Static Detection of API Error-Handling Bugs via Mining 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

Incorrect handling of errors incurred after API invocations (in short, API errors) can lead to security and robustness problems, two primary threats to software reliability. Correct handling of API errors can be specified as formal specifications, verifiable by static checkers, to ensure dependable computing. But API error specifications are often unavailable or imprecise, and cannot be inferred easily by source code inspection. In this paper, we develop a novel framework for statically mining API error specifications automatically from software package repositories, without requiring any user input. Our framework adapts a compile-time push-down model-checker to generate interprocedural static traces, which approximate run-time API error behaviors. Data-mining techniques are used on these static traces to mine specifications that define the correct handling of errors for relevant APIs used in the software packages.The mined specifications are then used to uncover API error-handling bugs. We have implemented the framework, and validated the effectiveness of the framework on 82 widely used open-source software packages with approximately 300KLOC in total [1].

Submission Category: Testing, Verification, and Validation.

Keywords: Software Reliability, Mining, Static Traces, Specifications, API Error-Handling, Robustness

1 Introduction

A software system interacts with third-party libraries through various APIs (Application Programming Interfaces). Incorrect handling of errors incurred after API invocations (in short, API errors) can lead to security and robustness violations in complex software systems. These violations often lead to system crashes, leakage of sensitive information, and complete security compromises. Robustness is formally defined as the degree to which a software component behaves correctly in the presence of exceptional inputs or stressful environmental conditions [1]. 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. Stressful conditions, and hence API errors, however rare, should be gracefully handled. Traditional software testing focuses on correctness of functionality and is often insufficient for assuring the absence of API-level robustness violations. Robustness testing approaches [10, 11, 15] consider the target applications or operating systems as a black box, and send random or exceptional input values through their APIs. However, robustness testing approaches cannot easily generate *implicit* return exceptions through APIs, which are an important type of sources for robustness problems.

Correct handling of API errors can be specified as formal specifications verifiable by static checkers to ensure the absence of error-handling bugs. Writing such specifications, which are usually temporal in nature, requires identifying *API details* such as (1) the relevant APIs that fail with errors, (2) different error-checks that should follow such APIs (depending on different API error conditions), and (3) proper error-handling or clean-up in the case of program exits. Furthermore, APIs in error-handling blocks might depend on the APIs called prior to the error being handled. As these API details are often inter-procedurally scattered and not always correctly coded by the programmers, manually inferring specifications from source code becomes hard and inaccurate, necessitating automatic specification inference.

To detect API error-handling bugs in the absence of spec-

[1]This work is supported in part by NSF grant CNS-0720641 and ARO grant W911NF-07-1-0431. Contact Author: Mithun Acharya, Tel: +1 919 515 2858.

ifications, we develop a novel framework for statically mining API error-handling specifications directly from software package repositories, without requiring any user input. Our framework adapts a *trace generator* to approximate run-time API behaviors. The trace generator uses a compile-time push-down model-checker to generate inter-procedural static traces. Data mining techniques are used on these static traces to mine specifications that define correct handling of errors for relevant APIs used in the software packages. The mined specifications are then formally verified against the same software packages (or other software packages, which use these relevant APIs) to uncover API error-handling bugs.

Two of our previous approaches [2,3] used the trace generator, adapted in this work for mining API error specifications, for different tasks. One approach [3] uses the trace generator to infer API details such as return values on API failure and success. As opposed to intra-procedural trace generation in our previous approach [3], in this paper, both trace generation and bug checking are inter-procedural. The inter-procedural analysis allows our framework to mine API error-check and clean-up details (and hence bugs) scaterred across different procedures. The other approach [2] uses the trace generator to mine API *usage scenarios* and specifications using a partial-order miner [18]. API usage scenarios dictate how a given set of APIs are used for a particular task. Both specifications and usage scenarios mined by our previous approach [2] were ordering requirements between multiple, user-specified APIs only, and not error checks. To capture different possible orderings (summarized as partial orders) among user-specified set of APIs, the static traces generated were from the *start-to-end* of the analyzed program. To detect API error-handling bugs, our framework in this paper adapts the trace generator to generate traces around *relevant* API error paths (described in Sections 3.2 and 3.3). Both our previous approaches require users to specify the APIs of interest. In contrast, the framework proposed in this paper automatically infers the relevant APIs, the APIs that fail with errors. Finally, the framework proposed in this paper employs sequence mining to infer proper API clean-up in case of program exits, not mined by our previous approaches. Frequent-sequence mining [23] (as opposed to the more costly partial-order mining) sufficed as error checking and cleaning up are unique for a given API along error paths. In summary, this paper makes the following main contributions:

Static approximation of run-time API error behaviors. We adapt previous trace generation framework to statically approximate run-time API error behaviors. Our techniques allow mining of open source systems for API error-handling bugs without requiring environment setup for system executions or availability of sufficient system tests. Furthermore, our framework to detect API error-handling bugs expects no user input in the form of specifications, programmer annotations, profiling, instrumentation, random inputs, or a set of relevant APIs.

Specification Extraction. We present novel applications of frequent-sequence mining [23] on static traces to mine specifications that dictate correct handling of API errors.

Implementation and Experience. We implement the framework and validate the effectiveness of the framework on 82 widely used open-source software packages with approximately 300K LOC in total.

The remainder of this paper is structured as follows. Section 2 starts with an example that motivates our framework. Section 3 describes the various components of our framework in detail. Section 4 presents the implementation details and evaluation results. Section 5 discusses related work. Finally, Section 6 concludes.

2 Example

This section illustrates how our framework automatically detects API error-handling bugs via mining program source code, without requiring any user-input. The only input to our tool is compilable source code of a single software or a set of software packages. Figure 1(a) shows a simple code snippet in `C` that uses APIs from a header file, say `<abcdef.h>`, namely, `a`, `b`, `c`, `d`, `e`, and `f`. Next, we explain the various components of our framework on the sample code snippet at a high level, after defining a few terms used throughout the paper. The formal details of the framework are described in Section 3.

Definitions. We identify two types of specifications that determine correct handling of API errors along all paths in the program: *error-check* specifications and *multiple-API* specifications. Error-check specifications ensure that error-check conditionals exist after each call site of an API before its return value is *used* or the `main` procedure returns. Error-check conditionals check the API return value and error flag (such as `errno`) value against their possible error values. We classify an API error (return-value error or error-flag error) to be *critical*, if the program should not proceed after the critical error and has to exit (through an `exit(0)` call, for example). Critical API errors are caused because of stressful environment conditions such as network failures, disk failures, and memory exhaustion. We classify a conditional checking against any critical API error as *critical check conditional* (CCC; we use CCC(a) to denote CCC of API `a`). In this paper, we restrict the scope of error-check specifications to the presence of critical check conditional after an API call before its return value is used or the `main` procedure returns. For example, the POSIX API `setuid` returns -1 on failure, with possible error values of `EPERM` and `EAGAIN`. Checks should exist after each call-

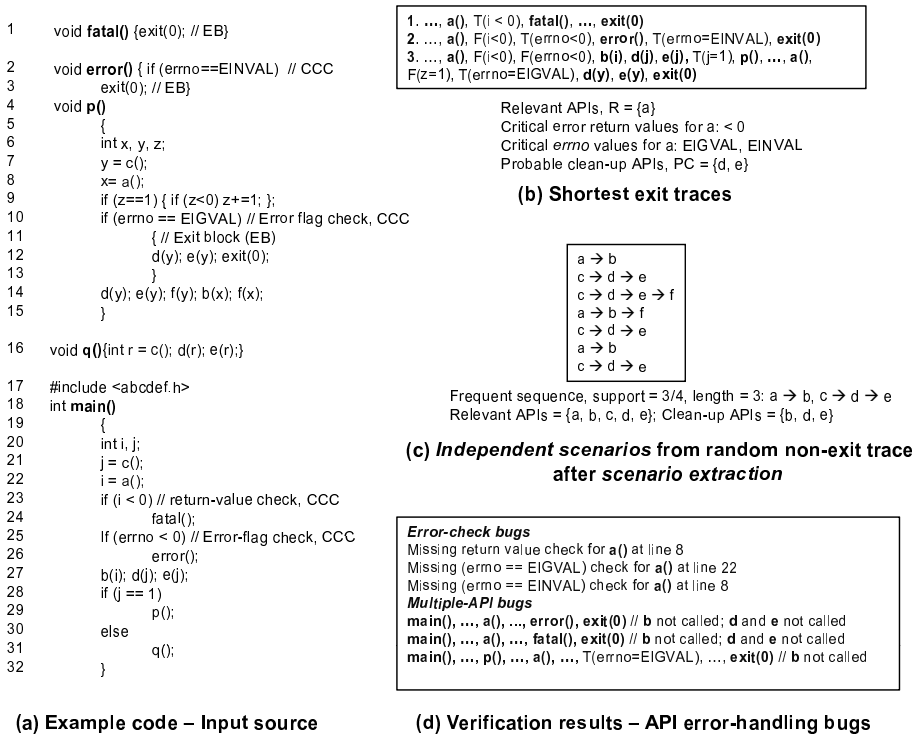

Figure 1. A simple example for illustrating our framework

site of `setuid` and these critical error conditions should be handled appropriately. Critical check conditionals should be followed by *exit blocks* (EB; we use EB(a) to denote EB of API `a`). Exit blocks handle the error and executes an exit call. Multiple-API specifications ensure that the right *clean-up* APIs are called in the exit blocks. Clean-up APIs are APIs called in the exit blocks, which may share a temporal relationship with any API called prior to the exit block. For example, the display pointer produced by the X11 API, `XOpenDisplay`, should be consumed by the `XCloseDisplay` API along all paths. Hence, each exit block reachable from a call to `XOpenDisplay` should have a call to `XCloseDisplay` (clean-up API). We define *relevant* APIs as APIs that can fail with critical errors, clean-up APIs, and APIs that share temporal relationship with clean-up APIs. Our framework automatically mines relevant APIs from the source code, and then generates specifications. A *trace* in a program is the print of all statements that exist along some control flow between any two statements, say S1 and S2. In our analysis, we restrict such traces to those that can be captured by a Finite State Machine (details in Section 3). An *exit trace* is any trace from the entry to the `main` procedure to some exit point in the program (through an `exit(0)` call, for example). A *non-exit* trace is any trace from the entry of the `main` procedure to the return of the `main` procedure.

To infer error-check specifications, our framework first gathers APIs that fail with critical API errors from the source code. Our framework generates the shortest exit trace for each exit path (in the program) that contains a critical check conditional. Figure 1(b) shows three shortest exit traces. For a predicate P, $T(P)$ means that the predicate P is true, and $F(P)$ means that the predicate is false. Hence $T($`i<0`$)$ in trace 1 means that the predicate `i<0` is true. For Trace 3, $T($`z==1`$)$ in procedure `p` implies a longer trace, which is not output by our framework. Since the critical check conditionals in exit traces pertain to API `a`, `a` is added to the set of relevant APIs (set R). Critical API errors are inferred from critical check conditionals present in exit traces. For `a`, critical API errors for the return value are all negative integer values, and `EINVAL` and `EIGVAL` are critical API errors for the error flag. Error-check specifications are then generated with the knowledge of critical API errors. The APIs present in the exit blocks for `a` are `d` and `e` (Lines 11-13). These APIs could probably be clean-up APIs (set PC) that share a temporal relationship with APIs called prior to critical check conditionals. Our framework then generates non-exit traces that involve APIs from set R and PC (`a`, `d`, and `e`), and APIs *related* to them. Two APIs are *related*, if they have some data-flow dependency between them. Non-exit traces are generated randomly (details in Section 3) from a set of all non-

exit traces until an upper limit is reached on the number of traces. Figure 1(c) shows non-exit traces generated by our framework after *scenario extraction* (see Section 3.4), with an upper limit of seven. Scenario extraction generates *independent scenarios* from non-exit traces. An independent scenario in a non-exit trace is a sequence of APIs that are related through some data-flow dependency. For example, one non-exit trace in the example code has `j=c()`, `i=a()`, `b(i)`, `d(j)`, `e(j)`, `y=c()`, `x=a()`, `d(y)`, and `e(y)` as an API invocation sequence. There are three independent scenarios in this non-exit trace, namely, (`j=c()`, `d(j)`, `e(j)`), (`i=a()`, `b(i)`), and (`y=c()`, `d(y)`, `e(y)`). Multiple-API specifications are derived from these traces using frequent-sequence mining. Based on the observation by Weimer and Necula [24], exit traces are not used to infer multiple-API specifications because programmers tend to commit mistakes along exit paths when using clean-up APIs. The frequent sequences (shown in Figure 1(c)) are mined from random non-exit traces with support 3/4, and they imply that APIs `b` should always be called after `a`, and that `d` and `e` should be called after API `c`. The error-check and multiple-API specifications are then verified against the source code to detect API error-handling bugs. Figure 1(d) shows the detected API error-handling bugs. Our framework outputs the shortest path for each bug in the program, instead of all buggy traces, thus making bug inspection easier for the users.

3 Framework

A high-level overview of our framework is shown in Figure 2. The only input to our framework is compilable source code of a single software package or a set of software packages. Our framework then finds API error-handling bugs, if any, in the source code. There are three main stages in our framework: trace generation, specification extraction, and verification, as shown by dotted boxes in the figure. The trace generation stage generates two types of traces, shortest exit traces and random non-exit traces. The specification extraction stage generates two types of specifications, error-check specifications and multiple-API specifications, inferred from the traces using different mining algorithms. In the verification stage, the inferred specifications are verified against software packages to detect API error-handling bugs. Section 3.1 introduces the trace generation mechanism, also used by our previous approaches [2, 3]. Sections 3.2 and 3.3 explain how our framework adapts trace generation to generate API exit and non-exit traces. Sections 3.4 and 3.5 describe the specification extraction and verification stages, respectively.

3.1 Trace Generation

Our framework mines error-check specifications from exit traces and multiple-API specifications from random non-exit traces. Trace generation forms the basis for generating exit and non-exit traces. Informally, a trace is a sequence of program statements between two points in a program along some control-flow path. However, generating all traces along all execution paths is an uncomputable problem and a trace can be of infinite size. Furthermore, a generated trace can be infeasible. These problems will be addressed in subsequent sections. Here we first formalize the problem of trace generation as below. The problem is to generate traces between two points in a program and then to extract relevant statements (such as API invocation, return-value checks, and exits) from each trace. To describe trace generation, we assume that the user wants to extract API (from a known set of APIs) invocation sequences from traces. We then summarize the Push-Down Model Checking (PDMC) process [7, 9], which we adapt for trace generation. Next, we introduce the concept of $Triggers$ required for trace generation. Finally, we discuss the soundness and complexity of trace generation.

Let us assume that $\mathcal{A}$ is a set of APIs. To simplify the definitions, let us assume that all APIs in $\mathcal{A}$ are empty methods, do not take any arguments, and return `void`, so that they do not have any data dependencies with other statements. We show how to generate sequences of API (from set $\mathcal{A}$) invocations along different program paths. Formally, let Σ be the set of valid program statements in the given program source code. A *trace* $t \in \Sigma^*$, a sequence of statements executed by a path p, is *feasible* if path p is feasible in the program. Let $T \subset \Sigma^*$ be the set of all feasible traces in the program. For a given $t \in T$, let $A(t) \in \mathcal{A}^*$ be the API invocations along the trace t expressed as a string. $A(t)$ can be an empty string if t does not have any invocation of APIs from the set $\mathcal{A}$. Let $T' \subseteq T$ be the set of all feasible traces such that if $t \in T'$, $A(t)$ is not empty. However, the set T' is uncomputable and $t \in T'$ can be of infinite size. A computable approximation of T' is generated from the program. $A(t)$ is extracted for all t in the approximate set, using *Triggers* (explained later in this section). We now describe PDMC process required to understand Triggers.

3.1.1 Push-Down Model Checking (PDMC)

Given a property represented using a Finite State Machine (FSM), PDMC [9] checks to see if there is any path in the program that puts the FSM in its final state. For example, if the property FSM is specified as shown in Figure 3, PDMC reports all program paths in which `a` is followed by either `b` or `c`. PDMC models the program as a Push Down Automata (PDA) and the property as an FSM. PDMC then combines

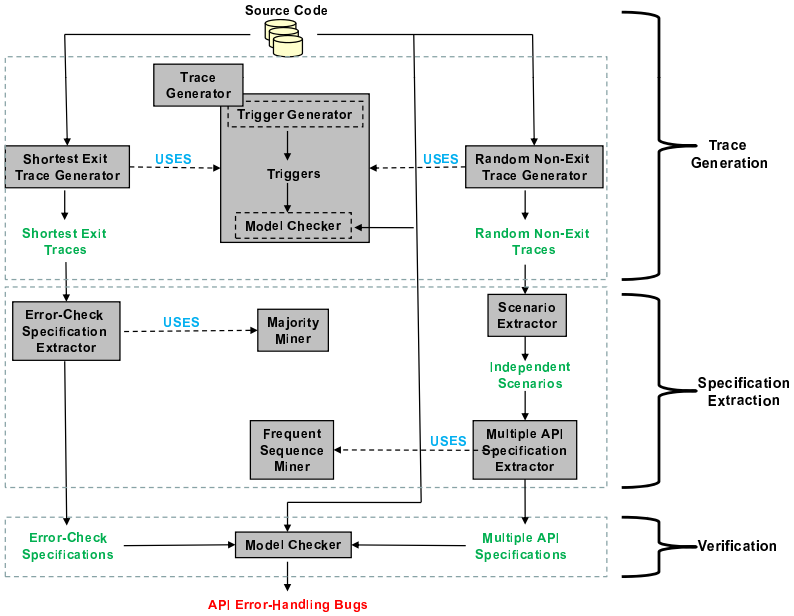

Figure 2. Our framework for detecting API error-handling bugs

the program PDA and the property FSM to generate a new PDA; the new PDA is then model checked to see if any *final configuration* in the PDA is reachable. A *configuration* of a PDA $\mathbf{P}$ is a pair $c = \langle q, \omega \rangle$, where q is the state in which the PDA is in and ω is a string of stack symbols in the PDA stack at that state. A configuration is said to be a final configuration, if q belongs to the set of final states in the FSM. If a final configuration is reachable, PDMC outputs the paths (in the program) that cause the resultant PDA to reach this final configuration. The resulting trace can either be feasible or infeasible because of data-flow insensitivity (being incomplete). However, if there is a program trace that puts the FSM in the final state, PDMC reports it (being sound). We next describe our *Triggers* technique [2] that adapts PDMC to generate API invocation sequences in a program.

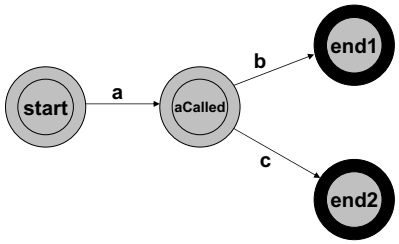

Figure 3. A property FSM with end1 and end2 as final states

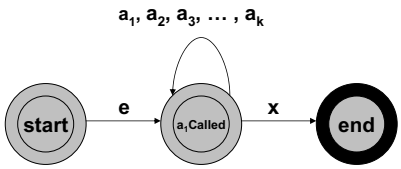

Figure 4. Trigger FSM that accepts the regular language $e(a_1 + a_2 + ... + a_k)^* x$

3.1.2 Triggers

Our goal is to generate the set $T' \subseteq T$ from the program and extract $A(t)$ for all $t \in T'$, $A(t) \in \mathcal{A}^*$, $\mathcal{A} = \{a_1, a_2, a_3, ..., a_k\}$. Let us assume that we give the FSM shown in Figure 4 to PDMC to be verified against a program $\mathcal{P}$. The FSM in Figure 4 accepts any string of the form $e(\sum_{i=1,2,...,k} a_i)^* x$, where e and x are any two points in the program. Given this Trigger FSM, PDMC outputs all program paths that begin with e and end with x in the program.

Let $B \subseteq \Sigma^*$ be all sequences of program statements in $\mathcal{P}$ that put the FSM in Figure 4, say $\mathbb{F}$, in its final state. As defined earlier, $T \subset \Sigma^*$ is the set of all feasible traces in the program, in this case, $\mathcal{P}$. If $T \cap B = \phi$, then the final state of $\mathbb{F}$ is never reached. Since B and T are arbitrary languages and T is uncomputable, deciding if $T \cap B = \phi$ is an undecidable problem. Hence PDMC restricts the form of B and T by modeling B to be a regular language accepted by $\mathbb{F}$ ($B = L(\mathbb{F})$), and T as a context-free language accepted by a PDA $\mathbb{P}$ (of program $\mathcal{P}$). In general, we have $T \subseteq L(\mathbb{P})$, which then implies $T \cap B \subseteq L(\mathbb{F}) \cap L(\mathbb{P})$. Consequently, if $L(\mathbb{F}) \cap L(\mathbb{P})$ is empty, $T \cap B$ is definitely empty. However, if $L(\mathbb{F}) \cap L(\mathbb{P})$ is not empty, $T \cap B$ could either be

empty or not. Since $L(\mathbb{F})$ is a regular language and $L(\mathbb{P})$ is a context-free language, $L(\mathbb{F}) \cap L(\mathbb{P})$ can be captured by a PDA, say **P**, and hence the final state of $\mathbb{F}$ is unreachable if and only if the PDA **P** accepts the empty language. There are efficient algorithms to determine if the language accepted by the PDA is empty [14]. Once **P** is constructed, PDMC checks to see if any final configuration is reachable in **P**. Chen and Wagner [8] use the preceding analysis to adapt PDMC for light-weight property checking. We use the preceding analysis for static trace generation. We call the FSMs such as the one used in Figure 4 as Triggers. By using Triggers, we have achieved two purposes:

- We have produced T_{ex}, the set of traces in the program that begin with e and end with x instead of $T' \subseteq T$.
- The knowledge of $\mathcal{A} = \{a_1, a_2, a_3, ..., a_k\}$ allows us to extract $A(t)$ from any $t \in T_{ex}$. Similarly, other statements like return-value checks, error-flag checks can also be retrieved from the traces.

3.1.3 Soundness

The consequence of using a context-free language for T introduces imprecision but retains the soundness of analysis. Infeasible traces might occur (being incomplete) because of data-flow insensitivity of the PDMC process, but all the program traces that put the FSM in its final state are reported (being sound). Since determining if $T \cap B = \phi$ is undecidable, no tool can be sound and complete at the same time. Consequently, there could be some infeasible API sequences. The model-checker that we use is data-flow insensitive. We implement simple data flow extensions to the PDMC process as described in Section 4. However, we do not implement a potentially expensive pointer or alias analysis. We intend to explore data-flow-sensitive model-checkers in future work for trace generation. Also, along some feasible paths, the implicit API ordering rules might be violated and APIs could be used incorrectly (producing buggy traces with actual errors). Hence the API sequences might contain certain wrong API sequences. However, we assume that most programs that we analyze are well written. Hence, we expect only few feasible paths to be buggy, if at all. We expect to handle buggy traces by selecting an appropriate min_sup value. The traces generated by PDMC with Triggers can still be of infinite size (for example if there is a loop). We address this problem in Section 4.

3.1.4 Complexity

PDMC constructs PDA $\mathbb{P}$ from the program Control Flow Graph (a directed graph $G = (N, E)$) where each node represents a program point and each edge represents a valid program statement. PDMC takes $O(E)$ time to construct the PDA $\mathbb{P}$ from the CFG G, takes $O(E \times |Q|)$ (Q is the number of states in the FSA) for computing $\mathbf{P}$, the product of FSA $\mathbb{F}$ and PDA $\mathbb{P}$, takes $O(|Q|^2 \times E)$ for deciding if the PDA $\mathbf{P}$ is empty and $O(|Q|^2) \times lg|Q| \times E \times lgN$ for backtracking. The derivations are shown by Chen [7].

Our framework has four components: the shotest exit-trace generation, random non-exit trace generation, specification extraction, and verification, described in subsequent sections. Figure 5 lists the algorithm used by our framework to detect API error-handling bugs. The algorithm summarizes the various steps in each of our framework components.

3.2 Shortest Exit Trace Generation

Our framework generates the shortest trace for each exit path (in the program) that contains a critical check conditional (CCC). We use $\mathcal{P}$ to denote the input program and $\mathbb{F}$ to denote the Trigger FSM. CCC(a) denotes CCC for API `a`. Line 4 in Figure 5 sets the Trigger to collect the shortest exit traces. A path from the entry of the `main` procedure to an exit point in the exit block of an API must always go through CCC for that API. Hence, in generating exit traces, we collect CCCs for APIs that fail with critical errors. Critical errors (return-value errors and error-flag errors) for an API are inferred from the CCCs of that API. We generate the shortest exit traces instead of enumerating all exit traces (or generating traces randomly with an upper limit) because, for an API, say `a`, which can fail with a critical error, program statements between the `main`'s entry and the invocation of `a` do not yield additional information to infer critical errors for `a`. To collect critical API errors, it suffices to generate program statements between CCC(a) and the program exit in the exit block of `a`, EB(a). The shortest exit traces for a given trigger are generated by computing the shortest path from *source* to *sink* nodes in the graph obtained after *saturating* PDA **P** [7]. When generating the shortest exit traces, all APIs that fail with critical errors are added to the set of relevant APIs (set R). Also, APIs in the exit blocks of APIs that fail with critical errors are flagged as potential clean-up APIs (set PC), which could share temporal relationship with any API called prior to the exit block.

3.3 Non-Exit Trace Generation

Non-exit traces are generated randomly from a set of all non-exit traces until the number of traces reach an upper limit (L). Random traces are generated by a random walk from *source* to *sink* nodes in the graph obtained after *saturating* PDA **P** [7]. Line 21 in Figure 5 sets the Trigger required to generate non-exit traces. The PDMC process slices the program $\mathcal{P}$ based on the transition edges in the trigger $\mathbb{F}$. APIs related to APIs in sets R and PC are cap-

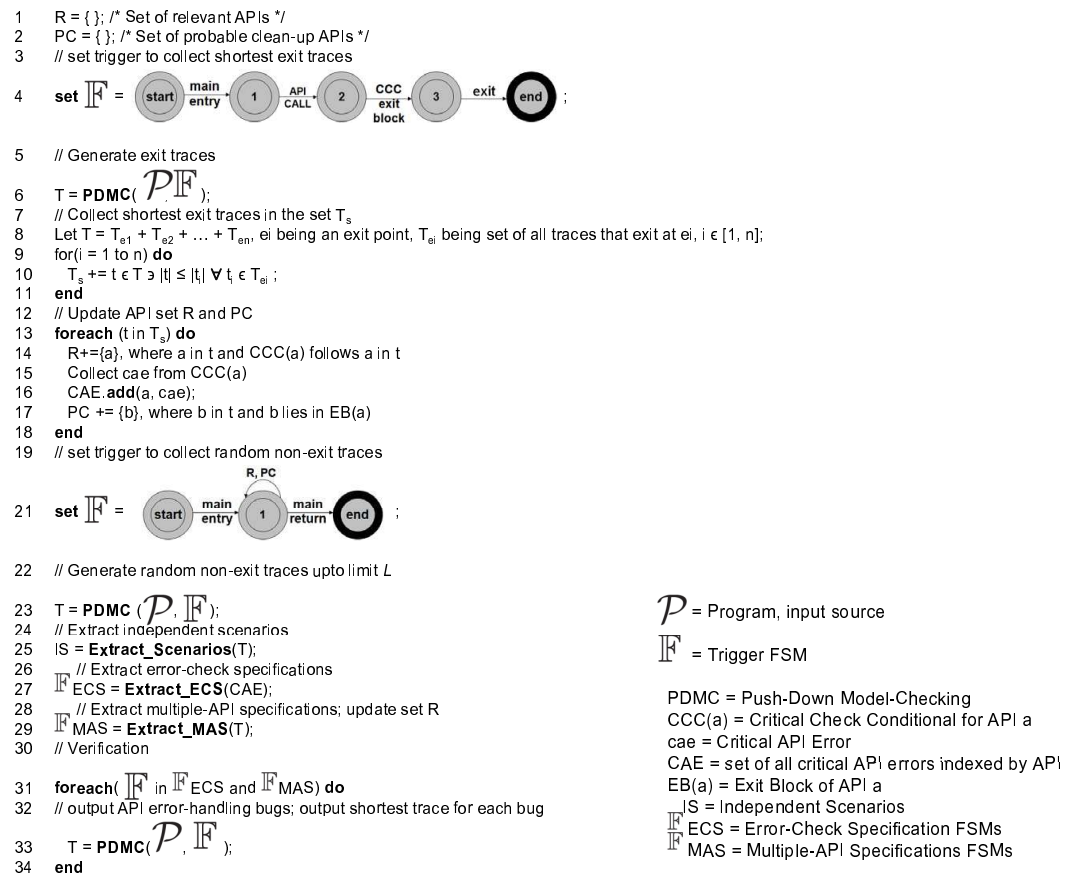

Figure 5. Algorithm for Detecting API Error-Handling Bugs

tured in the generated non-exit traces. As APIs related to R and PC could occur anywhere in the program, we generate random traces instead of shortest non-exit traces (in fact there is only one shortest trace from the entry of the `main` procedure to the return of the `main` procedure). Based on the observation by Weimer and Necula [24], exit traces are not used to infer multiple-API specifications because programmers tend to make mistakes along exit paths when using clean-up APIs. We next describe how specifications are extracted from traces.

3.4 Specification Extraction

Error-check specifications are derived from the shortest exit traces. In generating the shortest exit traces, we collect CCCs for APIs that fail with critical errors. Critical errors (return-value errors and error-flag errors) for an API are inferred from the CCCs of that API. Error-check specifications are derived from the inferred critical API errors.

Multiple-API specifications are derived from non-exit traces. A single non-exit trace generated by the model checker might involve several API *clean-up scenarios*, being often interspersed. A clean-up scenario in a non-exit trace consists of an API and its corresponding clean-up APIs in the right sequence. We have to separate different API clean-up scenarios (or match each API with its corresponding clean-up APIs) from a given non-exit trace, so that each clean-up scenario can be fed separately to the miner. We use the usage scenario extraction algorithm [2], which 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 output as an independent clean-up scenario. For example, in Figure 1, `a` is the producer API, while `b` is the consumer API for `a`. Also, APIs `d` and `e` are consumers for the producer API `c`. `f` is the consumer API once for `a` (in procedure `p`) and once for `c` (in procedure `main`). Separate producer-consumer chains are output as independent scenarios.

Multiple-API specifications are derived from independent scenarios using frequent-sequence mining [23]. Independent scenarios are obtained after applying the scenario

extraction algorithm over random non-exit traces. Let IS be the set of independent scenarios. We apply a maximal sequence mining algoirthm [23] on the set IS with user specified support min_sup ($min\_sup \in [0, 1]$), which outputs a set FS of frequent sequences that occur as subsequences in at least $min\_sup \times |IS|$ sequences in the set IS. Here we consider maximal subsequences, that is, every sequence in FS is not a subsequence of any other sequence in FS.

3.5 Verification

In Section 3.1, the PDMC process was used for trace generation. Here we use the same PDMC process for property verification. The specifications inferred by our framework represent the properties to be verified at this stage. The error-check and multiple-API specifications are verified against the software packages to detect API error-handling bugs, using the Push-Down Model Checker. The inferred specifications can also be used to verify correct API usage in other software packages, which use these relevant APIs. Our framework outputs the shortest path for each bug in the program, instead of all buggy traces, thus making bug inspection easier for the users.

4 Implementation and Evaluation

To generate static traces, we adapted a publicly available model checker called MOPS [8]. We used BIDE [23] to mine frequent sequences. The process of generating error traces from a final configuration $\langle q, \omega \rangle$ (ω is the stack content containing a list of return addresses) of PDA $\mathbf{P}$ is called *backtracking* [7]. Multiple program paths (and hence graph paths) can violate a given property specified by a FSM (such as the one shown in Figure 3), and many such violations could be similar because they indicate the same programming error. So instead of reporting all program traces that violate a given property, the MOPS model checker clusters similar traces and reports the shortest trace as a candidate trace for each violation. This mechanism would save the user's time considerably because the user has to review each trace manually. However, for our purposes, given a Trigger, we need all the traces in the program that contain the APIs specified in the Trigger. We modified the backtracking algorithm of MOPS, wherein, instead of clustering traces, we consider all program paths that satisfy the Trigger, and output a random number of traces by random walking the graph generated by the PDMC process.

Because the basic MOPS static checker is data-flow insensitive, it assumes that a given variable might take any value. Therefore, it assumes that both branches of a conditional statement may be taken and that a loop may execute anywhere between zero to infinite iterations. Data-flow insensitivity causes MOPS to output infeasible traces. Furthermore, the trace size and the number of traces can be infinite due to loops. MOPS monitors backtracking and aborts if it detects a loop. We wrote extensions to the MOPS pattern matching [8]; these extensions make it possible to track the value of variables that take the return status of an API call along the different branches of conditional constructs. For each possible execution sequence, our extensions associate a value to the variable that is being tracked using pattern matching. MOPS pattern matching allows our framework to correlate two program statements related by program variables (as an example, `FILE* fp = fopen(...)` and `fread(fp)` are related through the file pointer variable, `fp`). Our extensions enable our framework to mine properties such as "If API `a` returns `NULL`, then API `b` should always be called along the `NULL` path". The basic trace generator was also used in our previous approaches [2, 3], and here we adapt it for generating exit and non-exit traces. Our current implementation does not consider aliasing.

We have applied our framework on 10 open source packages (approximately, 100,000 LOC), mostly from the Redhat 9.0 distribution, and 72 clients (approximately, 200,000 LOC) of X11 APIs from the `X11R6.9.0` distribution. Figure 6 lists the packages used in our evaluation. Our framework mined about 100 APIs that fail with critical errors from the source code. Error-check specifications were generated for these critical APIs and verified against the software packages. 56 error-check violations were reported. We manually inspected the violations (by inspecting the source code) and found 49 violations to be real bugs, while 7 were false positives. False positives are caused because of the lack of sophisticated data-flow analysis in our model checker. Multiple-API specifications were mined as frequent sequences from non-exit traces, with a min_sup value of 0.8. 11 multiple-API specifications (9 real, 2 false) were mined resulting in 27 multiple-API violations (all violations being real bugs) in the packages used in our evaluation. We did not use the 2 false specifications in the verification phase after we verified the mined specifications against POSIX manual in UNIX and X11 Inter-client Communication Conventions Manual (ICCCM) [21] from the X Consortium standards. We next present an example of a multiple-API bug from `xrdb/xrdb.c`. Our framework mines a specification which dictates that the display pointer produced by the X11 API, `XOpenDisplay`, should be consumed by the `XCloseDisplay` API along all paths. However, our framework detects an exit path in the program where the specification is violated; `fdopen` is called after `XOpenDisplay` along this path, and the exit block of `fdopen` (reached inter-procedurally through a call to an error-handling procedure, `fatal(...)`) forgets to invoke `XCloseDisplay` before a program exit. At the time of this submission, we are still in the process of verification and validation (by manual inspection of source code), and we

ftp-0.17-17
ncompress-4.2.4-33
routed-0.17-14
rsh-0.17-14
sysklogd-1.3.31-3
sysstat-4.0.7-3
SysVinit-2.84-13
tftp-0.32-4
traceroute-1.4a12-9
zlib-1.1.3-3

(a) Packages from *Redhat-9.0* distribution

appres	beforelight	bitmap	dpsexec	dpsinfo	editres	glxgears	glxinfo
iceauth	ico	listres	luit	makepsres	oclock	proxymngr	rstart
setxkbmap	showfont	smproxy	texteroids	twm	viewres	x11perf	xauth
xbiff	xcalc	xclipboard	xclock	xcmsdb	xconsole	xditview	xdpyinfo
xev	xeyes	xf86dga	xfd	xfindproxy	xfontsel	xfsinfo	xfwp
xgamma	xgc	xhost	xinit	xkbevd	xkbprint	xkbutils	xkill
xload	xlogo	xlsatoms	xlsclients	xlsfonts	xmag	xman	xmessage
xmh	xmodmap	xpr	xrandr	xrdb	xrefresh	xset	xsetmode
xsetpointer	xsetroot	xstdcmap	xterm	xtrap	xvidtune	xvinfo	xwud

(b) X11 clients from *X11R6.9.0* distribution

Figure 6. Open source packages used in our evaluation

expect the number of API error-handling bugs that we report to increase.

5 Related Work

Previous work has mined API properties from program execution traces. For example, Ammons et al. [5] mine API properties as probabilistic finite state automata from execution traces. Perracotta developed by Yang et al. [26] mines temporal properties (in the form of pre-defined templates involving two API calls) from execution traces. These approaches require setup of runtime environments and availability of sufficient system tests that exercise various parts of the program. Furthermore, stressful environment conditions need to be simulated to expose API-error behaviors. Such simulations of stressful environment conditions might not sufficiently expose API errors. In contrast, our new framework mines properties related to correct error handling from static traces without suffering from the preceding issues.

Related approaches developed by other researchers also mine properties from static source code for finding bugs. For example, PR-Miner developed by Li and Zhou [16] mines programming rules (involving multiple code elements such as function calls) from source code. DynaMine developed by Livshits and Zimmermann [17] mines simple rules (involving mostly function-call pairs) from software revision histories. Ramanathan et al. [19, 20] mine function preconditions or function precedence protocols (involving function-call pairs) from static traces. Both Chang et al. [6] and Thummalapenta and Xie [22] detect missing conditionals by mining source code. None of these previous approaches find bugs related to API error handling as targeted by our new framework. A detailed comparison of our new framework with our previous approaches [2, 3] was already given in Section 1.

From exception-handling code in Java applications, Weimer and Necula [24] mine temporal safety rules that involve pairs of API calls used in Java's exception handling. In contrast, our new framework focuses on C applications and mine multiple-API specifications (as frequent sequences) in error handling beyond pairs of API calls. In addition, our framework mines error-check specifications, not being mined by their approach.

A number of approaches [4, 13, 25] apply static analysis or model checking on the API implementation code to synthesize permissive API usage patterns that are allowed by the API implementation. Different from these approaches, our framework analyzes API client code (rather than API implementation code) and applies a miner on static traces extracted from the client code. Mining from API client code is complementary to mining from API implementation code, and mining API client code can be applied where API implementation code is not available.

6 Conclusions and Future Work

We have described our novel framework to detect API error-handling bugs in software packages without requiring any user input. Our framework uses a compile-time push-down model checker to generate inter-procedural static traces, which approximate run-time API error behaviors. Data mining techniques are used on these static traces to mine specifications that define correct handling of errors for

relevant APIs used in the software packages. The mined specifications are then formally verified against the same (or other) software packages to uncover API error-handling bugs. We have implemented our framework, and validated the effectiveness of the framework on 82 widely used open-source software packages with approximately 300K LOC in total. The model checker used in our framework is data-flow-insensitive. This limitation leads to infeasible traces. In future work, we plan to explore the utility of data-flow-sensitive model checkers such as BLAST [12] for trace generation. Although we have applied our framework on clients written in C, the basic idea is generally applicable to even object-oriented languages such as Java and C#.

References

[1] *IEEE Computer Society, IEEE Standard Glossary of Software Engineering Terminology, IEEE STD 610.12-1990*. December 1990.

[2] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 25–34, 2007.

[3] M. Acharya, T. Xie, and J. Xu. Mining interface specifications for generating checkable robustness properties. In *Proc. International Symposium on Software Reliability Engineering (ISSRE)*, pages 311–320, 2006.

[4] R. Alur, P. Cerny, P. Madhusudan, and W. Nam. Synthesis of interface specifications for Java classes. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 98–109, 2005.

[5] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 4–16, 2002.

[6] R. Y. Chang and A. Podgurski. Finding what's not there: A new approach to revealing neglected conditions in software. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 163–173, 2007.

[7] H. Chen. *Lightweight Model Checking for Improving Software Security*. PhD thesis, University of California, Berkeley, 2004.

[8] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *Proc. ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, 2002.

[9] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking push down systems. In *Proc. International Conference on Computer Aided Verification (CAV)*, pages 232–247, 2000.

[10] J. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proc. USENIX Windows Systems Symposium*, pages 69–78, 2000.

[11] J. Haddox, G. Kapfhammer, C. Michael, and M. Schatz. Testing commercial-off-the-shelf software components. In *Proc. International Conference and Exposition on Testing Computer Software*, 2001.

[12] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proc. Workshop on Model Checking Software*, pages 235–239, 2003.

[13] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 31–40, 2005.

[14] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[15] P. Koopman and J. DeVale. The exception handling effectiveness of POSIX operating systems. *IEEE Trans. Softw. Eng.*, 26(9):837–848, 2000.

[16] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 306–315, 2005.

[17] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *Proc. European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 296–305, 2005.

[18] J. Pei, H. Wang, J. Liu, K. Wang, J. Wang, and P. Yu. Discovering frequent closed partial orders from strings. *IEEE Transactions on Knowledge and Data Engineering*, 18(11):1467–1481, 2006.

[19] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *Proc. International Conference on Software Engineering (ICSE)*, pages 240–250, 2007.

[20] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pages 123–134, 2007.

[21] D. Rosenthal. *Inter-client communication Conventions Manual (ICCCM), Version 2.0. X Consortium, Inc.* 1994.

[22] S. Thummalapenta and T. Xie. NEGWeb: Static defect detection via searching billions of lines of open source code. Technical Report TR-2007-24, North Carolina State University Department of Computer Science, Raleigh, NC, August 2007.

[23] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. International Conference on Data Engineering (ICDE)*, pages 79–90, 2004.

[24] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 461–476, 2005.

[25] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 218–228, 2002.

[26] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proc. International Conference on Software Engineering (ICSE)*, pages 282–291, 2006.