SCMiner: Localizing System-Level Concurrency Faults from Large System Call Traces

Tarannum Shaila Zaman, Xue Han, Tingting Yu
Department of Computer Science
University of Kentucky
tarannum.zaman@uky.edu, xha225@g.uky.edu, tyu@cs.uky.edu

***Abstract*—Localizing concurrency faults that occur in production is hard because, (1) detailed field data, such as user input, file content and interleaving schedule, may not be available to developers to reproduce the failure; (2) it is often impractical to assume the availability of multiple failing executions to localize the faults using existing techniques; (3) it is challenging to search for buggy locations in an application given limited runtime data; and, (4) concurrency failures at the system level often involve multiple processes or event handlers (e.g., software signals), which cannot be handled by existing tools for diagnosing intra-process (thread-level) failures. To address these problems, we present SCMiner, a *practical* online bug diagnosis tool to help developers understand how a system-level concurrency fault happens based on the logs collected by the default system audit tools. SCMiner achieves online bug diagnosis to obviate the need for offline bug reproduction. SCMiner does not require code instrumentation on the production system or rely on the assumption of the availability of multiple failing executions. Specifically, after the system call traces are collected, SCMiner uses data mining and statistical anomaly detection techniques to identify the failure-inducing system call sequences. It then maps each abnormal sequence to specific application functions. We have conducted an empirical study on 19 real-world benchmarks. The results show that SCMiner is both effective and efficient at localizing system-level concurrency faults.**

I. Introduction

Due to the worldwide spread of multi-core architecture, concurrent systems are becoming more pervasive. Debugging concurrent programs is difficult because of the non-deterministic behavior and the specific sequences of interleaving in the execution flow. It often takes a tremendous amount of time and effort to reproduce and localize concurrency faults [1].

A concurrency fault may occur either during testing or in the production environment. If the failure occurs in production, developers often have to diagnose it in a different (debugging) environment to identify the root cause. However, this is challenging primarily because a program can behave differently in a different environment for each execution, especially for a concurrent system with non-deterministic behaviors. In addition, customers may not be willing to share their inputs for being used to reproduce failures due to privacy concerns. Therefore, it is hard to apply existing offline debugging tools [2]–[4] to diagnose concurrency failures that cannot be reproduced outside the production environment. While previous research [2], [5]–[12] have been conducted to help developers in debugging concurrency faults, it takes advantage of fine-grained logging for deterministic record-and-replay, which is infeasible in the production environment due to the unbearable performance overhead.

To relieve the burden of debugging, there has been some research on analyzing the runtime information and automatically localize faults [13]–[16]. For example, Falcon [14] collects both passing and failing execution traces by instrumenting each memory access of a concurrent program. It then uses statistical analysis to rank interleaving patterns involving the memory accesses. This approach is intended to be used in the pre-deployment environment because of the heavy-weighted instrumentation. Cooperative Concurrent bug Isolation (CCI) [13] leverages statistical debugging and views interleavings as predicates, which are collected at the runtime and analyzed to find the location of the concurrency fault. CCI induces less overhead than Falcon but still requires code instrumentation on the predicates. Therefore, the two approaches can be impractical for being used in the production environment. In addition, both Falcon and CCI require multiple failed and passed runs to perform the statistical analysis. However, this assumption often does not hold in the production environment – it is difficult to obtain multiple failed runs because a concurrency fault often manifests itself under specific interleavings and inputs [17].

In this work, we propose SCMiner, a practical online failure diagnosis tool to help developers understand why a concurrency failure occurs in production and localizes the cause of the failure in specific application functions. SCMiner focuses on inter-process concurrency faults, where multiple operating-system components (e.g., processes, software signals, and interrupts) incorrectly share resources [18]. The main difference between an inter-process (system-level) concurrency fault and an intra-process (thread-level) concurrency fault is that a system-level concurrency fault corrupts the persistent storage and the other system-wide resources, which can crash the entire system; whereas a thread-level concurrency fault often corrupts the volatile memory within a process [17]. Research has shown that more than 73% of the race conditions reported in the Linux distributions were system-level races [11].

SCMiner works as follows. When an anomaly (failure) is discovered by the user, SCMiner is triggered to perform online fault localization that outputs a list of abnormal system call sequences and their associated application functions ranked in

This research is supported in part by the NSF grant CCF-1652149.

terms of their likelihood of causing the failure. To achieve this goal, SCMiner analyzes a window of recent system calls. The rationale behind our approach is twofold. First, a system call trace can be easily collected via system audit tools [19] in production (e.g., cloud computing infrastructures) with low overhead (19%). Second, system-level concurrency failures are often caused by incorrect synchronizations of system calls on shared resources between two application processes. Therefore, we can detect buggy locations by monitoring system calls. However, it is challenging to identify the abnormal system call sequences from a trace potentially containing millions of system calls. Even if the sequences are identified, a modern server system typically consists of tens of thousands of functions – mapping a sequence to specific functions in the programs is a non-trivial task.

To address the above challenges, SCMiner is designed to have two major phases. In the first phase, SCMiner uses principal component analysis (PCA) – an unsupervised learning approach [20] to identify abnormal system call sequences. Since the number of system calls in the trace can be enormous, SCMiner splits the trace into a list of execution segments and generates a feature vector representation for each segment, where each element in the vector is a system call sequence. The segments together with their feature vectors are used to perform PCA for identifying abnormal system call sequences. PCA is efficient because its runtime is linear with the number of vectors so the detection can scale to large traces.

In the second phase, SCMiner maps each abnormal sequence to specific application functions. Since SCMiner does not assume the availability of source code, it is impossible to use static analysis to link system calls to application functions. Instead, SCMiner obtains multiple system call traces outside the production environment by using binary instrumentation for building a map between system calls and function names. However, Due to inconsistencies between production and non-production environment and the lack of inputs, an exact matching is almost impossible. To address this problem, SCMiner uses frequent pattern mining to extract the frequently executed system calls from each function as a *function signature*. The function signatures can serve as a high-level matching to detect and rank a list of functions that potentially map to an abnormal system call sequence.

SCMiner has several distinguishing features, which make it more advantageous over existing approaches. First, SCMiner does not require developers to reproduce bugs on their side to achieve fault localization. Second, SCMiner uses the default *auditd* [19] daemon in Linux and does not require heavy program instrumentation, which makes the tool transparent and practical for production use. Third, existing techniques often require multiple failing and passing executions to localize faults [6], but it is hard to collect multiple failing executions especially for concurrent programs. Instead, SCMiner only assumes the existence of one failing system call trace generated by the *auditd* daemon. Finally, SCMiner can capture the buggy functions for inter-process failures, whereas existing techniques [6], [15], [21], [22] focus on intra-process failures.

```
syscall=47, ..., pid=2911, comm="gedit" , name="/lib64", inode=269100
syscall=59, ..., pid=11589, comm="bash", name="/lib64/ld.so", inode=269126
syscall=2, ..., pid=11587, comm="bash" , name="~/bash-3.0/history", inode=1721943
syscall=2, ..., pid=11589, comm="bash", name="~/bash-3.0/history", inode=1721943
syscall=1, ..., pid=11587, comm="bash", name="~/bash-3.0/history", inode=1721943
syscall=1 , ., pid=11589, comm="bash", , name="~/bash-3.0/history", inode=1721943
syscall=3, ., pid=11589, comm="bash", , name="~/bash-3.0/history", inode=1721943
syscall=3, ., pid=11587, comm="bash", , name="~/bash-3.0/history", inode=1721943
```

47=rcvmsg, 59 = execve, 1 = write, 2 = open, 3= close.

Figure 1. A partial system call trace

To evaluate SCMiner, we conducted an empirical study on 19 applications with known real-world concurrency failures. Our results show that SCMiner effectively identifies the abnormal system call sequences and their associated application functions leading to the concurrency failures. We also found that the use of optimization and function signature techniques can improve the effectiveness and efficiency of SCMiner. Finally, we found that SCMiner was highly robust in handling system call traces with different sizes. Overall, we consider these results to be strong and they indicate that SCMiner could be a useful approach for helping developers to automatically localize system-level concurrency failures in production given an arbitrarily-sized system call trace.

In summary, this paper makes the following contributions:

- We propose SCMiner, the first fully automated tool for fault localization in multi-process applications.
- We implement SCMiner and conduct an empirical study to demonstrate its effectiveness and efficiency on real-world Linux applications [23].

II. Background and Motivation

In this section, we first define our problem and then show a motivating example. We also discuss the Principle Component Analysis (PCA) briefly.

A. Problem Statement

We define the production-level fault localization in multi-process applications as follows. Given the binaries of a set of Processes under Debugging (PuDs) and system call traces generated by these PuDs from the system built-in *auditd* daemon, we compute a short system call sequence S leading to the failure and the associated the application functions F of the system calls in S.

We assume that a concurrent system consists of a set of processes $\{P_1 \dots P_m\}$ and a set of software signals $\{S_1 \dots S_n\}$. Each process may create multiple threads, but for ease of presentation, we focus only on the process-level concurrency failure in this work while assuming each process has one thread. A *failing process* P_F is a process that generates the failure (or anomaly).

A system call trace contains a sequence of system calls generated from all applications running on the system. Each entry in the trace includes system call number, process ID, process

name, parent process ID, resource name, inode number, and execution command parameters. Each system call number is mapped to a specific system call name, which can be obtained from the system call table [24].

System-level concurrency faults. A system-level concurrency fault occurs when multiple processes, signals, or interrupts access a system-wide resource (e.g., file, device, etc.) without proper synchronization [25]. Such resources are often accessed through system calls. Thus, handling system-level concurrency fault requires the modeling of read/write effects and synchronization operations involving system calls. For example, the `lstat` system call on file f reads the metadata of f. The `clone` system call creates a new process inode under the $/proc$ directory (write). Synchronization operations control process interactions through kernel process scheduler. Common process-level synchronization primitives include `fork`, `wait`, `exit`, `pipe`, and `signal`.

B. A Motivating Example

Debian - 283702 [26] is a real-world bug in `bash` version 3.0-10. `Bash` is an intuitive and flexible standard GNU shell for common users [27]. It keeps a history of executed commands in a history file so that users can easily view the commands that are recently executed. However, problem occurs when multiple bash shells execute concurrently and corrupts the shell history file.

Figure 1 shows a piece of system call trace recorded by Linux `auditd` [19]. The full trace contains around 3,000K system calls from 31 processes recorded within 30 minutes while `bash` was actively running. To simplify presentation, we show system call number, process ID, process name, resource name, and inode number. The trace can grow quickly depending on how users interact with the shell and the behaviors of other programs running in the system.

When applying SCMiner to diagnosing the bug generated by `bash`, the goal is to identify abnormal system call sequences leading to the concurrency failure and their associated application functions. In Figure 1, the system call sequence (the grey area) <open(file), write(file), write(file)> from two different bash processes indicates the root cause of the failure. When one bash process (pid #11589) opens *bash-3.0/history* file before writing to it, another bash process (pid #11587) opens this file too and writes to it. The failed execution produces only one history message, whereas two messages are expected from the two processes. This is because the second process overwrites the message generated by the first process. The abnormal system call sequence is then mapped to the application functions, where the root cause is stemming from the function `history_do_write` in the `bash` application. This function is responsible for reading and writing the bash history file.

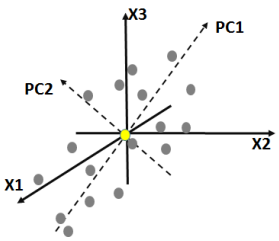
Challenges. In practice, it is difficult to localize the root cause of abnormal system call sequence and the associated application from only system call traces. The first challenge is to identify the processes involved in the erroneous execution. In the above example, only the bash processes are actually relevant. Therefore, we need to quickly weed out irrelevant processes. Moreover, in some cases, the failing process might not be the process that contains the bug. A bug in one process may propagate to another process (e.g., when a corrupted file generated by `bash` is accessed by a `cat` process and it is the `cat` process that reports the error). The second challenge is that a system call trace can easily become massive. Identifying the abnormal system call sequences are difficult especially in the absence of multiple failed executions, where existing fault localization techniques [13], [14] cannot be applied. Third, even if the abnormal system call sequences are identified, searching the buggy functions associated with them among the large number functions in the target application is challenging. For example, the `bash` program contains 456 functions. Since the Linux system has only 33 system calls and a sequence could appear in many functions, an exact match between the system call names and the abnormal system call sequences could return a number of irrelevant functions.

C. Principal Component Analysis

Principal component analysis (PCA) analyzes a data matrix (X) where each row is an observation and each column is a feature. The data points in X are described by several inter-correlated quantitative dependent variables (features) [28]. If we have data with a large number of features, some might be correlated. The correlation between features can cause redundancies in the information. Therefore, in order to reduce the computational cost and complexity, we can use PCA to transform the original features into their independent, linear combinations (PCs) [29]. For example, in Figure 2a, we displayed a 3-dimensional variable space and plotted the observations.

Applying PCA to X yields a set of m principal components. The first principal component (e.g., PC_1 in Figure 2a) captures the variance of the data to the greatest degree possible on a single axis. The next principal components (PC_2 to PC_m) then each captures the maximum variance among the remaining orthogonal directions. Variance measures how far a data set is spread out, which provides us a general idea of the spread of the data [30]. Each observation (a dot in Figure 2a) may now be projected onto the PCs to obtain a coordinate value along with each PC-line. This new coordinate value is known as the PCA score [31].

In this way, we can identify the first $(k-1)$ principal components and conclude that the k^{th} principal component corresponds to the maximum variance of the residual. The difference between the original data and the data mapped onto the first $(k-1)$ principal axes is called the residual [32]. Here, k is the number of dimensions required to capture at least $n\%$ of the variance in data [33], [34]. Therefore, in the case of abnormal items detection, where these items are assumed to be rare, PCA can capture the dominant items and construct a $(k-1)$-dimensional *normal* subspace S_d. The remaining dimensions construct the *abnormal* subspace S_a. The abnormal items can be identified by calculating the

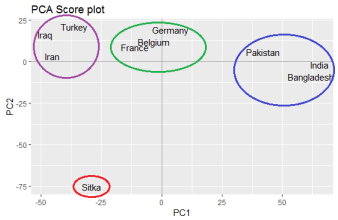


(a) First two PCs in a plane

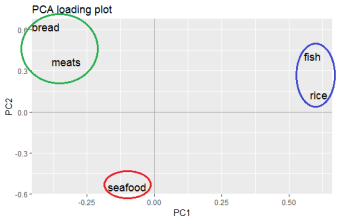
Data Matrix

	rice	bread	fish	meats	seafood
Bangladesh	78	15	65	25	2
India	72	24	80	18	4
Pakistan	61	28	69	45	2
Iran	3	65	15	60	5
Iraq	14	71	12	78	1
Turkey	28	78	18	69	2
Sitka	1	20	1	20	60
Germany	16	82	78	22	20
Belgium	20	68	59	36	13
France	10	61	53	41	8

(b) A sample data set $\mathcal{T}$



(c) Score plot of PC1 and PC2



(d) Loading plot of PC1 and PC2

Figure 2. Principle Component Analysis

distance of each item form the normal subspace. The item with the longest distance from S_d is marked as an abnormal item.

An example. Figure 2b shows an example of applying PCA to an example of dataset $\mathcal{T}$. The dataset has five features and 10 observations, which represents food consumption habit of people from different countries. Each data point in the table represents the percentage of the population in a country, who eat a specific kind of food. After applying PCA to $\mathcal{T}$, we find that the first two principal components (PCs) can explain almost 85% of the data variance. Hence, k is set to 2, which divides the original data set into $(k-1)$ normal sub-spaces and the rest as abnormal sub-spaces.

When plotting the PCA score vector for the example of Figure 2b, data points that are correlated are placed together. Figure 2c shows that countries from the same regions are grouped together. This is because people in the same region eat the same kind of food. On the other hand, the country inside the red circle, which is far from the other countries, and has the longest distance from the first PC, indicates that it has a distinct food consumption criteria (i.e., an "abnormal" item). Likewise, we can find the correlation between the features (e.g., rice, bread, etc.) by plotting the loading vector of the first two PCs. The loading vector contains the data in a rotated coordinate [35]. Features contributing similar information are grouped together, which means they are correlated. In this example, the feature, *seafood* separates the country *Stika* from the other countries. This country is characterized as having a high consumption of seafood. Therefore, we can conclude that *Stika*, which is the farthest country from the normal subspace in Figure 2c, has some rare kind of food habit. By observing the loading plot 2d, we can identify that, the feature *seafood* has the strongest impact to make *Stika*'s food consumption criteria rare.

III. SCMiner Approach

Figure 3 provides an overview of SCMiner. It consists of two major steps: 1) Identifying abnormal system call sequences; 2) Mapping abnormal sequences into a ranked list of buggy functions. To carry out the first step, SCMiner processes the system call trace into trace segments that are suitable for PCA by using filtering and a set of optimization techniques. It uses PCA to identify a set of potential abnormal inter-process system call sequences. To map these sequences into the application's functions, SCMiner performs dynamic analysis outside the production environment to extract function signatures, where each function signature indicates the frequently executed system call sequences within that function. It then uses function signatures to match against the abnormal system call sequences to identify and rank a list of functions that are likely to be the root cause of the system-level concurrency failure.

A. Identifying Abnormal System Call Sequences

Algorithm in Figure 4 shows the steps of identifying abnormal system call sequences. The input to the algorithm includes a set of system call traces T collected by built-in tools, such as linux *auditd* daemon [19]. The output is a set of potential abnormal system call sequences Seq_a. SCMiner first merges the traces into a single trace according to their timestamps in ascending order. It then extracts information that is relevant to system-level concurrency faults from the raw system call traces (Line 4). Next, it groups related system calls from the extracted information to construct feature vectors, i.e., a data table, for PCA (Lines 5-9). Specifically, SCMiner splits the trace into segments (Line 5), where the segments are expected to contain similar program behaviors (i.e., handling an HTTP request), so system calls grouped into each segment are intrinsically determined by program logic. SCMiner then encodes the feature vector by generating a set of system call sequences from each segment and counting their appearance (Lines 6-9). Next, we apply PCA to analyze the feature vectors for finding the most uncommon segments (Line 10). Finally, from those selected uncommon segments, SCMiner identifies the unique system call sequences, which describe the data points that deviate from the others.

1) Extracting Relevant System Calls: Since our target is diagnosing system-level concurrency failures, we need to identify system calls and their associated processes that can potentially lead to system-level concurrency failures. As discussed in Section II-A, system-level concurrency faults are due to incorrectly shared resource accesses between processes, so a system call s is considered "relevant" if its associated shared resource (passed as a parameter) is accessed by at least one other process that is different from the one associated with s.

In addition, the `execve` system call is always considered to be relevant because it indicates the start of the execution

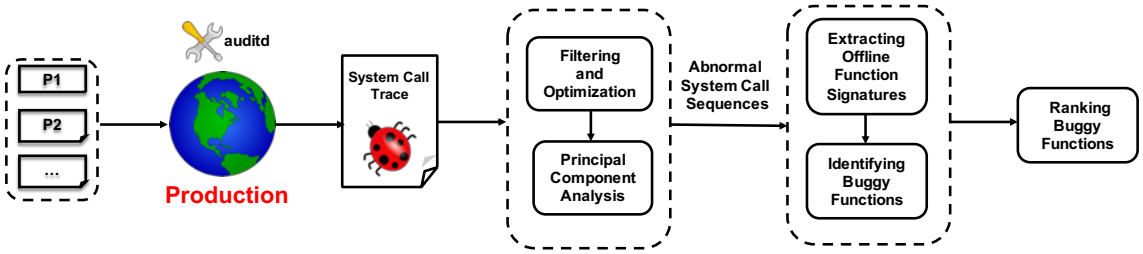


Figure 3. The overview of SCMiner framework

Abnormal System Call Identification Algorithm

1: **Inputs:** T
2: **Outputs:** Seq_a
3: **begin**
4: $T_R \leftarrow$ `ExtractTrace` (T)
5: $Seg \leftarrow$ `CreateSegments` (T_R)
6: **for** each $seg_i \in Seg$
7: $List_{seq_i} \leftarrow$ `GenerateSequences` (seg_i)
8: $PVector$.`update` $(List_{seq_i})$
9: **endfor**
10: $PC \leftarrow$ `ComputePCA` $(Seg, PVector)$
11: $Seq_a \leftarrow$`IdentifyAbnormalSeq` (PC)
12: **return** Seq_a

Figure 4. Identifying abnormal system call sequences

of a program, which will be used to build feature vectors for PCA. Therefore, SCMiner iterates through all system call entries in the traces and retain only relevant system calls. Our observation on 19 real-world Linux applications shows that, on average, only 23% (Column NOS_f of Table III) system calls are relevant. In the example of Figure 1, the system calls related to `bash` are retained for further analysis.

2) PCA-Based Anomaly Detection: We use principal component analysis (PCA) – an unsupervised learning approach to identify abnormal system call sequences from the relevant traces. We use unsupervised learning because it does not require manually labeling the data to build training sets, which needs extensive manual effort and the large training data is sometimes difficult to obtain in the production environment. The key idea of using PCA in our context is to discover the statistically dominant system call segments and thereby detect the rare segments, as well as the abnormal system call sequences (i.e., outliers) inside rare segments. The insight behind using PCA is that we observe low effective dimensionality in the data table, where each row (i.e., dimension) is a system call segment corresponding to a certain program behavior (e.g., an HTTP request) and each column is a candidate abnormal system call sequence.

Representing system call traces. We need to convert the trace containing relevant system calls to a numerical representation suitable for applying PCA detector. The whole set of system calls in the trace can be represented by an *M*N* matrix (Section II-C). In SCMiner, each row (i.e., observation) in the matrix corresponds to the trace segments by splitting the traces. Each segment contains a set of consecutive system calls describing a certain program behavior (e.g., processing an HTTP request).

For each segment, SCMiner generates a set of system call sequences with different lengths to create feature vectors, where each index in the vector represents a system call sequence and the corresponding value represents the number of times the sequence appears in the segment.

Table I shows an example of the numerical representation of a system call trace. Here, each row indicates a vector representation of a trace segment. Each item in the vector is a sequence of system calls, where a system call $\texttt{sc}_{sv_n}$ indicates system call sc accesses a shared resource sv from the process ID n.

Identifying trace segments. SCMiner splits the extracted system call traces into fine-grained segments of closely related system calls. To do this, for each trace, SCMiner divides it into a set of segments based on the execution system call `execv`, where each segment begins with `execv`. The `execv` is called when a new process starts and the first parameter of `execv` is the execution command. The intuition is that most segments go through similar program execution paths and process interleaving patterns. This results in high correlation and thus low intrinsic dimensionality, which is suitable for applying PCA. For example, each time when the user issues a command in `bash`, it will cause the execution to start from main for triggering the `execv` system call. On the other hand, the minority components may contain sequences with interleaved system calls, which are the root causes of concurrency fault. However, any bug can occur during the transition from one process to another, which means the `execv` system call may also present in the buggy system call sequence. To make sure that this kind of system call sequence is detected by our technique, we keep the last system call S_s form the previous segment as the first system call of the new segment.

Generating vector representations. SCMiner generates a feature vector representation for each trace segment. Each item (feature) in the vector is a system call sequence, which is a candidate of the abnormal sequence. To generate a list of features F for each vector, SCMiner first identifies sequences of semantically related system calls according to the shared resources. The intuition behind this is that most system-level concurrency failures occur in the case of a particular interleaving of system calls accessing a shared resource [36].

The output of this step is a set of system sequences (Seq_{sv}), where system calls in each sequence access the same shared resource.

Each sequence in Seq_{sv} can still be long and may not be helpful in understanding the bug. For example, in the Apache [37] server bug, the length of sequence with respect to a shared resource is 983. To reduce the size of the sequence encoded as a feature in the vector, SCMiner utilizes the A-priori candidate generation algorithm [38] to generate a set of shorter sequences for each S_{sv}. Basically, the A-priori candidate generation algorithm uses a lattice structure to enumerate the list of all possible item-sets [39], resulting in an overly expensive computational cost O(2^N), where N is the number of system calls in Seq_{sv}.

To minimize the number of system call sequences and reduce the computational cost, SCMiner employs three optimization methods. First, the traditional A-priori algorithm [38] exhaustively computes the short sequences regardless of the orders of the system calls in each execution. However, we need to consider the program execution flow and thus keep only system call sequences actually appeared in the trace. Therefore, the computational cost is reduced to O(N^2). For example, given a sequence $\{S_1, S_2, S_3, S_4\}$ in Seq_{sv}, our modified candidate generation algorithm will output six instead of 16 sequences: $\{S_1,S_2\}$, $\{S_1, S_2, S_3\}$, $\{S_1, S_2, S_3\}$, $\{S_2, S_3\}$, $\{S_2, S_3, S_4\}$, $\{S_3, S_4\}$. Each sequence is encoded as a feature in the feature vector.

Second, SCMiner removes the system call sequences that are not relevant to system-level concurrency bugs. Specifically, a sequence is removed if both of them involve *read* access.

Third, we propose a *sequence abstraction method* to minimize the size of Seq_{sv} and thus reduce the number of short sequences generated by the A-priori algorithm. The key idea is to detect system call sequences that are frequently executed sequences in all Seq_{sv}s and replace each frequent sequence with a symbolic name. We use frequent pattern mining algorithm [38] to obtain the frequent sequences. For example, given Seq_{sv_1} = $\{S_1, S_2, S_3, S_4\}$ and Seq_{sv_2} = $\{S_2, S_3, S_5\}$. Suppose $\{S_1, S_2\}$ is a frequent pattern, it is replaced with a symbolic name A. As a result, Seq_{sv_1} = $\{S_1$, A, $S_4\}$ and Seq_{sv_1} = $\{$A, $S_5\}$.

In this case, the cost of candidate generation algorithm can be reduced to O((N−P)^2), where N is the number of system calls and P is the number of frequent patterns. At the end of the first phase, if an abnormal system call contains a symbolic name, it will be replaced with the real system calls.

Ultimately, a feature vector is generated for each trace segment, in which each item (or observation) corresponds to a system call sequence extracted from the segment and the value of the item indicates the number of times the sequence appears in the segment. The size of the vector is the unique number of system call sequences from all trace segments.

Applying PCA detector. We create a Feature Matrix D to perform PCA, where each row corresponds to a feature vector from a trace segment. In the example of Table I, each column is a feature (i.e., candidate system call sequence) and each

Table I
AN NUMERICAL REPRESENTATION OF A SYSTEM CALL TRACE

<open$_{f_2}$, read$_{f_2}$>	< write$_{f_2}$, open$_{f_2}$ >	<read$_{f_2}$, write$_{f_2}$, stat$_{f_2}$>.
1	0	2 .
...	...	...

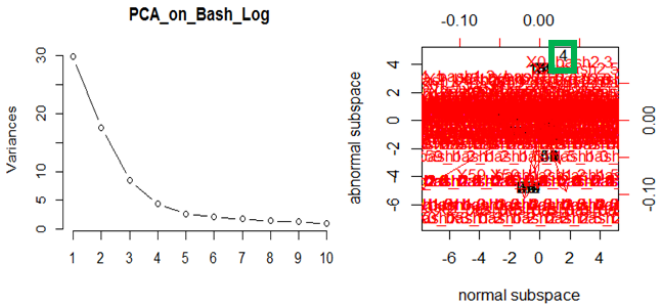


Figure 5. Variance plot and biplot of PCA on bash traces

row is an observation (i.e., trace segment). PCA finds a low-dimensional representation of the Matrix D that contains as much as possible of the variation [40].

In our benchmark programs, even though there are 200 segments on average, we found that 85% of the variance can be captured by five principal components on average which shows the low effective dimensionality in the feature metrics of our benchmarks. For our feature vector, each dimension corresponds to a certain execution sequence in the program. As the execution sequences are determined by the program logic, the sequences in a group are correlated. In the passing executions, it is natural that we find most of the sequences to be highly correlated with each other. For example, Figure 5 (left one) shows the plot of the variances (y-axis) associated with the PCs (x-axis). This indicates that only 5 principle components can capture 86% variance [30] of the data of bash program which have 184 segments and 924 unique system call sequences. The plot in the right side of the figure 5 represents both the PCA score and loading in the normal subspace S_d and in the abnormal subspace S_a. This plot indicates that segment "4" has a significantly different score than the other segments and has the longest distance from the normal subspace. Thus, we can separate this segment from the other data points. We then calculate the distances of the features and select the unique features of segment 4 and obtain 38 system call sequences. Furthermore, these features are less co-related with the other features and mostly co-related with each other.

3) Finalize Abnormal System Call Sequences: With the help of PCA, we isolate the anomalous system call sequences and after that, we prepare different sets of them. In order to prepare the sets, we identify the shortest unique system call sequences first. Then, sort out the supersets of a system call sequence and group them all in the same set. In the same set, the smallest system call sequence will be placed in the top. If a system call sequence does not have any supersets, we consider that sequence as a set.

For example, we have five system call sequences $<P_1P_2P_3$,

P_1P_8, P_1, P_4, P_4P_5>. These five sequences can be divided into two different sets 1. <P_1, P_1P_8, $P_1P_2P_3$> and 2. <P_4, P_4P_5> . The top item of a set is the most frequent subset of all system call sequences of that set.

B. Localizing Buggy Functions in Applications

Mapping an abnormal system call sequence to specific bug-related *application functions* is challenging, especially when the source code is not available. We propose to leverage off-line profiling to associate application functions with the abnormal system call sequences. Specifically, We obtain system call traces using dynamic binary instrumentation outside the production environment and match against the abnormal sequence.

One challenge is that an offline execution trace is unlikely to be exactly the same as the production trace due to environmental inconsistencies, the unavailable inputs, or the non-deterministic interleavings. To address this problem, SCMiner proposes an *offline function signature mapping* method that creates a signature for each function outside the production environment. The signature is obtained by a set of closed frequent system call sequences for each function across multiple executions. Therefore, we can map the abnormal sequence back to each function signature to determine the suspected buggy function.

The benefits of using a signature are that we do not require the exactly same inputs, environment, or workload to localize the buggy functions. Since the mapping table is obtained offline outside the production environment, it does not induce production runtime overhead.

1) Extracting Offline Function Signatures: Given an abnormal system call sequence, we obtain the processes contained in the sequence as the process under debugging (PuDs). We then use PIN [41], a dynamic binary instrumentation tool, to instrument PuDs and execute them against a set of randomly generated inputs multiple times. At the end of each execution, we obtain a *function execution list*, where each entry in the list contains system call numbers, resource ID, and the function name associated with them.

Once all executions are finished, SCMiner groups all entries for all function execution lists by the same function and the same process name together. Next, SCMiner extracts the function signature, which is the maximal frequent system call sequence, from each group. The signature can characterize the behavior of a function. For example, suppose there are three function execution lists for a function f in application A is: <open, write, close>, <stat, open, read>, and <lstat, open, write, close>. The function signature of f with respect to R (the minimum support) is <open, write, close> because it is the maximal frequent system call sequence [42].

2) Identifying and Ranking Buggy Functions.: Algorithm in Figure 6 shows the steps of localizing bug-related functions. SCMiner takes as input the offline function signatures (SIG_A) for PuDs and one item set of the abnormal system call sequences SC_{ab}. It outputs a list of top N ranked application functions F_{bug} that are likely to contain bugs. Each function in F_{bug} is associated with a ranking score.

Buggy Function Identification Algorithm

```
1:  Inputs: SC_ab, SIG_A
2:  Outputs: F_bug
3:  begin
4:    for each sc in ordered SC_ab
5:      for each sc_A in sc
6:        R_m ← SIG_A.match (sc_A)
7:        F_bug.add(sc_A, R_m)
8:      endfor
9:      F_bug ← F_bug.rank()
10:   endfor
11: return F_bug
```

Figure 6. Algorithm for locating the buggy sequence in the buggy function

Specifically, SCMiner iterates through SC_{ab}, beginning with the top-ranked system abnormal call sequence, and for each found sequence sc, SCMiner extracts the system calls sharing the same application name into an application-specific system call sequence sc_A. For example, in a sc = <write, read, write>, suppose the two writes are from the same function f_1 and the read is from function f_2, then sc_{f_1} = <write, write> and sc_{f_2} = <read>. Specifically, SCMiner treats each sc_A in application A from sc as a query and searches sc_A against all function signatures in A. The search problem is formulated as the the longest common sub-string matching problem. The matching score R_m is determined by the percentage of the matched system calls in each function signature. For example, suppose there are three function signatures F_1: <stat, read, write >, F_2: < unlink, rename, read >, and F_3: <read, write>. The abnormal system sequence sc_A is <read, write>. When matching sc_A against the three functions, the scores are 2/3, 1/3, and, 1. Therefore, F_3 is ranked at the top.

IV. Experiments

We developed SCMiner as a software tool based on several open-source platforms. Specifically, we used a Linux built-in audit daemon *auditd* [19] to collect system traces. Our abnormal system call sequence identification algorithm is implemented by SPMF [43], an open source data mining tool and Principle Component Analysis (PCA) library defined in R programming language [44]. The offline trace collection in fault localization is implemented by PIN [41].

In order to evaluate SCMiner, we consider three research questions:

RQ1: How effective is SCMiner at localizing abnormal system call sequences and buggy functions?

RQ2: How efficient is SCMiner at localizing abnormal system call sequences and buggy functions?

RQ3: What are the roles of PCA optimization and signature function matching in improving the effectiveness and the efficiency of SCMiner?

A. Benchmarks and Evaluation Metrics

All our benchmarks are real Linux applications with known concurrency failures due to incorrectly shared resources be-

Table II
BENCHMARK, DESCRIPTION AND RESOURCE INFORMATION OF THE FAILURES

Application	$NLOC$	NOF	*Bug ID*	*Bug Description*	NOP	NOR	$NOSR$	NOS
mv	7,002	77	Bugzilla-438076	another process terminates ("file is missing")	96	148	8	413,345
rm	5525	76	Bugzilla-1211300	rm terminates ("directory not empty")	234	165	6	762,104
mkdir	4,213	26	Debian-304556	file permission mode is modified	68	144	6	576,210
mknod	3,840	26	Debian-304556	file permission mode is modified	72	177	6	606,628
mkfifo	3,959	26	Debian-304556	file permission mode is modified	64	139	6	534,983
ln	3,890	81	Debian-357140	ln terminates ("file does not exist")	115	251	8	588,233
tail	4,492	104	Changelog	output not updated after attached process exits	193	242	12	680,640
chmod	3,983	57	GNU-11108	file permission mode is modified	56	79	6	277,520
pxz	370	5	Bugzilla-1182024	file permission mode is modified	79	134	6	327,402
cp1	4,010	70	Changelog	file permission mode is modified	148	269	8	685,872
cp2	4,132	70	Changelog	Directory creates fails("directory exists")	161	268	6	417,944
gzip	7,252	35	Debian-303927	file permission mode is modified	112	232	8	831,420
bzip2	9,263	136	Debian-303300	file permission mode is modified	84	110	8	457,884
bash	39,102	456	Debian-283702	corrupted history file	287	424	31	3,059,987
findutils	32538	271	Debian 67782	new database would be empty	292	342	26	916,842
lighttpd-1	37,919	883	Lighttpd-2217	http timeout	264	284	18	4,356,560
lighttpd-2	41,292	927	Lighttpd-2542	incorrect output	296	438	21	1,264,552
apache	195,005	5665	Apache -43696	server shutdown command is ignored	534	2529	29	3,629,040
locate	32,538	271	Debian 461585	File is missing	284	367	17	894,480

$NLOC$ = the number of non-comment lines of code. NOF = the number of functions. NOP = the number of processes. NOR = the number of unique system-wide resources accessed by the system calls in the log. $NOSR$ = the number of unique system-wide shared resources accessed by the system calls in the log. NOS = the number of system calls.

tween processes and/or signal handlers. These benchmarks are identified by searches from open-source repositories such as GNU, Bugzilla, and Debian. There are 19 program versions from 17 unique applications, among which 12 applications were from Linux Coreutils. To minimize bias, searches from these open-source repositories are conducted by a student who is not involved in the SCMiner project. These benchmarks have been used in other research [18], [45] for handling process-level concurrency bugs. The total number of benchmarks in this experiment is also comparable with prior work.

The student collected a system call trace for each benchmark by running multiple test cases multiple times and at least one execution can trigger the failure described in the bug report. The offline traces are collected by running a set of black-box (or functional) test cases to mimic the production runs against different input scenarios. The black-box test cases are often designed based on system parameters and knowledge of functionality [46]. The student followed this approach, using the category-partition method [47], which employs a Test Specification Language (TSL) to encode choices of parameters and environmental conditions that affect system operations and combine them into test cases. However, we did not know the root causes of these failures until we finished running and analyzing the results of SCMiner. Table II shows the statistics for each benchmark. The last column indicates the size of the system call traces.

B. Evaluation Metrics

Identifying abnormal system call sequences. To evaluate the effectiveness of abnormal system call sequence identification, we use the measurement of precision [48]. Precision represents the percentage of the ground truth (i.e., the actual abnormal) system call sequences from the system call sequence generated by our technique. To determine the ground truth, we manually examined the solution discussed in the corresponding issue report and the patch used for fixing the issue.

Localizing buggy functions. SCMiner reports top-N functions that are likely to be buggy and by default, N=20. In order to assess the effectiveness of localizing buggy functions, we measure two metrics. The first metric measures the rank number (position) of functions identified as bug-related. Again, the ground truths are determined by manually examining the solution discussed in the corresponding issue report and the patch used for fixing the issue.

For the second metric, we use Mean Average Precision (MAP). MAP is a single-figure measure of ranked retrieval results independent of the size of the top list [49]. It is designed for general ranked retrieval problems, where a query can have multiple relevant documents (e.g., an abnormal system call sequence may associate with more than one function), we compute the average ranking. To compute MAP, it first calculates the average precision (AP) for each individual query Q_i, and then calculates the mean of APs on the set of queries:

$$MAP = \frac{1}{|Q|} \cdot \sum_{Q_i \in Q} AP(Q_i)$$

To illustrate the MAP calculation, suppose there are bug-related functions f_1 and f_2 If Technique-I ranks the two options at the 1^{st} and 2^{nd} positions among all 500 functions and Technique-II ranks the two functions at the 1^{st} and 3^{rd} positions, then the MAP of Technique-I is (1/1 + 2/2) / 2 = 1 and the MAP of Technique-II is (1/1 + 2/3) / 2 = 0.8.

C. Results and Analysis

Table III summarizes the results of applying SCMiner to the benchmark programs. The results showed that 83% of system calls were removed after the filtering process. Column SC_{seq} shows the abnormal system call sequence, in the format of $SystemCall_{process}$. Column *Func* shows the function names associated with the abnormal system call sequence.

Table III
RESULTS OF APPLYING SCMINER OVER BENCHMARK APPLICATIONS

Prog	NOS_f	*#Seg.*	*#Ftr.*	*Syscall seq.*		*Func.Location*		*Root Cause*		*Time*
				Seq.	*prec.*	*Rank*	*MAP*	SC_{seq}	*Func*	*(sec)*
mv	19260	189	223	1	100	1,2	1	$unlink_{mv}$, $open_{cat}$, $rename_{mv}$	mv: copy_internal(), cat: main()	15.74
rm	23080	184	640	1	100	1,3,4	0.81	$fstatat_{rm}$, $symlink_{ln}$, $opentat_{rm}$	rm: fts_open(), fts_build(), ln: do_link()	29.083
mkdir	7920	135	35	1	100	1,2	1	$mkdir_{mkdir}$, $symlink_{ln}$, $chmod_{mkdir}$	mkdir: main(), ln: do_link()	19.42
mknod	7209	148	55	1	100	1,2	1	$mknod_{mknod}$, $symlink_{ln}$, $chmod_{mknod}$	mknod: main(), ln: do_link()	23.89
mkfifo	7200	148	55	1	100	1,2	1	$mknod_{mkfifo}$, $symlink_{ln}$,$chmod_{mkfifo}$	mkfifo: main(), ln: do_link()	21.857
ln	5376	187	36	1	100	1,2	1	$stat_{ln}$, $unlink_{rm}$ $symlink_{ln}$	ln: do_link(), rm: remove_entry()	16.69
tail	21600	200	120	2	100	1,3,4	0.8	$read_{tail}$, $rename_{mv}$, $fstat_{tail}$	tail: tail_forever_inotify(), dump_reminder(), mv: copy_internal()	22.302
chmod	4278	179	21	1	100	1,2,3	1	$stat_{chmod}$, $symlink_{ln}$, $fchmodat_{chmod}$	chmod: fts_open(), main(), ln: do_link()	9.86
pxz	30803	181	171	2	79.74	1,2	1	$umask_{pxz}$, $symlink_{ln}$, $chmod_{pxz}$	pxz: main(), ln: do_link()	31.112
cp1	9446	190	74	2	100	2,3	0.58	$mkdir_{cp}$, $fchmod_{chmod}$, $stat_{cp}$	cp1: copy_internal() chmod: main()	20.28
cp2	16728	190	105	2	75	2,3	0.58	$stat_{cp}$, $mkdir_{mkdir}$, $mkdir_{cp}$	cp2: copy_internal(), mkdir: main()	27.78
gzip	10560	190	153	2	100	1,2	1	$close_{gzip}$, $symlink_{ln}$, $chmod_{gzip}$	gzip: treat_file(), ln: do_link()	19.97
bzip2	16665	200	190	1	100	1,2	1	$close_{bzip2}$, $symlink_{ln}$, $chmod_{bzip2}$	bzip2: compressStream(), compress(), ln: do_link()	29.76
bash	236096	174	924	2	100	1	1	$open_{bash1}$, $write_{bash2}$, $write_{bash1}$	bash: history_do_write()	208.999
findutils	150967	180	843	1	100	1,2	1	$unlink_{mv}$, $opentat_{rm}$, $rename_{mv}$	mv: copy_internal(), rm: rm()	61.52
lighttpd-1	731663	398	3240	4	78.28	1,4	0.75	$exit_{cgi}$, $rt\_sigreturn_{light}$, $wait_{light}$	lighttpd: fdevent_event_del()	230.45
lighttpd-2	293367	292	1711	2	66.67	1,2,4	0.92	$close_{light}$, $wait_{cgi}$, $wait_{light}$	lighttpd: fdevent_unregister(), plugins_call_handle_subrequest	178.32
apache	1661990	320	56953	3	100	2,3	0.58	$rt\_sigpromask_{httpd}$, $rt\_sigaction_{bash}$	apache: ap_mpm_run(), bash: set_signal_handler()	273.62
locate	144824	186	427	1	100	1,2	1	$unlink_{mv}$, $fchmodat_{chmod}$, $rename_{mv}$	mv: copy_internal(), chmod: fts_open()	53.63

NOS_f = the number of system calls after filtering. *#Seg.* = the number of segments. *#Ftr.* = the number of features (system call sequences). #Seq = the number of abnormal sequence sets. *Rank* = the ranking position of the ground truth. *MAP* = the MAP score. SC_{seq} = the abnormal system call sequence. *Func* = the buggy functions. *Time* = the time spent on the analysis.

1) RQ1: Effectiveness of SCMiner: SCMiner is successful in *finding abnormal system call sequences* and bug-related functions in all 19 programs. The number of abnormal system call sequences computed by SCMiner ranged from 1 to 4. The size of each system call sequence ranged from 2 to 4 across all applications. Given the total number of system calls in the trace (Column "NOS" in Table II), the results indicate that developers only need to examine from 0.01% to 0.04% system calls among all system calls in the trace, with an average of 0.02%. The results also show that the identification of the buggy system call sequence is 66.67% to 100% precise (Column "prec." in Table III), with an average of 93% for all benchmark applications. In addition, SCMiner successfully *localized buggy functions* in all 19 applications. The average ranking position is 1.3 overall applications. The "rank" column contains multiple ranks because we have multiple ground-truth functions. The MAP score ranged from 0.58 to 1, with an average of 0.79. The MAP score indicates that all buggy functions identified SCMiner are ranked at the top-5. Given the total number of functions in a program (Column "NOF" in Table II), developers are required to examine at most 0.04% to 20% of all functions across all applications, with an average of 5%.

We conclude that *SCMiner is effective at detecting abnormal system call sequences and localizing buggy functions with respect to system-level concurrency failures in production.*

2) RQ2: Efficiency of SCMiner: The last column of Table III reports the end-to-end total run time of SCMiner, including filtering, PCA analysis, optimization, and buggy function localization. The overhead of collecting system call traces by the *auditd* daemon is almost negligible, ranging from zero to 2X, with an average of 0.31X overall applications.

For the binary instrumentation used to localize buggy functions, the overhead ranged from 1.3X to 36X, with an average of 8.5X. These overheads are in the similar order of magnitude as that of other profilers [50], [51]. We consider these overheads to be acceptable for out-of-production usage, which is the intended usage of collecting function signatures.

The above results indicate that *SCMiner is efficient and practical for being used for localizing system-level concurrency faults.*

3) RQ3: The Role of Optimization and Function Signature: To evaluate the role of the optimization techniques used in finding abnormal system calls (i.e., removing irrelevant system calls, sequence abstraction), we computed the total time of SCMiner without optimization, denoted by SCMiner$_{nop}$. Figure 7 shows the time spent by SCMiner and SCMiner$_{nop}$, respectively. Compared to SCMiner$_{nop}$, SCMiner is 1.5 times faster on average in terms of the end-to-end analysis time across all applications. The speedup is more significant in larger applications (e.g., `bash`, `apache`). This is primarily because the optimization reduced the size of feature vectors used for PCA, reduced the overall number of system call sequences, and thus also reduced the time of searching frequent system call sequences in the source code. Overall, these results indicate that the *use of optimization techniques contributed to enhancing the efficiency of SCMiner.*

To evaluate whether the use of function signatures can improve the effectiveness of identifying buggy functions, we use a baseline version SCMiner$_{nfs}$ to compare with SCMiner. SCMiner$_{nfs}$ does not compute function signatures. Instead, it collects a single trace outside the production environment and then uses a simple exact string matching approach [52] to determine if an abnormal system call belongs to certain functions. For example, a system call from the buggy sequence is considered as a query and will be searched in the system call sequence of the single execution trace. Figure 8 shows the MAP scores of both SCMiner and SCMiner$_{nfs}$ across the 19

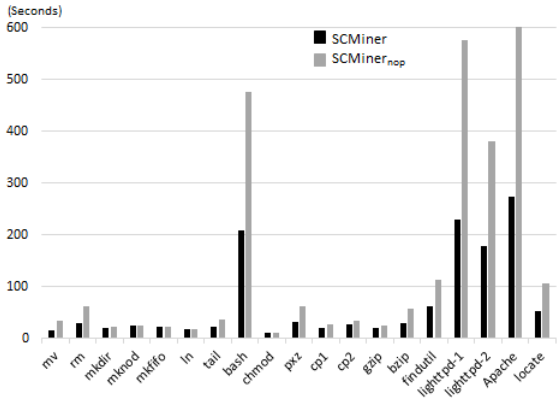


Figure 7. Comparing the time taken by SCMiner and SCMiner$_{nop}$

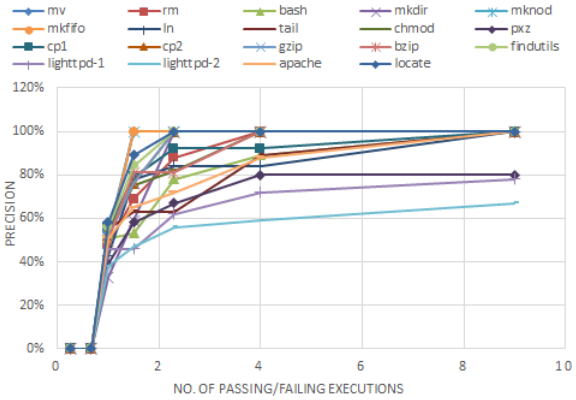


Figure 9. Precision changes with the content of traces.

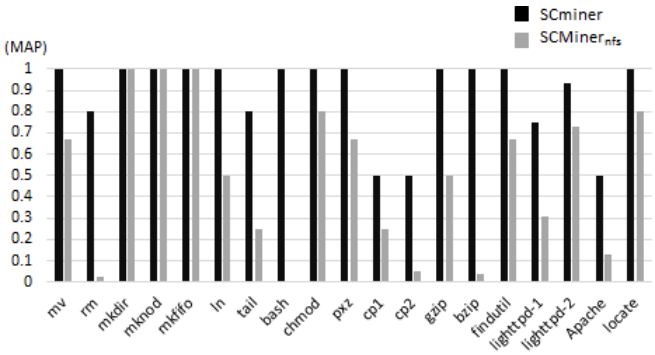


Figure 8. Comparing the effectiveness of SCMiner and SCMiner$_{nfs}$

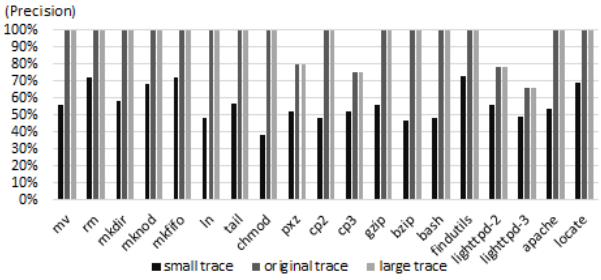


Figure 10. Precision changes with the size of traces.

applications. The results show that the use of function signature increased the effectiveness of SCMiner in 16 applications, ranging from 0.3% to 100%, with an average of 44%. On the `bash` program, we observed a 100% improvement because the buggy function was not ranked in the top 20 functions by SCMiner$_{nfs}$.

The above results indicate that *the function signature technique is more effective in localizing buggy functions than a simple string matching approach.*

V. Limitations and Dicussion

A. Limitations

SCMiner assumes each logged system call contains sufficient information on resources being accessed. SCMiner may not process logs, in which resource information is not available in each system call. Second, function signatures are collected from the execution traces. Therefore, the accuracy of the signatures largely depend on the quality of inputs. Existing automated test case generation techniques [47] can be leveraged to cover as many functions as possible for improving the quality of traces.

B. Discussion

Quality of traces. We investigated how the quality of system call traces influence the effectiveness of SCMiner. We varied the percentage of the passing and failing executions in the log under analysis. As shown in Figure 9, the x-axis indicates the ratio of the percentage of passing and failing executions and the y-axis indicates the precision scores of SCMiner in detecting abnormal system call sequences. For example, the ratio score 9 means, there are 90% passing executions and 10% failing executions. The precision score is 0% when the buggy sequence cannot be captured by SCMiner and it happens when the failing executions occupy a large percentage than the passing executions. These results show that there is a trend when the ratio between the passing and failing executions increases, the precision increases. The precision score reaches its peak for all applications when the percentage of passing executions is about 90%.

The above results indicate that SCMiner is most useful when the number of normal system call sequences is a dominant majority in the trace and they appear frequently. This is due to the PCA algorithm used in the approach.

Scalability. We further examined the effectiveness and efficiency of SCMiner when handling system call traces with different sizes. In addition to the original traces, we consider two variations of the original traces generated from the 19 applications: 1) small-size trace and 2) large-size trace. To create small-size traces, we removed 50% of executions from each original trace. To create large-size traces, we added an additional 50% of executions to each original trace.

Figure 10 plots the precision scores of SCMiner. The results indicate that precision varied on all applications when changing the size of the trace from "small" to "original". On all applications, the precision scores generally remain the same when the trace size is increased from "original" to

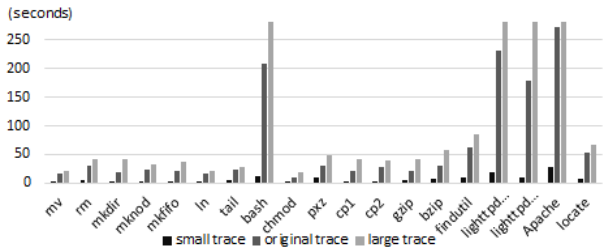


Figure 11. Time changes with the size of traces.

"large". Figure 11 plots the efficiency. The results indicate that compared to the original traces, the time spent on analyzing small traces was 88% (7.30 seconds on average) less and that on analyzing large traces was 53% (104.24 seconds on average) more.

The above results imply that SCMiner is able to handle large-size traces with little extra cost.

VI. Related Work

Fault localization for concurrent programs. There has been a lot of work on fault localization for concurrent programs [6], [16], [53]. For example, Park et al. [6] monitor memory-access patterns associated with a program's pass/fail results. Wang et al. [15] identify shared memory access pairs that behave distinctively in failed and successful runs, and pinpoint root causes using different test procedures. This technique targets order violations and does not rank concurrency violation patterns. CCI [13] ranks only shared variable accesses (predicates) and thus provides less contextual information. However, these techniques assume multiple failing and passing executions, which are often hard to obtain in practice. In addition, they require instrumenting memory accesses and thus are intended to work outside of the production environment. In contrast, SCMiner is a production-level fault localization tool that uses system-generated system call traces with little overhead. In addition, SCMiner assumes that failing executions happen more rarely than normal executions, which is a practical assumption in the production environment.

Process-level concurrency failures. RacePro [11] leverages the vector-clocks algorithm to detect a process-level race if it happens during test execution. It tracks the accesses of shared kernel resources via system calls and records executions of multiple processes. SimRacer [18] and RacePro [11] aim to detect process-level concurrency faults by testing for different interleavings of system calls. Descry [17] can reproduce system-level concurrency failures by combining static and dynamic analysis techniques to generate test inputs. [11] However, all of these techniques have different goals from SCMiner; none of them focus on detecting abnormal system calls from traces or localizing buggy functions.

Anomaly detection from runtime logs. There has been some research on detecting anomalies [33], [54], [55] from logs. For example, Xu et al. [33], [54] mine console logs and identify the abnormal log message patterns. Liu et al. [55] and Du et al. [56] analyze the characteristics of system logs to identify the abnormal behaviors of a system that are caused by attacks. Lakhina et al. [32] use PCA anomaly detection algorithm to diagnose network-wide traffic anomalies. This method uses Principal Component Analysis to identify an anomalous subspace of the network traffic which is noisier and contains significant traffic spikes. In contrast, SCMiner focuses on finding system call sequences for diagnosing system-level concurrency faults. Moreover, SCMiner can pinpoint the root causes of failures associated with abnormal system calls.

VII. Conclusions

We have presented SCMiner, the first automated tool to diagnose system-level concurrency failures in multi-process applications. SCMiner can detect abnormal system call sequences from the traces generated by the default system auditd daemon by using a combination of dynamic analysis, data mining, and statistical analysis techniques. SCMiner can also localize buggy application functions associated with the abnormal system call traces. We have evaluated SCMiner on 19 real-world multi-process applications. The results showed that SCMiner is both effective and efficient in diagnosing system-level concurrency failures.

References

[1] R. Capuano, "Interactive visualization of concurrents programs," in *19th IEEE International Conference on Automated Software Engineering (ASE)*, 2004, pp. 418–421.

[2] G. Altekar and I. Stoica, "Odr: Output-deterministic replay for multicore debugging," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 193–206.

[3] H. Cleve and A. Zeller, "Locating causes of program failures," in *International Conference on Software Engineering*, 2005, pp. 342–351.

[4] J. Clause and A. Orso, "A technique for enabling and supporting debugging of field failures," in *International Conference on Software Engineering*, 2007, pp. 261–270.

[5] A. V. Thakur, R. Sen, B. Liblit, and S. Lu, "Cooperative crug isolation," in *Proceedings of the International Workshop on Dynamic Analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), (WODA)*, 2009, pp. 35–41.

[6] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE*, 2010, pp. 245–254.

[7] J.-D. Choi and A. Zeller, "Isolating failure-inducing thread schedules," in *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '02, 2002, pp. 210–220.

[8] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering, (ICSE)*, 2005, pp. 342–351.

[9] N. Jalbert and K. Sen, "A trace simplification technique for effective debugging of concurrent programs," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2010, pp. 57–66.

[10] J. Huang, P. Liu, and C. Zhang, "LEAP: lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2010, pp. 207–216.

[11] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive detection of process races in deployed systems," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, 2011, pp. 353–367.

[12] C. Zamfir and G. Candea, "Execution synthesis: A technique for automated software debugging," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, 2010, pp. 321–334.

[13] A. Thakur, R. Sen, B. Liblit, and S. Lu, "Cooperative crug isolation," in *Proceedings of the Seventh International Workshop on Dynamic Analysis (WODA)*, 2009, pp. 35–41.

[14] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *International Conference on Software Engineering*, 2010, pp. 245–254.

[15] W. Wang, C. Wu, P. Yew, X. Yuan, Z. Wang, J. Li, and X. Feng, "Concurrency bug localization using shared memory access pairs," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, (PPoPP)*, 2014, pp. 375–376.

[16] S. Park, R. W. Vuduc, and M. J. Harrold, "A unified approach for localizing non-deadlock concurrency bugs," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST*, 2012, pp. 51–60.

[17] T. Yu, T. S. Zaman, and C. Wang, "DESCRY: reproducing system-level concurrency failures," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, 2017, pp. 694–704.

[18] T. Yu, W. Srisa-an, and G. Rothermel, "Simracer: An automated framework to support testing for process-level races," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013, 2013, pp. 167–177.

[19] auditd(8) - linux man page. [Online]. Available: https://linux.die.net/man/8/auditd

[20] (2008) Principal component analysis (pca) procedure. [Online]. Available: https://onlinecourses.science.psu.edu/stat505/node/51/

[21] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP)*, 2003, pp. 237–252.

[22] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006, pp. 308–319.

[23] Scminer. [Online]. Available: https://github.com/Tarannum-Zaman/Scminer

[24] F. Valsorda. Searchable linux syscall table for x86 and x86_64. [Online]. Available: https://filippo.io/linux-syscall-table/

[25] O. Laadan, N. Viennot, C.-C. Tsai, C. Blinn, J. Yang, and J. Nieh, "Pervasive detection of process races in deployed systems," in *ACM symposium on Operating Systems Principles*, 2011, pp. 353–367.

[26] (2004) Debian bug report logs - #283702. [Online]. Available: https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=283702

[27] Bash guide for beginners. [Online]. Available: https://www.tldp.org/LDP/Bash-Beginners-Guide/html/$sect$_01_01.html

[28] H. Abdi and L. J. Williams, "Principal component analysis," *Wiley interdisciplinary reviews: computational statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[29] J. Shlens, "A tutorial on principal component analysis," in *Systems Neurobiology Laboratory, Salk Institute for Biological Studies*, 2005.

[30] S. H. To. (2019) Variance: Simple definition, step by step examples. [Online]. Available: https://www.statisticshowto.datasciencecentral.com/probability-and-statistics/variance/

[31] L. Eriksson. (2018) What is principal component analysis (pca) and how it is used? [Online]. Available: https://blog.umetrics.com/what-is-principal-component-analysis-pca-and-how-it-is-used

[32] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2004, pp. 219–230.

[33] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, "Detecting large-scale system problems by mining console logs," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP)*, 2009, pp. 117–132.

[34] I. Jolliff, *Principal Component Analysis*, 2nd ed. Springer, 2002.

[35] H. Lohninger. (2012) Pca - loadings and scores. [Online]. Available: http://www.statistics4u.com/$fundstat\_eng/cc\_pca\_loadscore$.html

[36] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 329–339.

[37] "Apache Deadlock," 2003, http://marc.info/?l=apache-httpd-bugs&m=105967988713871.

[38] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, 1994, pp. 487–499.

[39] P.-N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining (2Nd Edition)*, 2nd ed. Pearson, 2018.

[40] S. M. Holland. (2008) Principal components analysis (pca). [Online]. Available: https://strata.uga.edu/software/pdf/pcaTutorial.pdf

[41] (2012) Pin - a dynamic binary instrumentation tool. [Online]. Available: https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[42] B. Ziani and Y. Ouinten, "Mining maximal frequent itemsets: A java implementation of fpmax algorithm," in *Proceedings of the 6th International Conference on Innovations in Information Technology (IIT)*, 2009, pp. 11–15.

[43] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, and V. S. Tseng, "Spmf: A java open-source pattern mining library," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3389–3393, Jan. 2014.

[44] (2014) Principal component analysis based unsupervised anomaly detection. [Online]. Available: http://www.bistaumanga.com.np/blog/pccAnomalyProbe/

[45] 2016, http://cs.uky.edu/ tyu/research/descry.

[46] A. Causevic, D. Sundmark, and S. Punnekkat, "An industrial survey on contemporary aspects of software testing," in *IEEE International Conference on Software Testing, Verification and Validation*, 2010, pp. 393–401.

[47] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Commun. ACM*, pp. 676–686, 1988.

[48] W. Koehrsen, "Beyond accuracy: Precision and recall," https://towardsdatascience.com/beyond-accuracy-precision-and-recall-3da06bea9f6c, March 3 2018.

[49] P. R. . H. S. Christopher D. Manning. (2008) Evaluation of ranked retrieval results. [Online]. Available: https://nlp.stanford.edu/IR-book/html/htmledition/evaluation-of-ranked-retrieval-results-1.html

[50] L. Gong, M. Pradel, and K. Sen, "JITProf: Pinpointing JIT-unfriendly JavaScript code," in *ACM SIGSOFT Symposium on Foundations of Software Engineering*, 2015, pp. 357–368.

[51] A. Nistor, L. Song, D. Marinov, and S. Lu, "Toddler: Detecting performance problems via similar memory-access patterns," in *International Conference on Software Engineering*, 2013, pp. 562–571.

[52] A. Reyes. (2015) Exact string matching algorithms. [Online]. Available: https://www.hackerearth.com/practice/notes/exact-string-matching-algorithms/

[53] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, and M. Hsu, "Mining sequential patterns by pattern-growth: The prefixspan approach," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 11, pp. 1424–1440, 2004.

[54] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan, "Online system problem detection by mining patterns of console logs," in *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM)*, 2009, pp. 588–597.

[55] Z. Liu, T. Qin, X. Guan, H. Jiang, and C. Wang, "An integrated method for anomaly detection from massive system logs," *IEEE Access*, vol. 6, pp. 30 602–30 611, 2018.

[56] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1285–1298.