# Mining API Error-Handling Specifications
# from Source Code

Mithun Acharya and Tao Xie

Department of Computer Science, North Carolina State University, Raleigh, NC, USA, 27695
{acharya,xie}@csc.ncsu.edu

**Abstract.** API error-handling specifications are often not documented, necessitating automated specification mining. Automated mining of error-handling specifications is challenging for procedural languages such as C, which lack explicit exception-handling mechanisms. Due to the lack of explicit exception handling, error-handling code is often scattered across different procedures and files making it difficult to mine error-handling specifications through manual inspection of source code. In this paper, we present a novel framework for mining API error-handling specifications automatically from API client code, without any user input. In our framework, we adapt a trace generation technique to distinguish and generate static traces representing different API run-time behaviors. We apply data mining techniques on the static traces to mine specifications that define correct handling of API errors. We then use the mined specifications to detect API error-handling violations. Our framework mines 62 error-handling specifications and detects 264 real error-handling defects from the analyzed open source packages.[1]

## 1   Introduction

**Motivation.** A software system interacts with third-party libraries through various Application Programming Interfaces (API). Throughout the paper, we overload the term API to mean either a set of related library procedures or a single library procedure in the set – the actual meaning should be evident from the context. Incorrect handling of errors incurred after API invocations can lead to serious problems such as system crashes, leakage of sensitive information, and other security compromises. API errors are usually caused by stressful environment conditions, which may occur in forms such as high computation load, memory exhaustion, process related failures, network failures, file system failures, and slow system response. As a simple example of incorrect API error handling, a *send* procedure, which sends the content of a file across the network as packets, might incorrectly handle the failure of the socket API (the socket API can return an error value of -1, indicating a failure), if the *send* procedure returns without releasing system resources such as previously allocated packet buffers and opened file handlers. Unfortunately, error handling is the least understood, documented, and tested part of a system. Toy's study [14] shows that more than 50% of all system failures in

---

a telephone switching application are due to incorrect error-handling algorithms. Cristian's survey [7] reports that up to two-thirds of a program may be devoted to error detection and recovery. Hence, correct error handling should be an important part of any reliable software system. Despite the importance of correct error handling, programmers often make mistakes in error-handling code [4, 10, 17]. Correct handling of API errors can be specified as formal specifications verifiable by static checkers at compile time. However, due to poor documentation practices, API error-handling specifications are often unavailable or imprecise. In this paper, we present a novel framework for statically mining API error-handling specifications automatically from software packages (API client code) implemented in C.

**Challenges.** There are three main unique challenges in automatically mining API error-handling specifications from source code. (1) Mining API error-handling specifications, which are usually temporal in nature, requires identifying API *details* from source code such as (a) *critical* APIs (APIs that fail with errors), (b) different error checks that should follow such APIs (depending on different API error conditions), and (c) proper error handling or clean up in the case of API failures, indicated by API errors. Furthermore, clean up APIs might depend on the APIs called before the error is handled. Static approaches [17, 16] exist for mining or checking API error-handling specifications from software repositories implemented in object-oriented languages such as Java. Java has explicit *exception-handling* support and the static approaches mainly analyze the `catch` and `finally` blocks to mine or check API error-handling specifications. Procedural languages such as C do not have explicit exception-handling mechanisms to handle API errors, posing additional challenges for automated specification mining: API details are often scattered across different procedures and files. Manually mining specifications from source code becomes hard and inaccurate. Hence, we need interprocedural techniques to mine critical APIs, different error checks, and proper clean up from source code to automatically mine error-handling specifications. (2) As programmers often make mistakes along API error paths [4, 10, 14, 17], the proper clean up, being common among error paths and normal paths, should be mined from normal traces (i.e., static traces without API errors along normal paths) instead of error traces (i.e., static traces with API errors along error paths). Hence, we need techniques to generate and distinguish error traces and normal traces, even when the API error-handling specifications are not known *a priori*. (3) Finally, API error-handling specifications can be *conditional* – the clean up for an API might depend on the actual return value of the API. Hence, trace generation has to associate conditions along each path with the corresponding trace.

**Contributions.** To address the preceding challenges, we develop a novel framework for statically mining API error-handling specifications directly from software packages (API client code), without requiring any user input. Our framework allows mining system code bases for API error-handling violations without requiring environment setup for system executions or availability of sufficient system tests. Furthermore, our framework detects API error-handling violations, requiring no user input in the form of specifications, programmer annotations, profiling, instrumentation, random inputs, or a set of relevant APIs. In particular, in our framework, we apply data mining techniques on generated static traces to mine specifications that define correct handling of errors for

the APIs used in the analyzed software packages. We then use the mined specifications to detect API error-handling violations. In summary, this paper makes the following main contributions:

– **Static approximation of different API run-time behaviors.** We adapt a static trace generation technique [2] to distinguish and approximate different API run-time behaviors (e.g., error and normal behaviors), thus generating error traces and normal traces inter-procedurally.

– **Specification mining and violation detection.** We apply different mining techniques on the generated error traces and normal traces to identify clean up code, distinguish clean up APIs from other APIs, and mine specifications that define correct handling of API errors. To mine conditional specifications, we adapt trace generation to associate conditions along each path with the corresponding trace. We then use the mined specifications to detect API error-handling violations.

– **Implementation and Experience.** We implement the framework and validate the effectiveness of the framework on 10 packages from the `Redhat-9.0` distribution (52 KLOC), `postfix-2.0.16` (111 KLOC), and 72 packages from the `X11-R6.9.0` distribution (208 KLOC). Our framework mines 62 error-handling specifications and detects 264 real error-handling defects from the analyzed packages.

The remainder of this paper is structured as follows. Section 2 starts with a motivating example. Section 3 explains our framework in detail. Section 4 presents the evaluation results. Section 5 discusses related work. Finally, Section 6 concludes our paper.

## 2   Example

In this section, we use the example code shown in Figures 1(b) and 1(c) to define several terms and notations (summarized in Figure 1(a)) used throughout the paper. We also provide a high-level overview of our framework using the example code.

**API errors.** All APIs in the example code are shown in bold font. In Figure 1(c), `InitAAText` and `EndAAText` are *user-defined procedures*. In the figure, user-defined procedures are shown in italicized font. The user-defined procedure in which an API is invoked is called the *enclosing procedure* for the API. In Figure 1(c), `EndAAText`, for instance, is the enclosing procedure for the APIs `XftDrawDestroy` (Line 27), `XftFontClose` (Line 28), and `XftColorFree` (Line 29). APIs can fail because of stressful environment conditions. In procedural languages such as C, API failures are indicated through *API errors*. API errors are special return values of the API (such as `NULL`) or distinct `errno` flag values (such as `ENOMEM`) indicating failures. For example, in Figure 1(b), API `recvfrom` returns a negative integer on failures. The API error from `recvfrom` is reflected by the return variable `cc`. APIs that can fail with errors are called as *critical* APIs. A condition checking of API return values or `errno` flag in the source code against API errors is called *API-Error Check* (AEC); we use AEC(a) to denote AEC of API `a`. For example, AEC(`recvfrom`) is `if(cc<0)`.

**Error block.** The block of code following an API-error check, which is executed if the API fails is called the *error block*. Error blocks contain error-handling code to handle API failures. We use EB(a) to denote the error block of API `a`. For example, Lines

| Definitions and Acronyms | | | Specification (S) | Error-Check Specification (ErCS) |
|---|---|---|---|---|
| Library Application Program Interface (API) | | | | Multiple-API Specification (MAS) |
| API-Error Check (AEC). AEC(a) is the required error check for API a. | | | Violation (V) | Error-Check Violation (ErCV) |
| Error Block (EB). EB(a) is the error block of API a. AEC(a) precedes EB(a). | | | | Multiple-API Violation (MAV) |
| Path (P) | Error Path (ErP) | Error Exit-Path (ErExP) | | |
| | | Error Return-Path (ErRP) | | |
| | Normal Path (NP) | | | |
| Trace (T) | Error Trace (ErT) | Error Exit-Trace (ErExT) | | |
| | | Error Return-Trace (ErRT) | | |
| | Normal Trace (NT) | | | |

**(a) Definitions and Acronyms**

```
~/Redhat-9.0/routed/ripquery/query.c
1  #include <sys/socket.h>
2  int main(...){
3      ...
4      s = socket(...);
5      ...
6      cc = recvfrom(s, ...)
7      ...
8      if (cc < 0){
9          ...
10         close(s);
11         exit(1);
12         }
13     ...
14     close(s)
15     ...
16     }
```

**(b) Example code from Redhat-9.0/routed-0.17-14**

```
~/X11-R6.9.0/x11perf/do_text.c
1  #include <X11/Xft/Xft.h>
2  ...
3  static XftFont *aafont;
4  static XftDraw *aadraw;
5  static XftColor aacolor;
6  ...
7  int InitAAText(XParms xp, Parms p, int reps){
8      ...
9      aafont = XftFontOpenName (...);
10     if (aafont == NULL) {
11         ...
12         return 0;
13         }
14     aadraw = XftDrawCreate (...);
15     if (!XftColorAllocValue (..., &aacolor)){
16         ...
17         XftFontClose (xp->d, aafont);
18         XftDrawDestroy (aadraw);
19         ...
20         return 0;
21         }
22     ...
23     }
24  ...
25  void EndAAText(XParms xp, Parms p){
26     ...
27     XftDrawDestroy (aadraw);
28     XftFontClose (xp->d, aafont);
29     XftColorFree (..., &aacolor);
30     ...
31     }
```

**(c) Example code from X11-R6.9.0/x11perf**

**Fig. 1.** Terminologies and example code

9-11 in Figure 1(b), Lines 11-12 and 16-20 in Figure 1(c) represent EB(recvfrom), EB(XftFontOpenName), and EB(XftColorAllocValue), respectively. A given API can have multiple error blocks depending on the different ways that it can fail (not shown in the examples for simplicity).

**Paths, Traces, and Scenarios.** A control-flow path exists between two program *points* if the latter is reachable from the former through some set of control-flow edges, i.e., Control Flow Graph (CFG) *edges*. Our framework identifies two types of paths - *error path* and *normal path*. There are two types of error paths. Any path from the beginning of the main procedure to an exit call (such as exit) in the error block of some API is called the *error exit-path*. For example, all paths ending at the exit call at Line 11 in Figure 1(b) are error exit-paths (exit call inside EB(recvfrom)). Any path from the beginning of the main procedure to a return call in the error block of some API is called the *error return-path*. For example, in Figure 1(c), all paths ending at the return call at Lines 12 (return call inside EB(XftFontOpenName)) and 20 (return call inside EB(XftColorAllocValue)) are error return-paths. Error exit-paths and error return-paths are together known as *error path*s. A *normal path* is any path from the beginning of the main procedure to the end of the main procedure without any API errors. For example, any path from Line 3 to Line 15 in Figure 1(b) is a normal path. For a given path, a trace is the print of all statements that exist along that path. Error paths,

error exit-paths, error return-paths, and normal paths have corresponding traces: *error traces*, *error exit-traces*, *error return-traces*, and *normal traces*. Error exit-traces and error return-traces are together known as error traces. Two APIs are *related* if they manipulate at least one (or more) common variable(s). For example, in Figure 1(b), APIs recvfrom and close are related to API socket. The socket API *produces* s, which is *consumed* by the APIs recvfrom and close. A *scenario* is a set of related APIs in a given trace. A given trace can have multiple scenarios. For example, if there were multiple socket calls in Figure 1(b), then each socket call, along with its corresponding related APIs, forms a different scenario.

**API error-handling specifications.** We identify two types of API error-handling specifications that dictate correct error handling along all paths in a program: *error-check specifications* and *multiple-API specifications*. Error-check specifications dictate that correct AEC(a)'s (API-Error Checks) exist for each API a (which can fail), before the API's return value is *used* or the main procedure returns. For a given API a, the absence of AEC(a) causes an *error-check violation*. Multiple-API specifications dictate that the right *clean up* APIs are called along all paths. Clean up APIs are APIs called, generally before a procedure's return or program's exit, to free resources such as memory, sockets, pipes, and files or to *rollback* the state of a global resource such as the system registry and databases. For example, in Figure 1(c), XftFontClose (Line 17) and XftDrawDestroy (Line 18) are the clean up APIs in EB(XftColorAllocValue). In Figure 1(c), one error-check specification (the return value of XftColorAllocValue should be checked against NULL) and two multiple-API specifications (XftFontOpenName should be followed by XftFontClose, and XftDrawCreate should be followed by XftDrawDestroy) are evident. Violation of a multiple-API specification along a given path is a *multiple-API violation*. Multiple-API violations along error exit-paths could be less serious as the operating system might reclaim unfreed memory and resource handlers along program exits. However, there are several cases where explicit clean up is necessary even on program exits. For instance, unclosed files could lose recorded data along an error exit-path if the buffers are not flushed out to the disk. In addition, any user-defined procedure altering a global resource (such as the system registry or a database) should *rollback* along error exit-paths to retain the integrity of the global resource. Next, we present the high-level overview of our framework using the example code.

The only input to our framework is the compilable source code of software package(s) implemented in C. To mine specifications, our framework initially distinguishes and generates API error traces and normal traces, for reasons explained later. Our framework then detects API error-handling violations in the source code using the mined specifications. In particular, our framework consists of the following three stages:

**Error/normal trace generation.** The trace generation stage distinguishes and generates error traces (error exit-traces and error return-traces) and normal traces interprocedurally. Along normal paths, it is difficult to distinguish clean up APIs from other APIs. Hence, our framework identifies *probable* clean up APIs from the error traces. For example, in Figure 1(b), our framework identifies the API close (Line 10) from the error exit-trace that goes through EB(recvfrom). In Figure 1(c), our framework identifies XftFontClose (Line 17) and XftDrawDestroy (Line 18) from the error

return-trace that goes through EB(XftColorAllocValue). Note that, in Figure 1(c), the clean up APIs can also be invoked through the user-defined procedure EndAAText, inter-procedurally. However, even in the error block, there could be other APIs that are not necessarily clean up APIs (hence the term, *probable*). The final set of actual clean up APIs and the APIs related to them are determined during the specification mining stage.

**Specification mining.** The specification mining stage generates error-check specifications and multiple-API specifications. Our framework mines error-check specifications from error traces by determining API-error checks (AEC) for each API. For example, our framework determines AEC(recvfrom) to be if (cc < 0) from the error-exit trace that goes through EB(recvfrom). Programmers often make mistakes along API error paths. Hence, proper clean up, being common among error paths and normal paths, should be mined from normal traces instead of error traces. Once probable clean up APIs are mined from error traces, our framework mines APIs that might be related to the probable clean up APIs from normal traces. For example, in Figure 1(c), our framework determines from normal traces that XftFontClose is related to XftFontOpenName, and XftDrawDestroy is related to XftDrawCreate (Figure 1(c), however, does not show normal paths or traces for simplicity). Our framework generates multiple-API specifications by applying sequence mining on normal traces.
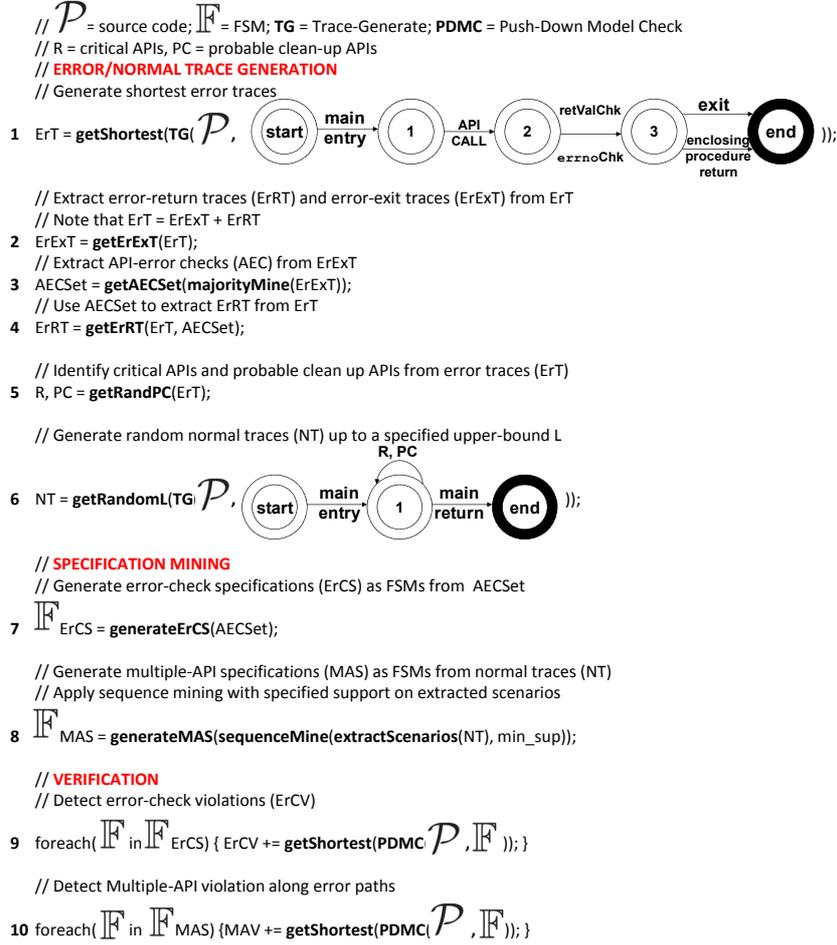
**Verification.** Our static verifier uses the mined specifications (error-check and multiple-API specifications) to detect violations (error-check and multiple-API violations) in the source code. Next, we present our framework in detail.

## 3   Framework

The algorithm presented in Figure 2 shows the details of our framework. There are 3 stages and 10 steps (numbered 1-10) in our algorithm. Section 3.1 describes the error/normal trace generation stage (Steps 1-6). Section 3.2 (Steps 7-8) explains the steps involved in mining API error-handling specifications from the static traces. Finally, Section 3.3 describes the verification stage for detecting API error-handling violations of the mined specifications (Steps 9-10). Our framework adapts a trace generation technique developed in our previous work to generate static traces representing different API run-time behaviors. The trace generation technique uses *triggers* to generate static traces. Triggers are represented using finite state machines. The static traces generated by the trace generation technique with a given trigger depend on the the transitions in the trigger. Readers may refer to our previous work [2] for further details.

### 3.1   Error/Normal Trace Generation

In this section, we explain how we adapt the trace generation technique [2] for generating API error and normal traces from source code. As shown in Figure 2, the error/normal trace generation stage has six steps: generate error traces (Step 1), process error traces (Steps 2-4), identify critical APIs and probable clean up APIs from error traces (Step 5), and finally, generate normal traces (Step 6). The various steps are explained next.

```
// 𝒫 = source code; 𝔽 = FSM; TG = Trace-Generate; PDMC = Push-Down Model Check
// R = critical APIs, PC = probable clean-up APIs
// ERROR/NORMAL TRACE GENERATION
// Generate shortest error traces

1  ErT = getShortest(TG( 𝒫 , (start) --main entry--> (1) --API CALL--> (2) --retValChk / errnoChk--> (3) --exit / enclosing procedure return--> (end) ));

// Extract error-return traces (ErRT) and error-exit traces (ErExT) from ErT
// Note that ErT = ErExT + ErRT
2  ErExT = getErExT(ErT);
// Extract API-error checks (AEC) from ErExT
3  AECSet = getAECSet(majorityMine(ErExT));
// Use AECSet to extract ErRT from ErT
4  ErRT = getErRT(ErT, AECSet);

// Identify critical APIs and probable clean up APIs from error traces (ErT)
5  R, PC = getRandPC(ErT);

// Generate random normal traces (NT) up to a specified upper-bound L
6  NT = getRandomL(TG( 𝒫 , (start) --main entry--> (1) --R, PC (self-loop) / main return--> (end) ));

// SPECIFICATION MINING
// Generate error-check specifications (ErCS) as FSMs from AECSet
7  𝔽 ErCS = generateErCS(AECSet);

// Generate multiple-API specifications (MAS) as FSMs from normal traces (NT)
// Apply sequence mining with specified support on extracted scenarios
8  𝔽 MAS = generateMAS(sequenceMine(extractScenarios(NT), min_sup));

// VERIFICATION
// Detect error-check violations (ErCV)
9  foreach( 𝔽 in 𝔽 ErCS) { ErCV += getShortest(PDMC( 𝒫 , 𝔽 )); }

// Detect Multiple-API violation along error paths
10 foreach( 𝔽 in 𝔽 MAS) {MAV += getShortest(PDMC( 𝒫 , 𝔽 )); }
```

**Fig. 2.** The algorithm for mining API error-handling specifications

**Step 1 - Generate error traces.** An error trace starts from the beginning of the `main` procedure and ends in some API error-block with an exit call (causing the program to exit) or a `return` call (causing the enclosing procedure to return). The trigger FSM, say $\mathbb{F}$ (Step 1, Figure 2), is used by our trace generator (procedure `TG` in the figure) to generate error traces from the program source code ($\mathcal{P}$). The procedure `TG` represents our trace generation technique, which adapts the push-down model checking (PDMC) process. Transitions `retValChk` and `errnoChk` in the trigger $\mathbb{F}$ (from State 2 to State 3) identify the return-value check and error-flag check, respectively, for the API. Transitions from State 3 to the final state (State `end`) in the trigger $\mathbb{F}$ capture code blocks following the `retValChk` or `errnoChk` in which the program exits or the enclosing procedure returns. The procedure `TG` generates all traces in $\mathcal{P}$ that satisfy the trigger $\mathbb{F}$. However, the procedure `getShortest` (Step 1, Figure 2) returns only the shortest

trace from the set of all traces generated by `TG`. As we are interested only in the API-error check and the set of probable clean up APIs (PC) in the API error block for a given API from error traces, the program statements prior to the API invocation are not needed. Hence, it suffices to generate the shortest path for each API invocation with a following `retValChk` or `errnoChk`. If there are multiple `retValChk` or `errnoChk` for an API call site, then our framework generates the shortest trace for each of the checks. The trigger $\mathbb{F}$ captures the elements of `retValChk`, `errnoChk`, and the code block after these checks, even if these elements are scattered across procedure boundaries. However, the traces generated by this step can also have traces where `retValChk` or `errnoChk` is followed by a normal return of the enclosing procedure. Such traces, which are not error traces, are pruned out in the next step.

**Steps 2, 3, and 4 - Process error traces.** Our framework easily extracts error exit-traces from error traces (procedure `getErExT`, Step 2, Figure 2): error traces that end with an exit call are error exit-traces. We assume that the API `retValChk` or `errnoChk`, which precedes an exit call in an error-exit trace, is an API-error check. We then distinguish between the *true* and *false* branches of the API-error check. For example, in Figure 1(b), since `exit(...)` appears in the true branch of AEC(`recvfrom`) (`if(cc<0)`), we assume that `<0` is the error return value (API error) of `recvfrom`. For each API, our framework records API-error check with majority occurrences (procedure `majorityMine`, Step 3, Figure 2) among error exit-traces (procedure `getAECSet`, Step 3, Figure 2). As mentioned in the previous step, the traces generated in Step 1 can also have traces where `retValChk` or `errnoChk` is followed by a normal return of the enclosing procedure. Our framework uses the API-error check set computed from error exit-traces to prune out such traces to generate error return-traces (procedure `getErRT`, Step 4, Figure 2).

**Step 5 - Identify critical APIs and probable clean up APIs from error traces.** Our framework computes the set R (critical APIs) and the set PC (probable clean up APIs) in this step (procedure `getRandPc`, Step 5, Figure 2). The set R of critical APIs is easily computed from error exit-traces and error return-traces. A key observation here is that it is much easier to find clean up APIs along error paths than normal paths. It is because, on API failures, before the program exits or the enclosing procedure returns, the primary concern is clean up. Along normal paths, however, it is difficult to separate clean up APIs from other APIs. Hence, our framework identifies probable clean up APIs (the set PC) from the error traces. The term *probable* indicates that the APIs that occur in error blocks need not always be clean up APIs. The mining phase prunes out the non-clean-up APIs from the set PC. In the next step, we show how our framework identifies APIs related to the probable clean up APIs. These related APIs occur prior to API-error checks in the source code.

**Step 6 - Generate normal traces.** A normal trace starts from the beginning of the `main` procedure and ends at the end of the `main` procedure. The procedure `TG` uses the trigger FSM, say $\mathbb{F}$ (Step 6, Figure 2), to generate normal traces from the program source code ($\mathcal{P}$). The edges for State 2 in the trigger $\mathbb{F}$ are critical (set R) and probable clean up APIs (set PC). Our framework generates normal traces (involving critical and probable clean up APIs) randomly up to a user-specified upper bound $L$ (procedure `getRandomL`, Step 6, Figure 2), inter-procedurally. The traces contain the probable clean up APIs and

the APIs related to them, if any. Finally, as API error-handling specifications can be conditional, the clean up for an API might depend on the actual return value of the API. As a simple example, for the `malloc` API, the `free` API is called only along paths in which the return value of `malloc` is not `NULL` (condition). Hence, normal paths (normal traces) are associated with their corresponding conditions involving API return values. The conditions, along with API sequences, form a part of normal traces and are used in the specification mining stage, explained next.

### 3.2 Specification Mining

The specification mining stage mines error-check and multiple-API specifications from the static traces (Steps 7-8). The scenario extraction and sequence mining are performed in Step 8.

**Step 7 - Mine error-check specifications.** Our framework generates error-check specifications (procedure `generateErCS`, Step 7, Figure 2) as Finite State Machines (FSM, $\mathbb{F}_{ErCS}$) from the mined API-error check set. The FSMs representing the error-check specifications specify that each critical API should be followed by the correct error checks.

**Step 8 - Mine multiple-API specifications.** Our frameork mines multiple-API specifications from normal traces (procedure `generateMAS`, Step 8, Figure 2) as FSMs ($\mathbb{F}_{MAS}$). Normal traces include the probable clean up APIs (PC), APIs related to the set PC, and the conditions (involving API return values). The main observation used in mining multiple-API specifications from normal traces is that programmers often make mistakes along error paths [4, 10, 14, 17]. Hence, our framework mines related APIs from only normal traces and not from error traces. However, a single normal trace generated by the trace generator might involve several API scenarios, being often interspersed. A scenario (see Section 2) is a set of related APIs in a given trace. Our framework separates different API scenarios from a given normal trace, so that each scenario can be fed separately to our miner. We use a scenario extraction algorithm (procedure `extractScenarios`, Step 8, Figure 2) [2] that is based on identifying *producer-consumer* chains among APIs in the trace. The algorithm is based on the assumption that an API and its corresponding clean up APIs have some form of data dependencies between them such as a producer-consumer relationship. Each producer-consumer chain is generated as an independent scenario. For example, in Figure 1(c), the API `XftFontOpenName` (Line 9) produces `aafont`, which is consumed by the API `XftFontClose` (Line 17). The APIs `XftFontOpenName` and `XftFontClose` are generated as an independent scenario.

Our framework mines multiple-API specifications from independent scenarios using frequent-sequence mining (procedure `sequenceMine`, Step 8, Figure 2). Let $IS$ be the set of independent scenarios. We apply a frequent sequence-mining algorithm [15] on the set $IS$ with a user-specified support $min\_sup$ ($min\_sup \in [0, 1]$), which produces a set $FS$ of frequent sequences that occur as subsequences in at least $min\_sup \times |IS|$ sequences in the set $IS$. Note that our framework can mine the different error-handling specifications for the different errors of a given API as long as the different specifications have enough support among the analyzed client code.

### 3.3   Verification

Our framework uses the specifications to find API error-handling violations (Steps 9-10).

**Steps 9 and 10 - Detect error-check and multiple-API violations.** In Steps 1 and 6, we adapt the push-down model checking (PDMC) process for trace generation by the procedure TG. Here we use the PDMC process for property verification. The specifications mined by our framework as FSMs ($\mathbb{F}_{ErCS}$ and $\mathbb{F}_{MAS}$) represent the error-handling properties to be verified at this stage. Our framework verifies the property FSMs in $\mathbb{F}_{ErCS}$ and $\mathbb{F}_{MAS}$ against the source code ($\mathcal{P}$). The mined specifications can also be used to verify the correct API error handling in other software packages. For verifying conditional specifications, we adapt the PDMC process to track the value of variables that take the return value of an API call along the different branches of conditional constructs. Our framework generates (procedure getShortest) the shortest path for each detected violation (i.e., a potential defect) in the program, instead of all violating traces, thus making defect inspection easier for the users.

## 4   Evaluation

To generate static traces, we adapted a publicly available model checker called MOPS [6] with procedures (Steps 1-10) shown in Figure 2. We used BIDE [15] to mine frequent sequences. We have applied our framework on 10 packages from the Redhat-9.0 distribution (52 KLOC), postfix-2.0.16 (111 KLOC), and 72 packages from the X11-R6.9.0 distribution (208 KLOC). The analyzed packages use the APIs from the POSIX and X11 libraries. We selected POSIX and X11 clients because the POSIX standard [1] and the Inter-Client Communication Conventions Manual (IC-CCM) [13] from the X Consortium standard were readily available. These standards describe rules for how well-behaved programs should use the APIs, serving as an oracle for confirming our mined results. We ran our evaluation on a machine with Redhat Enterprise Linux version 2.6.9-5ELsmp, 3GHz Intel Xeon processor, and 4GB RAM. For specification mining and violation detection, the analysis cost ranges from under a minute for the smallest package to under an hour for the largest one. We next explain the evaluation results (summarized in Figure 3(a)) for the various stages of our framework.

**Trace generation.** The number of error exit-traces and error return-traces generated by our framework are shown in Columns 3 (**ErExT**) and 4 (**ErRT**) of Figure 3, respectively. To evaluate trace generation, we manually inspected the source code for each error exit-trace produced by our framework and each error exit-trace missed by our framework. Error exit-traces missed by our framework can be determined by manually identifying the exit statements in the analyzed program not found in any of the generated error exit-traces. There are five sub-columns in Column 3 (**ErExT**): $\Sigma$ (total number of error exit-traces generated or missed by our framework), $\Sigma^{op}$ (total number of error exit-traces actually generated by our framework), $\mathbf{FN} = \Sigma - \Sigma^{op}$ (total number of error exit-traces missed by our framework), $\mathbf{FP}$ (false positives: generated traces that are not actually error exit-traces), and $\mathbf{IP}$ (inter-procedural: the number of traces in which the API invocation, API-error check, and error blocks were scattered across procedure boundaries).

| 1. Packages | 2. LOC | 3. ErExT | | | | |
|---|---|---|---|---|---|---|
| | | $\Sigma$ | $\Sigma^{op}$ | FN = $\Sigma$- $\Sigma^{op}$ | FP | IP |
| 10-Redhat-9.0-pkgs | 52 K | 338 | 320 | 18 | 35 | 18 |
| postfix-2.0.16 | 111 K | 124 | 92 | 32 | 3 | 124 |
| X11-R6.9.0 | 208 K | 286 | 248 | 38 | 27 | 164 |
| $\Sigma$ | 371 K | 748 | 660 | 88 (12%) | 65( 10%) | 306(41%) |

| 4. ErRT | 5. ErCS | | 6. ErCV | | 7. MAS | | 8. MAV | |
|---|---|---|---|---|---|---|---|---|
| | $\Sigma$ | FP | $\Sigma$ | FP | $\Sigma$ | FP | $\Sigma$ | FP |
| 205 | 31 | 3 | 58 | 1 | 40 | 6 | 4 | 3 |
| 30 | 31 | 3 | 4 | 2 | 40 | 6 | 0 | 0 |
| 305 | 31 | 3 | 170 | 13 | 40 | 6 | 56 | 9 |
| 540 | 31 | 3(10%) | 232 | 16(7%) | 40 | 6(15%) | 60 | 12(20%) |

**(a) Traces and violations**

| (R)XGetVisualInfo | (R)XpQueryScreens | (R)XpGetAttributes |
|---|---|---|
| XGetWindowProperty(12) | (R)XScreenResourceString | (R)XpGetOneAttribute |
| XQueryTree(5) | (R)XGetAtomName | (R)glXChooseVisual |
| (R)XFetchBytes | (R)malloc | XGetIMValues(3) |
| (R)XGetKeyboardMapping | XGetWMProtocols(3) | (R)XGetWMHints |

**(b) Multiple-API specifications for the clean up API**
**`XFree`, mined by our framework**

$\Sigma$: Total, **IP:** Interprocedural, **FP:** False Positives, **FN:** False Negatives, **ErExT**: Error Exit-Traces, **ErRT**: Error Return-Traces, **ErCS**: Error-Check Specifications, **ErCV**: Error-Check Violations, **MAS**: Multiple-API Specifications, **MAV**: Multiple-API Violations

**Fig. 3.** Evaluation Results

We observed that the number of false negatives (FN) and false positives (FP) were low, at 12% (88/748) and 10% (65/660), respectively. The main reason for false negatives in the traces generated by our framework is the lack of aliasing and pointer analysis. For example, in xkbvleds/utils.c, the variable outDpy takes the return value of the API XtDisplay. Then the value of outDpy is assigned to another variable inDpy, and inDpy is compared to NULL. If inDpy is NULL, a user-defined procedure uFatalError is called, which then calls exit. Our framework did not capture the aliasing of outDpy to inDpy, and hence missed the trace. However, as the number of false negatives was low, our framework still generated enough traces for the mining process. Some of the traces generated by our framework were not error exit-traces, leading to false positives. For example, in tftp/tftpd.c, the variable f (process id) takes the return value of the API fork. The program exits on f>0 (parent process; not an error). Although the trace was generated by our framework, it is not an error exit-trace (fork fails with a negative integer). However, as the number of false positives was low, false error exit-traces were pruned by the mining process. 41% (306/748) of all the error

exit-traces were scattered across procedure boundaries, highlighting the importance of inter-procedural trace generation. Specifically, all error exit-traces from the `postfix` package crossed procedure boundaries.

Our framework identifies the set of probable clean up APIs from the error traces (Step 5, Figure 2). After discarding string-manipulating APIs (such as `strcmp` and `strlen`), printing APIs (such as `printf` and `fprintf`), and error-reporting APIs (such as `perror`), which frequently appear (but unimportant) in error blocks, our framework identified 36 APIs as probable clean up APIs. Our framework used probable clean up APIs in generating normal traces. For each compilable unit in the analyzed packages, our framework randomly generated 20 normal traces, ensuring there are enough distinct traces for mining. Our framework discarded 14/36 APIs after mining the normal traces with one of the following reasons: (1) insufficient call sites and hence an insufficient number of traces to mine from (for example, the API `XEClearCtrlKeys` had only two traces), (2) no temporal dependencies with any APIs called prior to the error block (for example, the API `XtSetArg` appears in an exit trace from `xlogo/xlogo.c`. However, `XtSetArg` does not share any temporal dependencies with APIs called prior to the exit block), or (3) insufficient support among the scenarios. Our framework mined 40 multiple-API specifications from the remaining 22/36 probable clean up APIs (Column 7, **MAS**).

**Error-check specifications.** Our framework mined error-check specifications for only those APIs that occur more than three times among the error traces. In all, our framework mined 31 error-check specifications (Column 5, **ErCS**) from the error traces across all the analyzed packages. 3 (10%) out of the 31 (subcolumn $\Sigma$) mined specifications were false positives (subcolumn **FP**). For example, the API `geteuid` returns the effective user ID of the current process. The effective ID corresponds to the set ID bit on the file being executed [1]. Our framework encounters `geteuid()!=0` at least 5 times among error traces leading to a false error-check specification – '*geteuid fails by returning a non-zero integer*'. But, a non-zero return value simply indicates an unprivileged process.

**Error-check violations.** The error-check specifications mined from error traces are used in detecting error-check violations along the error paths in the analyzed software packages. Column 6 (**ErCV**) of Figure 3(a) presents the number of error-check violations detected by our framework. We manually inspected the violations reported by our framework. 16 (7%) out of the 232 (subcolumn $\Sigma$) reported error-check violations were false positives (subcolumn **FP**). The main reason for false positives in the reported violations is, once again, the lack of aliasing and pointer analysis in our framework. For example, in `twm/session.c` and `smproxy/save.c`, the variable `entry` takes the return value of `malloc`. Then the variable `entry` is assigned to another variable `pentry`. The variable `pentry` is then checked for `NULL`, which was missed by our framework.

**Multiple-API specifications.** Our framework mines multiple-API specifications from normal traces. Our framework produces a pattern as a multiple-API specification if the pattern occurred in at least five scenarios, with a minimum support ($min\_sup$) of 0.8 among the scenarios. Our framework mined 40 multiple-API specifications (Column 7, **MAS**) across all the packages, with 6 (15%) of them being false positives (subcolumn **FP**). All multiple-API specifications mined by our framework were conditional – the

clean up APIs in conditional multiple-API specifications depend on the return value or a parameter (that holds the return value). As an example of a conditional specification, for the API XGetVisualInfo, cleaning up through the API XFree is necessary only if the fourth input parameter of XGetVisualInfo (the number of matching *visual structures*) is non-zero. False positives among the mined specifications may occur if some patterns occuring in the analyzed source code are not necessarily specifications. This result is a limitation shared by all mining approaches, requiring human inspection and judgement to distinguish real specifications from false ones. For example, our framework considered the APIs XSetScreenSaver and XUngrabPointer as probable clean up APIs, as both APIs appeared in some error traces generated by our framework. The first parameter of both these APIs is the display pointer produced by the API XOpenDisplay. Hence, our framework mined the property "XSetScreenSaver and XUngrabPointer should follow XOpenDisplay", leading to a false positive. The number of false specifications mined by our framework is low as the code bases used by our framework for mining are sufficiently large.

Our framework mines the maximum number of multiple-API specifications around the clean up API XFree. From the static traces, 35 APIs from the X11 library were found to interact with the XFree API, leading to 15 multiple-API specifications with sufficient support. The specifications mined around the API XFree are shown in Figure 3(b). XFree is a general-purpose X11 API that frees the specified data. XFree must be used to free any objects that were allocated by X11 APIs, unless an alternate API is explicitly specified for the objects [13]. The pointer consumed by the XFree API can either be a return value or a parameter (that holds the return value) of some X11 API. The "(R)" in (R)XGetVisualInfo, for instance, indicates that the return value of the API XGetVisualInfo should be freed through the API XFree along all paths. The "(5)" in XQueryTree(5), for instance, indicates that the fifth input parameter of the API XQueryTree should be freed through the API XFree along all paths.

**Multiple-API violations.** Our framework uses the multiple-API specifications mined from normal traces to detect multiple-API violations in the analyzed software packages. Column 8 (**MAV**) presents the number of multiple-API violations detected by our framework. We manually inspected the violations reported by our framework. 12 (20%) out of the 60 (subcolumn $\Sigma$) reported multiple-API violations were false positives (subcolumn **FP**). To verify conditional specifications, we adapted MOPS to track the value of variables that take the return value of an API call along the different branches of conditional constructs. Tracking API return values while verifying multiple-API specifications decreases the number of false positives, which would have otherwise been reported. As a simple example, verifying conditional specifications causes false positives such as "a file is not closed before the program exits on the failure (NULL) path of the open API" not to be reported. Verifying conditional specifications by tracking return values avoided 87 false positives in the analyzed packages, which would have otherwise been reported. In all, our framework mines 62 error-handling specifications and detects 264 real error-handling violations in the analyzed packages. Due to pointer-insensitive analysis, our framework might not mine all the error-handling specifications or detect all the error-check and multiple-API violations in the analyzed software packages, leading to false negatives. For the mined specifications and the detected violations,

we have not quantified the false negatives of our framework. Quantifying the violations missed by our framework (through manual inspection of source code along all possible paths in the presence of function pointers and aliasing) is difficult and error prone.

## 5   Related Work

**Dynamic.** Previous work has mined API properties from program execution traces. For example, Ammons et al. [3] mine API properties as probabilistic finite state automata from execution traces. Perracotta developed by Yang et al. [18] mines temporal properties (in the form of pre-defined templates involving two API calls) from execution traces. Different from these approaches, our framework mines specifications from source code of API clients. Dynamic approaches require setup of runtime environments and availability of sufficient system tests that exercise various parts of the program and hence the violations might not be easily exposed. In contrast, our new framework mines API error-handling specifications from static traces without suffering from the preceding issues.

**Static.** Previous several related static approaches developed by other researchers also mine properties from source code for finding defects. Engler et al. propose *Meta-level Compilation* [8] to detect rule violations in a program based on user-provided, simple, system-specific compiler extensions. Their approach detects defects by statically identifying inconsistencies in commonly observed behavior. PR-Miner developed by Li and Zhou [11] mine programming rules as frequent *itemsets* (ordering among program statements is not considered) from source code. Apart from being intra-procedural, neither approach considers data-flow or control-flow dependences between program elements, required for mining error-handling specifications. Two recent approaches in static specification mining, most related to our framework, are from Chang et al. [5] and Ramanathan et al. [12]. Chang et al.'s approach [5] mines specifications as *graph minors* from *program dependence graphs* by adapting a frequent sub-graph mining algorithm. Specification violations are then detected by their heuristic graph-matching algorithm. The scalability of their approach is limited by the underlying graph mining and matching algorithms. Furthermore, their approach does not mine conditional specifications. Ramanathan et al. [12] mine preconditions of a given procedure across different call sites. To compute preconditions for a procedure, their analysis collects predicates along each distinct path to each procedure call. As their approach is not applicable to postconditions, it cannot mine error-handling specifications. Static approaches [17, 16] exist to analyze programs written in Java, which has explicit exception-handling support. Several proposals [9] exist for extending C with exception-handling support. In contrast, our framework is applicable to applications implemented in procedural languages with no explicit support for exception handling.

## 6   Conclusions

We have developed a framework to automatically mine API error-handling specifications from source code. We then use the mined specifications to detect API error-handling violations from the analyzed software packages (API client code). We have

implemented the framework, and validated its effectiveness on 10 packages from the `Redhat-9.0` distribution (52 KLOC), `postfix-2.0.16` (111 KLOC), and 72 packages from the `X11-R6.9.0` (208 KLOC). Our framework mines 62 error-handling specifications and detects 264 real error-handling defects from the analyzed packages.

# References

1. IEEE Computer Society. IEEE Standard for Information Technology - Portable Operating System Interface POSIX - Part I: System Application Program Interface API, IEEE Std 1003.1b-1993 (1994)
2. Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: From usage scenarios to specifications. In: Proc. ESEC/FSE, pp. 25–34 (2007)
3. Ammons, G., Bodik, R., Larus, J.: Mining specifications. In: Proc. POPL, pp. 4–16 (2002)
4. Bruntink, M., Deursen, A.V., Tourwe, T.: Discovering faults in idiom-based exception handling. In: Proc. ICSE, pp. 242–251 (2006)
5. Chang, R.Y., Podgurski, A.: Finding what's not there: A new approach to revealing neglected conditions in software. In: Proc. ISSTA, pp. 163–173 (2007)
6. Chen, H., Wagner, D.: MOPS: an infrastructure for examining security properties of software. In: Proc. CCS, pp. 235–244 (2002)
7. Cristian, F.: Exception Handling and Tolerance of Software Faults. In Software Fault Tolerance, ch. 5. John Wiley and Sons, Chichester (1995)
8. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Proc. SOSP, pp. 57–72 (2001)
9. Gehani, N.H.: Exceptional C for C with exceptions. Software Practices and Experiences 22(10), 827–848 (1992)
10. Gunawi, H., Rubio-Gonzalez, C., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Liblit, B.: EIO: Error handling is occasionally correct. In: Proc. USENIX FAST, pp. 242–251 (2006)
11. Li, Z., Zhou, Y.: PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In: Proc. ESEC/FSE, pp. 306–315 (2005)
12. Ramanathan, M.K., Grama, A., Jagannathan, S.: Static specification inference using predicate mining. In: Proc. PLDI, pp. 123–134 (2007)
13. Rosenthal, D.: Inter-client communication Conventions Manual (ICCCM), Version 2.0. X Consortium, Inc. (1994)
14. Toy, W.: Fault-tolerant design of local ESS processors. In: The Theory and Practice of Reliable System Design. Digital Press (1982)
15. Wang, J., Han, J.: BIDE: Efficient mining of frequent closed sequences. In: Proc. ICDE, pp. 79–90 (2004)
16. Weimer, W., Necula, G.C.: Finding and preventing run-time error handling mistakes. In: Proc. OOPSLA, pp. 419–431 (2004)
17. Weimer, W., Necula, G.C.: Mining temporal specifications for error detection. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 461–476. Springer, Heidelberg (2005)
18. Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: Mining temporal API rules from imperfect traces. In: Proc. ICSE, pp. 282–291 (2006)