPEx confirmed that error handling was indeed implemented correctly. We did not find any false negatives in any of those examples, *i.e.* we did not find any bugs that were filtered out at Step-II. However, after Step-III's optimization, among the bugs that did pass Step-II, we found 15 and 5 false negatives in Library and Application code respectively. Thus, the overall recall of EPEx was approximately 83%.

Table 5: Evaluation of EPEx

	Step II		Step III		Summary	
	Reported Bugs	False +ve	Reported Bugs	False +ve	True Bugs	Prec-ision
Library						
OpenSSL	51	2	50	2	48	0.96
GnuTLS	41	15	25	1	23	0.96
mbedTLS	35	16	21	2	19	0.90
WolfSSL	27	7	10	2	0	0.80
Total	154	40	106	16	90	0.84
Application						
Curl	6	2	4	2	2	0.5
Httpd	13	6	13	6	7	0.53
Lynx	5	2	3	2	1	0.33
Mutt	3	1	3	1	2	0.67
Wget	2	1	1	1	0	0.00
Total	29	12	24	12	12	0.50
Grand Total	183	52	130	28	102	0.78

In general, EPEx performs better for libraries than applications. There are three main reasons behind this: (i) unlike libraries, applications' error handling behavior is heavily dependent on their configuration parameters. For example, users can configure the applications to ignore certain errors. EPEx currently cannot differentiate between paths that have different values for these configuration parameters; (ii) Applications are more likely to use complex data types (*e.g.,* error code is embedded within an object) for propagating errors than libraries that are not currently supported by EPEx; and (iii) Applications prioritize user experience over internal consistency, so if the error is recoverable, they will attempt to use a fallback non-error value instead. However, none of these are fundamental limitations of our approach. EPEx can be enhanced to support such cases and improve its accuracy for applications too.

In the following section, we discuss the nature of the detected bugs and the vulnerabilities caused by them in detail with code examples from libraries in Section 5.3 and from applications in Section 5.4. All the described bugs have been reported to the developers, who, for almost all cases, have confirmed and agreed that they should be fixed.

5.3 Bugs in libraries

From the four SSL/TLS libraries that we tested, we describe seven selected examples. They arise due to various reasons including ignoring error codes, missing checks for certain error codes, checking with a wrong value, and propagating incorrect error values upstream. These bugs affect different modules of the SSL/TLS implementations, and at least 42 of them result in critical security vulnerabilities by completely breaking the security guarantees of SSL/TLS, as discussed below.

Incorrect random number generation. EPEx found 21 instances in OpenSSL where callers of the function `RAND_pseudo_bytes` do not implement the error handling correctly. We provide two such examples below. `RAND_pseudo_bytes` returns cryptographically secure pseudo-random bytes of the desired length. An error-free execution of this function is extremely important to OpenSSL as the security guarantees of all cryptographic primitives implemented in OpenSSL depend on the unpredictability of the random numbers that `RAND_pseudo_bytes` returns. The cryptographically secure random numbers, as returned by `RAND_pseudo_bytes`, are used for diverse purposes by different pieces of OpenSSL code, *e.g.,* creating initialization vectors (IVs), non-repeatable nonces, cryptographic keys. In case of a failure, `RAND_pseudo_bytes` returns 0 or -1 to indicate any error that makes the generated random numbers insecure and unsuitable for cryptographic purposes.

Example 1.

```
int PEM_ASN1_write_bio(...)
{
  int ret = 0;
  ...
  /* Generate a salt */
  if (RAND_pseudo_bytes(iv, enc->iv_len) <
     0)
    goto err;
  ...
  ret = 1;
err:
  OPENSSL_cleanse(iv, sizeof(iv));
  ...
  return ret;
}
```

Example 2.

```
int bnrand(...)
{
  int ret = 0;
  ...
  if (RAND_pseudo_bytes(buf, bytes) == -1)
    goto err;
  ...
  ret = 1;
err:
  ...
  return ret;
}
```

The above code shows two examples of incorrect error handling at different call-sites of `RAND_pseudo_bytes` in OpenSSL code. In **Example 1**, function `PEM_ASN1_write_bio` checks only if < 0, but not if $= 0$. In **Example 2**, function `bnrand` only checks for the -1 value but not for the 0 value. `bnrand` is used by all bignumber routines, which are in turn used for key generation by many cryptographic implementations like RSA. These bugs completely break the security guarantees of any cryptographic implementations (RSA, AES, *etc.*) and security protocol implementations (*e.g.,* SSL/TLS, SMIME, *etc.*) in OpenSSL that use such buggy code for random number generation. An attacker can leverage these bugs to launch man-in-the-middle attacks on SSL/TLS connections setup using OpenSSL.

The sources of errors in random number generation functions are diverse and depend on the underlying random number generation mechanism (see Listing 3 in the Appendix for a sample random number generation implementation in OpenSSL). For example, an error can occur due to memory allocation failures or module loading errors. Note that some of these failures can be triggered remotely by an attacker through denial-of-service attacks. Thus, if the errors are not handled correctly, an attacker can break the security guarantees of different cryptographic primitives by making them use insecure random numbers. We have received confirmation from OpenSSL developers about these issues.

Incorrect cryptography implementations. Here, we exhibit an example from OpenSSL demonstrating an error handling bug that EPEx found in the implementation of a cryptographic algorithm.

Insecure SRP keys. EPEx found that the function `SRP_Calc_server_key`, which is part of the SRP (Secure Remote Password) module in OpenSSL, contains an error handling bug while calling `BN_mod_exp`, as shown in the code below.

```
1  BIGNUM *SRP_Calc_server_key(BIGNUM *A,
        BIGNUM *v,
2       BIGNUM *u, BIGNUM *b, BIGNUM *N)
3  {
4    BIGNUM *tmp = NULL, *S = NULL;
5    BN_CTX *bn_ctx;
6    ...
7    if ((bn_ctx = BN_CTX_new()) == NULL ||
8        (tmp = BN_new()) == NULL ||
9        (S = BN_new()) == NULL)
10     goto err;
11
12   if (!BN_mod_exp(tmp, v, u, N, bn_ctx))
13     goto err;
14     ...
15 err:
16   BN_CTX_free(bn_ctx);
17   BN_clear_free(tmp);
18   return S;
19 }
```

The `BN_mod_exp` function takes four big numbers (arbitrary-precision integers) tmp, v, u, N, and a context bn_ctx as input. It then computes v raised to the u^{th} power modulo N and stores it in tmp (*i.e.* $tmp = v^u\%N$). However, `BN_mod_exp` can fail for different reasons including memory allocation failures. It returns 0 to indicate any such error. The call-site of `BN_mod_exp` (line 12), in fact, correctly checks for such an error and jumps to the error handling code at line 15. The error handling code frees the resources and returns S (line 18). However, S is guaranteed to be not NULL at this point as it has been allocated by calling a `BN_new` function at line 9. This leads `SRP_Calc_server_key` to return an uninitialized big number S. Thus, the functions upstream will not know about the error returned by `BN_mod_exp`, as `SRP_Calc_server_key` is supposed to return a NULL pointer in case of an error. This leads to silent data corruption that can be leveraged to break the security guarantees of the SRP protocol.

Incorrect X.509 certificate revocation. Here we cite two examples from mbedTLS and GnuTLS respectively showing different types of incorrect error handling bugs in implementations of two different X509 certificate revocation mechanisms: CRL (Certificate Revocation List) and OCSP (Online Certificate Status Protocol).

CRL parsing discrepancy. In mbedTLS, EPEx found that `x509_crl_get_version`, which retrieves the version of a X509 certificate revocation list, has an error handling bug while calling function `asn1_get_int` (line 7 in the code below). Function `asn1_get_int` reads an integer from an ASN1 file. It returns different negative values to indicate different errors. In case of a malformed CRL (Certificate Revocation List) file, it returns `POLARSSL_ERR_ASN1_UNEXPECTED_TAG` error value. In case of such an error, line 9-13 treats the CRL version as 0 (version 1). Thus, mbedTLS parses a malformed CRL file as version 1 certificate. However, other SSL implementations (*e.g.,* OpenSSL) treat these errors differently and parse it as a version 2 certificate. We are currently discussing the exploitability of this inconsistency with the developers.

```
1  int x509_crl_get_version(unsigned char **p,
2                           const unsigned char
                                *end,
3                           int *ver )
4  {
5  int ret;
6
7  if((ret = asn1_get_int(p, end, ver))!= 0)
8  {
9    if( ret ==
         POLARSSL_ERR_ASN1_UNEXPECTED_TAG )
10   {
11     *ver = 0;
12     return(0);
13   }
14   return(POLARSSL_ERR_X509_INVALID_VERSION +
         ret);
15 }
16 return( 0 );}
```

Incorrect OCSP timestamps. GnuTLS function `gnutls_ocsp_resp_get_single` is used to read the timestamp of an Online Certificate Status Protocol (OCSP) message along with other information. EPEx found an error handling bug in it while calling function `asn1_read_value`, as shown in line 5 in the following code. `asn1_read_value` reads the value of an ASN1 tag. It returns an error while failing to read the tag correctly. `gnutls_ocsp_resp_get_single` correctly checks for the error conditions (line 6), but instead of returning an error value, simply sets the `this_update` parameter to -1. However, further upstream, in `check_ocsp_response`, which calls the function `gnutls_ocsp_resp_get_single` (line 16), the corresponding variable `vtime` is not checked for an error value; only the return value is checked, but that is a non-error value. Further down the function, at line 22, `vtime` is used to check whether the message is too old. However, in the error path, since vtime is set to -1 from line 7, the left-hand side of the conditional check will always be a positive number. Due to a large value of the variable `now` (representing current time), the conditional check will always be positive, and result in categorizing all messages to be over the OCSP validity threshold irrespective of their actual timestamp. Depending on the configuration of GnuTLS, this may result in ignoring new OCSP responses containing information on recently revoked certificates.

```
1  int
2  gnutls_ocsp_resp_get_single (..., time_t *
       this_update)
3  {
4    ...
5    ret = asn1_read_value(resp->basicresp,
         name, ttime, &len);
6    if (ret != ASN1_SUCCESS) {
7      *this_update = (time_t) (-1);
8    }
9    ...
10   return GNUTLS_SUCCESS;
11 }
12
13 static int
14 check_ocsp_response(...)
15 { ...
16   ret = gnutls_ocsp_resp_get_single(...,&
         vtime);
17
18   if (ret < 0) {
19     ...
20   }
21
22   if (now - vtime >
         MAX_OCSP_VALIDITY_SECS) {
23     ...
24   }
25   ...
26 }
```

Incorrect protocol implementations. Here we show two examples from OpenSSL where error handling bugs occur in implementations of two different protocols: Secure/Multipurpose Internet Mail Extensions (S/MIME) and Datagram Transport Layer Security (DTLS).

Faulty parsing of X.509 certificates in S/MIME. EPEx found that the OpenSSL function `cms_SignerIdentifier_cert_cmp` does not check the return value returned by function `X509_get_serialNumber`, as shown in the code below. This code is part of the OpenSSL code that handles S/MIME v3.1 mail. Here, the error point (see line 6) is at the call site of `X509_get_serialNumber`, which returns a pointer to the `ASN_INTEGER` object that contains the serial number of the input x509 certificate. However, in case of a malformed certificate missing the serial number, `X509_get_serialNumber` returns NULL to indicate an error. In this case, the caller function `cms_SignerIdentifier_cert_cmp` does not check for an error and passes the return value directly to `ASN1_INTEGER_cmp`. Thus, the second argument of `ASN1_INTEGER_cmp` (y in line number 12) is set to NULL, in the case of an error. At line 16, `ASN1_INTEGER_cmp` tries to read $y->type$ and causes a NULL pointer dereference and results in a crash. This can be exploited by a remote attacker to cause a denial of service attack by supplying malformed X.509 certificates. This issue was confirmed by the corresponding developers but they believe that that it is up to the application programmer to ensure that the input certificate is properly initialized.

```
1  int cms_SignerIdentifier_cert_cmp(
       CMS_SignerIdentifier *sid, X509 *cert)
2  {
3    if (sid->type ==
         CMS_SIGNERINFO_ISSUER_SERIAL) {
4      ...
5      return ASN1_INTEGER_cmp(serialNumber,
6            X509_get_serialNumber(cert));
7    }
8    ...
9    return -1;
10 }
11
12 int ASN1_INTEGER_cmp(const ASN1_INTEGER *x,
       const ASN1_INTEGER *y)
13 {
14   int neg = x->type & V_ASN1_NEG;
15   /* Compare signs */
16   if (neg != (y->type & V_ASN1_NEG)) {
17     ...
18   }
19   ...
20 }
```

Faulty encoding of X.509 certificates in DTLS. EPEx found that the function `dtls1_add_cert_to_buf` that reads a certificate from DTLS [2] handshake message contains an error handling bug while calling `i2d_X509` (line 8 in the code below). Function `i2d_X509` encodes the input structure pointed to by x into DER format. It returns a negative value to indicate an error, otherwise it returns the length of the encoded data. Here, the caller code (line 8) does not check for error cases, and thus gives no indication of whether the read data was valid or not. In case of an error, this will lead to incorrect results and silent data corruption.

[2]Datagram Transport Layer Security: a protocol in SSL/TLS family

```
static int dtls1_add_cert_to_buf(BUF_MEM *
    buf, unsigned long *l, X509 *x)
{
  int n;
  unsigned char *p;
  ...
  p = (unsigned char *)&(buf->data[*l]);
  l2n3(n, p);
  i2d_X509(x, &p);
  *l += n + 3;

  return 1;
}
```

5.4 Bugs in applications

Beside libraries, we used EPEx to evaluate error handling implementations in application software that use SSL/TLS library APIs. We have performed tests on 5 programs that use the OpenSSL library: cURL [3], httpd [4], Lynx [5], Mutt [6], and Wget [7]. Our error specification included 29 OpenSSL APIs that are used by at least one of these applications. As the results show in Table 5, even though EPEx is not as accurate for applications as for libraries, and we had to discard 2 alerts because the callers did not follow the error protocol, it still found 12 real bugs.

In case of applications, unlike libraries, Step-III of EPEx was able to compare error behavior across multiple applications and libraries that use the same API. This allowed us to detect bugs in the cases where an application developer has consistently made error handling mistakes for an API function as long as other applications using the same API function are correctly handling the errors.

In terms of security effects, the bugs that we found range from causing serious security vulnerabilities to denial-of-service attacks. We found 2 bugs in cURL random number generation modules that can be exploited to make cURL vulnerable to man-in-the-middle attacks. We also found 4 bugs in httpd, Mutt, and Lynx that will cause denial-of-service attacks. The other bugs that we found mostly lead to resource leakage. We provide one example of the cURL random number generation bug below.

cURL ignores the return value of the Pseudorandom number generator `RAND_bytes`. In case of an error, `RAND_bytes` will return an output buffer with non-random values. In that case cURL will use it for generating SSL session keys and other secrets. Note that a failure in `RAND_bytes` can be induced by an attacker by launching a denial of service attack and causing memory allocation failures, file descriptor exhaustion, *etc.*

[3]http://curl.haxx.se/
[4]https://httpd.apache.org/
[5]http://lynx.invisible-island.net/
[6]http://www.mutt.org/
[7]https://www.gnu.org/software/wget/wget.html

```
int Curl_ossl_random(struct SessionHandle *
    data, unsigned char *entropy,
                     size_t length)
{
...
  RAND_bytes(entropy, curlx_uztosi(length));
  return 0; /* 0 as in no error */
}
```

5.5 Checking for correct error propagation

Besides producing bugs, EPEx also confirms whether a function call's error handling code correctly takes care of all possible errors. Note that EPEx only checks whether error values are propagated upstream but does not check whether other error handling tasks (e.g., freeing up all acquired resources) have been implemented correctly.

The following example shows an instance where EPEx confirmed that the error codes are correctly propagated by the error handling code. This piece of code is from GnuTLS 3.3.17.1 and contains the fix for the CVE-2014-92 vulnerability that we described in the introduction (Listing 1). EPEx confirmed that the fix indeed correctly handles the error case.

Besides fixing the bug, the updated version of the code has also been slightly refactored and reorganized as shown below. Code in ~~red~~ highlights the bug, while **green** shows the fix. The return type of function `check_if_ca` has been updated to `bool`, where returning false (0) indicates an error (see line 1 and 10). The caller function verify_crt is correctly checking $\neq 1$ (*i.e.* True) at line 17 to handle the error case.

```
1  ~~int~~ bool
2  check_if_ca(...)
3  { ...
4    if (ret < 0) {
5          gnutls_assert();
6          goto fail;
7      }
8
9  fail:
10   result = 0;
11   ...
12 return result;
13 }
14
15 bool verify_crt(...)
16 { ...
17   if (check_if_ca(...) ~~==0~~ != 1) {
18     result = 0;
19     goto cleanup;
20   }
21   ...
22 cleanup:
23 ...
24 return result;
25
26 }
```

We also used EPEx to successfully check the fixes for other CVEs mentioned in Section 1 (`CVE-2015-0208`, `CVE-2015-0288`, `CVE-2015-0285`, and `CVE--2015-0292`).

5.6 Imprecision in EPEx Analysis

The 130 potential bugs reported by EPEx includes 28 false positives and incorrectly excludes 20. In the libraries, most of the false positives appeared due to the limitations of underlying Clang symbolic analysis engine. The interprocedural analysis supported by Clang's symbolic analysis engine is currently limited to the functions defined within an input source file or functions included in the file through header files. Therefore, the symbolic analyzer is not able to gather correct path conditions and return values for the functions defined in other source files. For example, in the code below, EPEx reported error since the return value of `X509_get_serialNumber` is not checked at line 5. However, inside the callee, `ASN1_STRING_dup`, the error condition is checked at line 17 and the NULL value is returned. This return value (`serial`) is further checked at line 6. Since, `ASN1_STRING_dup` is implemented in a different file, EPEx could not infer that the `ASN1_STRING_dup` call in line 5 will always return NULL if `X509_get_serialNumber` returns an error. Note that if the pattern of not checking error for `X509_get_serialNumber` calls were consistent across all call-sites, EPEx would not have reported this false positive due to Step-III in Section 3).

```
1  AUTHORITY_KEYID *v2i_AUTHORITY_KEYID(...)
2  {
3    ...
4    serial = ASN1_INTEGER_dup(
5        X509_get_serialNumber(cert));
6    if (!isname || !serial) {
7          X509V3err(...);
8          goto err;
9    }
10   ...
11 }
12
13 ASN1_STRING *ASN1_STRING_dup(
14     const ASN1_STRING *str)
15 {
16     ASN1_STRING *ret;
17     if (!str)
18         return NULL;
19     ...
20 }
```

EPEx performed the worst in wolfSSL, mostly due to confusion arising from compile-time configuration settings affecting the function `mp_init`. EPEx raised 8 alerts for the function, but after contacting the developers, we learned that the corresponding functions can be configured, at compilation time, to be either fallible or infallible. All the reported call sites were only compiled if the functions were configured to be infallible. Therefore, our error specifications should not have marked these functions as fallible. On the other hand, in the applications, the most frequent causes are 5 instances of fallbacks, which are characteristic of applications. Still, missed checks in external functions are the second most frequent cause, at 3 cases. The remaining causes are alternative error propagation channels, and deliberate disregard for the error, due to either a conscious choice of the programmer, or a configuration parameter, as mentioned in Section 5.2.

Given the false positives due to checks by external functions, a natural solution would be to have all functions validate their input. While this is a good practice for library programmers, application programmers should not depend on functions, whose implementation they often do not control, to follow this practice. For debugging purposes, it would appear that the function receiving the invalid return value is at fault. Moreover, not all functions can cleanly handle invalid input. Comparison functions such as `ASN1_INTEGER_cmp` only return non-error values, so the only safe response would be to terminate the program, which is a drastic action that can easily be averted by checking the parameters in the first place.

5.7 Performance analysis

EPEx is integrated with the test project's building procedure through the Clang framework. We ran all our tests on Linux servers with 4 Intel Xeon 2.67GHz processors and 100 GB of memory, The following table shows the performance numbers. EPEx's execution time is comparable to that of other built-in, simple checkers in Clang (*e.g.,* division-by-zero) as shown in the table below.

	Regular build	Division-by-zero in-built checker	EPEx checker
wolfSSL	0.05m	3.08m	2.68m
mbedTLS	0.67m	3.72m	2.83m
GnuTLS	1.85m	13.28m	12.82m
OpenSSL	8.25m	186.9m	132.33m
cURL	0.18m	13.96m	12.95m
httpd	0.04m	4.68m	4.51m
Lynx	0.55m	71.35m	71.73m
Mutt	0.10m	13.03m	13.12m
Wget	0.03m	5.63m	5.66m

6 Related work

6.1 Automated detection of error handling bugs

Rubio-González *et al.* [45, 21] detected incorrect error handling code in the Linux file system using a static control and data-flow analysis. Their technique was designed to detect bugs caused by faulty code that either overwrite or ignore error values. In addition to these two cases, we check whether appropriate error values are propagated upstream as per global error protocol of the analyzed program. We use module-specific error specifications as opposed to hard coded error values like -EIO, -ENOMEM, *etc.* used by Rubio-González *et al.* This helps us in reducing the number of false positives significantly; for instance, unlike [45, 21], we do not report a bug when an error value is over-written by another error value that conforms to the global error protocol. Our usage of symbolic analysis further minimizes false positives as symbolic analysis, unlike the data-flow analysis used in [45, 21], can distinguish between feasible and infeasible paths.

Acharya *et al.* [1] automatically inferred error handling specifications of APIs by mining static traces of their run-time behaviors. Then, for a different subject system, they found several bugs in error handling code that do not obey the inferred specifications. The static traces were generated by MOPS [11] that handles only control dependencies and minimal data-dependencies. As observed by Acharya *et al.*, lack of extensive data-dependency support (*e.g.,* pointer analysis, aliasing, *etc.*) introduced imprecision in their results. By contrast, our symbolic execution engine with extensive memory modeling support minimizes such issues. Further, to identify error handling code blocks corresponding to an API function, Acharya *et al.* leveraged the presence of conditional checks on the API function's return value and/or ERRNO flag. They assumed that if such a conditional check leads to a return or exit call, then it is responsible for handling the error case. Such assumption may lead to false positives where conditional checks are performed on non-error cases. Also, as noted by Acharya *et al.*, for functions that can return multiple non-error values, they cannot distinguish them from error cases. By contrast, we create our error specifications from the program documentation and thus they do not suffer from such discrepancies.

Lawall *et al.* [31] used Coccinelle, a program matching and transformation engine, to find missing error checks in OpenSSL. By contrast, we not only look for error checks but also ensure that the error is indeed handled correctly. This allows us to find a significantly larger class of error handling problems. Also, unlike our approach, Lawall *et al.*'s method suffers from an extremely high false positive rate.

Several other approaches to automatically detect error/exception handling bugs have been proposed for Java programs [52, 53, 44, 8, 54]. However, since the error handling mechanism is quite different in Java than C (*e.g.,* the `try-catch-final` construct is not supported in C), these solutions are not directly applicable to C code.

Static analysis has been used extensively in the past to find missing checks on security critical objects [48, 57, 49]. However, none of these tools can detect missing/incorrect error handling checks. Complementary to our work, and other static approaches, dynamic analysis methods have been developed to discover the practical effects of error handling bugs, although they do so at the cost of lower coverage of error paths, as well as unknown failure modes. Fault injection frameworks such as LFI bypass the problem of the unlikelihood of errors by injecting failures directly into fallible functions. LFI includes a module for automatically inferring error specifications, although it is not usable in our case, since static analysis requires explicitly identifying error and non-error values, and not just differentiate between them [34].

6.2 Symbolic execution

The idea of symbolic execution was initially proposed by King *et al.* [29]. Concolic execution is a recent variant of symbolic execution where concrete inputs guide the execution [19, 10, 47]. Such techniques have been used in several recent projects for automatically finding security bugs [27, 46, 22, 20].

KLEE [9], by Cadar *et al.*, is a symbolic execution engine that has been successfully used to find several bugs in UNIX coreutils automatically. UC-KLEE [40], which integrates KLEE and lazy initialization [26], applies more comprehensive symbolic execution over a bounded exhaustive execution space to check for code equivalence; UC-KLEE has also been effective in finding bugs in different tools, including itself. Recently, Ramos *et al.* applied UC-KLEE to find two denial-of-service vulnerabilities in OpenSSL [41].

SAGE, by Godefroid *et al.* [20], uses a given set of inputs as seeds, builds symbolic path conditions by monitoring their execution paths, and systematically negates these path conditions to explore their neighboring paths, and generate input for fuzzing. SAGE has been successfully used to find several bugs (including security bugs) in different Windows applications like media players and image processors. SAGE also checks for error handling bugs, but only errors from user inputs, and not environmental failures, which are unlikely to appear when only user input is fuzzed.

Ardilla, by Kiezun et al. [27], automates testing of Web applications for SQL injection and cross-site scripting attacks by generating test inputs using dynamic taint analysis that leverages concolic execution and mutates the inputs using a library of attack patterns.

Existing symbolic execution tools are not well suited for finding error handling bugs for two primary reasons: (i) The existing symbolic execution tools depend on obvious faulty behaviors like crashes, assertion failures, *etc.* for detecting bugs. A large number of error handling bugs are completely silent and do not exhibit any such behavior. (ii) As the number of paths through any reasonable sized program is very large, all symbolic execution tools can only explore a fraction of those paths. The effects of most non-silent error handling bugs show up much further downstream from their source. An off-the-shelf symbolic execution tool can only detect such cases if it reaches that point. By contrast, our algorithm for identifying and exploring error paths enables EPEX to detect completely silent and non-silent error handling bugs at their sources. This makes it easy for the developers to understand and fix these bugs.

6.3 Security of SSL/TLS implementations

Several security vulnerabilities have been found over the years in both SSL/TLS implementations and protocol specifications [15, 43, 2, 5, 7, 4, 3]. We briefly summarize some of these issues below. A detailed survey of SSL/TLS vulnerabilities can be found in [13].

Multiple vulnerabilities in certification validation implementations, a key part of the SSL/TLS protocol, were reported by Moxie Marlinspike [38, 37, 36, 35]. Similar bugs have been recently discovered in the SSL implementation on Apple iOS [24]. Another certificate validation bug ("goto fail") was reported in Mac OS and iOS [30] due to an extra goto statement in the implementation of the SSL/TLS handshake protocol. The affected code did not ensure that the key used to sign the server's key exchange matches the key in the certificate presented by the server. This flaw made the SSL/TLS implementations in MacOS and iOS vulnerable to active Man-In-The-Middle (MITM) attackers. This bug was caused by unintended overlapping of some parts of a non-error path and an error path. However, this is not an error handling bug like the ones we found in this paper.

Hash collisions [50] and certificate parsing discrepancies between certificate authorities (CAs) and Web browsers [25] can trick a CA into issuing a valid certificate with the wrong subject name or even a valid intermediate CA certificate. This allows an attacker to launch a successful MITM attack against any arbitrary SSL/TLS connection.

Georgiev *et al.* [18] showed that incorrect usage of SSL/TLS APIs results in a large number of certificate validation vulnerabilities in different applications. Fahl *et al.* [17] analyzed incorrect SSL/TLS API usage for Android applications. Brubaker *et al.* [6] designed Frankencerts, a mechanism for generating synthetic X.509 certificates based on a set of publicly available seed certificates for testing the certificate validation component of SSL/TLS libraries. They performed differential testing on multiple SSL/TLS libraries using Frankencerts and found several new security vulnerabilities. Chen *et al.* [12] improved the coverage and efficiency of Brubaker *et al.*'s technique by diversifying the seed certificate selection process using Markov Chain Monte Carlo (MCMC) sampling. However, all these techniques are black-box methods that only focus on the certificate validation part of the SSL/TLS implementations. By contrast, our white-box analysis is tailored to look for flawed error handling code in any sequential C code.

Flawed pseudo-random number generation can produce insecure SSL/TLS keys that can be easily compromised [32, 23]. We have also reported several bugs involving pseudo-random number generator functions in this paper, although their origins are completely different, *i.e.*, unlike [32, 23], they are caused by incorrect error handling.

7 Future work

Automated inference of error specifications. One limitation of our current implementation of EPEx is that it requires the input error specifications to be created manually by the user. Automatically generating the error specifications will significantly improve EPEx's usability. One possible way to automatically infer the error specifications is to identify and compare the path constraints imposed along the error paths (*i.e.*, the paths along which a function can fail and return errors) across different callsites of the same function. However, in order to do so, the error paths must first be automatically identified. This leads to a chicken-and-egg problem as the current prototype of EPEx uses the input error specifications to identify the error paths.

To solve this problem, we plan to leverage different path features that can distinguish the error paths from non-error paths. For example, error paths are often more likely to return constant values than non-error paths [33]. Error paths are also more likely to call functions like exit (with a non-zero argument) than regular code for early termination. Further, since errors invalidate the rest of the computation, the lengths of the error paths (*i.e.*, number of program statements) might be, on average, shorter than the non-error paths. An interesting direction for future research will be to train a supervised machine learning algorithm like Support Vector Machines (SVMs) [14] for identifying error paths using such different path features. The supervised machine learning algorithm can be trained using a small set of error and non-error paths identified through manually created error specifications. The resulting machine learning model can then be used to automatically identify different error paths and infer error specifications by comparing the corresponding path constraints.

Automatically generating bug fixes. As error-handling code is often repetitive and cumbersome to implement, it might be difficult for developers to keep up with EPEx and fix all the reported bugs manually. Moreover, manual fixes introduced by a developer might also be buggy and thus may introduce new error handling bugs. In order to avoid such issues, we plan to automatically generate candidate patches to fix the error handling bugs reported by EPEx. Several recent projects [55, 39, 28] have successfully generated patches for fixing different types of bugs. Their main approach is dependent on existing test suites—they first generate candidate patches by modifying existing code and then validate the patches using existing test cases. While this generic approach can be applied in our setting, we cannot use the existing schemes as error handling bugs are, in general, hard to detect through existing test cases. Also, these approaches typically focus on bug fixes involving only one or two lines of code changes. However, the error handling bugs are not necessarily limited to such small fixes. Solving these issues will be an interesting direction for future work.

8 Conclusion

In this paper, we presented EPEx, a new algorithm and a tool that automatically explores error paths and finds error handling bugs in sequential C code. We showed that EPEx can efficiently find error handling bugs in different open-source SSL/TLS libraries and applications with few false positives; many of these detected bugs lead to critical security vulnerabilities. We also demonstrate that EPEx could also be useful to the developers for checking error handling code.

9 Acknowledgments

We would like to thank Ben Livshits, the shepherd of this paper, and the anonymous reviewers whose suggestions have improved the presentation of our work. We would also like to thank David Evans for his feedback on an earlier draft of this paper. This work is sponsored in part by Air Force Office of Scientific Research (AFOSR) grant FA9550-12-1-0162. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFOSR.

References

[1] M. Acharya and T. Xie. Mining API Error-Handling Specifications from Source Code. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, 2009.

[2] N. AlFardan and K. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[3] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and J. Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[4] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P. Strub. Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[5] D. Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In *International Cryptology Conference (CRYPTO)*, 1996.

[6] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in ssl/tls implementations. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[7] D. Brumley and D. Boneh. Remote timing attacks are practical. In *USENIX Security Symposium*, 2003.

[8] R. Buse and W. Weimer. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2008.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[10] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. In *International SPIN Workshop on Model Checking of Software (SPIN)*, 2005.

[11] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security (CCS)*, 2002.

[12] Y. Chen and Z. Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *ACM SIGSOFT International Symposium on the Foundations of Software (FSE)*, 2015.

[13] J. Clark and P. van Oorschot. SoK: SSL and HTTPS: Revisiting past challenges and evaluating certificate trust model enhancements. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[14] C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[15] T. Duong and J. Rizzo. Here come the ⊕ ninjas. `http://nerdoholic.org/uploads/dergln/beast_part2/ssl_jun21.pdf`, 2011.

[16] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles (SOSP)*, 2001.

[17] S. Fahl, M. Harbach, T. Muders, and M. Smith. Why Eve and Mallory love Android: An analysis of SSl (in)security on Android. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[18] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security (CCS)*, 2012.

[19] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2005.

[20] P. Godefroid, M. Levin, and D. Molnar. Automated whitebox fuzz testing. In *Network & Distributed System Security Symposium (NDSS)*, 2008.

[21] H. Gunawi, C. Rubio-González, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and B. Liblit. EIO: Error handling is occasionally correct. In *USENIX Conference on File and Storage Technologies (FAST)*, 2008.

[22] W. Halfond, S. Anand, and A. Orso. Precise interface identification to improve testing and analysis of web applications. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[23] N. Heninger, Z. Durumeric, E. Wustrow, and A. Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *USENIX Security Symposium*, 2012.

[24] CVE-2011-0228. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-0228`, 2011.

[25] D. Kaminsky, M. Patterson, and L. Sassaman. PKI layer cake: New collision attacks against the global X.509 infrastructure. In

FC, 2010.

[26] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003.

[27] A. Kiezun, P. Guo, K. Jayaraman, and M. Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *International Conference on Software Engineering (ICSE)*, 2009.

[28] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE)*, 2013.

[29] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[30] A. Langley. Apple's SSL/TLS bug. `https://www.imperialviolet.org/2014/02/22/applebug.html`, 2014.

[31] J. Lawall, B. Laurie, R. Hansen, N. Palix, and G. Muller. Finding error handling bugs in openssl using coccinelle. In *European Dependable Computing Conference (EDCC)*, 2010.

[32] A. Lenstra, J. Hughes, M. Augier, J. Bos, T. Kleinjung, and C. Wachter. Ron was wrong, Whit is right. `http://eprint.iacr.org/2012/064`, 2012.

[33] P. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(4), 2011.

[34] P. D. Marinescu, R. Banabic, and G. Candea. An extensible technique for high-precision testing of recovery code. In *USENIX Annual Technical Conference*, 2010.

[35] M. Marlinspike. IE SSL vulnerability. `http://www.thoughtcrime.org/ie-ssl-chain.txt`, 2002.

[36] M. Marlinspike. More tricks for defeating SSL in practice. DEFCON, 2009.

[37] M. Marlinspike. New tricks for defeating SSL in practice. Black Hat DC, 2009.

[38] M. Marlinspike. Null prefix attacks against SSL/TLS certificates. `http://www.thoughtcrime.org/papers/null-prefix-attacks.pdf`, 2009.

[39] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[40] D. Ramos and D. Engler. Practical, low-effort equivalence verification of real code. In *International Conference on Computer-Aided Verification (CAV)*, 2011.

[41] D. Ramos and D. Engler. Under-constrained symbolic execution: correctness checking for real code. In *USENIX Security Symposium*, 2015.

[42] Checker developer manual. `http://clang-analyzer.llvm.org/checker_dev_manual.html`.

[43] J. Rizzo and T. Duong. The CRIME attack. In *Ekoparty*, 2012.

[44] M. Robillard and G. Murphy. Analyzing exception flow in Java programs. In *ACM SIGSOFT International Symposium on the Foundations of Software (FSE)*, 1999.

[45] C. Rubio-González, H. Gunawi, B. Liblit, R. Arpaci-Dusseau, and A. Arpaci-Dusseau. Error propagation analysis for file systems. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2009.

[46] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *IEEE Symposium on Security and Privacy (S&P)*, 2010.

[47] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ACM SIGSOFT International Symposium on the Foundations of Software (FSE)*, 2005.

[48] S. Son, K. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. In *International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2011.

[49] V. Srivastava, M. Bond, K. McKinley, and V. Shmatikov. A security policy oracle: Detecting security holes using multiple API implementations. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.

[50] M. Stevens, A. Sotirov, J. Appelbaum, A. Lenstra, D. Molnar, D. Osvik, and B. Weger. Short chosen-prefix collisions for MD5 and the creation of a rogue CA certificate. In *International Cryptology Conference (CRYPTO)*, 2009.

[51] The Apache Software Foundation. Apache portable runtime: Error codes. Available at `https://apr.apache.org/docs/apr/1.4/group__apr__errno.html`, 2011.

[52] W. Weimer and G. Necula. Finding and preventing run-time error handling mistakes. In *International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2004.

[53] W. Weimer and G. Necula. Mining Temporal Specifications for Error Detection. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[54] W. Weimer and G. Necula. Exceptional situations and program reliability. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2008.

[55] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, 2009.

[56] D. A. Wheeler. Sloccount. Available at `http://www.dwheeler.com/sloccount/`, 2015.

[57] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck. Chucky: exposing missing checks in source code for vulnerability discovery. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

A Appendix

Listing 3: Sample implementation of RAND_pseudo_bytes in OpenSSL

```c
/* crypto/engine/hw_aep.c */
int aep_rand(unsigned char *buf, int len)
{
  ...
  AEP_RV rv = AEP_R_OK;
  AEP_CONNECTION_HNDL hConnection;

  rv = aep_get_connection(&hConnection);
  if (rv != AEP_R_OK) {
    AEPHKerr(AEPHK_F_AEP_RAND,
      AEPHK_R_GET_HANDLE_FAILED);
    goto err_nounlock;
  }

  if (len > RAND_BLK_SIZE) {
    rv = p_AEP_GenRandom(hConnection, len,
      2, buf, NULL);
    if (rv != AEP_R_OK) {
      AEPHKerr(AEPHK_F_AEP_RAND,
        AEPHK_R_GET_RANDOM_FAILED);
      goto err_nounlock;
    }
  }
  ...
  return 1;
err_nounlock:
    return 0;
}
```

Table 6: Tested functions and bug counts

	Function Name	Bug Count	False Positives
OpenSSL	ASN1_INTEGER_set	4	0
	BN_mod_exp	3	0
	BN_sub	2	0
	EC_KEY_up_ref	1	0
	EC_POINT_cmp	1	0
	PEM_read_bio_X509	2	0
	RAND_pseudo_bytes	20	1
	X509_get_serialNumber	3	1
	i2a_ASN1_INTEGER	3	0
	i2d_X509	9	0
	Total	48	2
GnuTLS	asn1_read_value	4	0
	asn1_write_value	3	0
	gnutls_openpgp_crt_get_subkey_idx	1	0
	gnutls_openpgp_privkey_get_subkey_idx	3	0
	gnutls_privkey_get_pk_algorithm	3	1
	gnutls_x509_crq_get_dn_by_oid	2	0
	gnutls_x509_crq_get_extension_info	1	0
	gnutls_x509_crq_get_pk_algorithm	2	0
	gnutls_x509_crt_get_serial	1	0
	gnutls_x509_privkey_import	0	1
	gnutls_x509_privkey_import_pkcs8	1	0
	record_overhead_rt	2	0
	Total	23	2
mbedTLS	aes_setkey_enc	0	1
	asn1_get_int	2	0
	asn1_get_tag	8	0
	md_hmac_starts	2	0
	md_init_ctx	2	0
	mpi_fill_random	5	0
	ssl_handshake	0	1
	Total	19	2
wolfSSL	wc_InitRsaKey	0	1
	wc_ShaHash	0	1
	mp_init	0	8
	Total	0	10
cURL	RAND_bytes	2	0
	SSL_get_peer_cert_chain	0	1
	SSL_shutdown	0	1
	Total	2	2
httpd	BIO_free	4	0
	BIO_new	1	1
	SSL_CTX_new	1	0
	SSL_CTX_use_certificate_chain_file	1	0
	SSL_get_peer_cert_chain	0	1
	SSL_get_peer_certificate	0	1
	SSL_get_verify_result	0	1
	SSL_read	0	1
	SSL_write	0	1
	Total	7	6
Lynx	SSL_set_fd	1	0
	SSL_CTX_new	0	2
	Total	1	2
Mutt	SSL_CTX_new	0	1
	BIO_new	1	0
	SSL_shutdown	1	0
	Total	2	1
Wget	BIO_new	0	1
	Total	0	1
Grand_Total		**102**	**28**