Automatically Detecting Missing Cleanup for Ungraceful Exits

Zhouyang Jia*
Shanshan Li
College of Computer Science,
National University of Defense
Technology, China
{jiazhouyang,shanshanli}@nudt.edu.cn

Tingting Yu
Department of Computer Science,
University of Kentucky, USA
tyu@cs.uky.edu

Xiangke Liao
Ji Wang
College of Computer Science,
National University of Defense
Technology, China
{xkliao,wj}@nudt.edu.cn

ABSTRACT

Software encounters ungraceful exits due to either bugs in the interrupt/signal handler code or the intention of developers to debug the software. Users may suffer from "weird" problems caused by leftovers of the ungraceful exits. A common practice to fix these problems is rebooting, which wipes away the stale state of the software. This solution, however, is heavyweight and often leads to poor user experience because it requires restarting other normal processes. In this paper, we design *SafeExit*, a tool that can automatically detect and pinpoint the root causes of the problems caused by ungraceful exits, which can help users fix the problems using lightweight solutions. Specifically, *SafeExit* checks the program exit behaviors in the case of an interrupted execution against its expected exit behaviors to detect the missing cleanup behaviors required for avoiding the ungraceful exit. The expected behaviors are obtained by monitoring the program exit under a normal execution. We apply *SafeExit* to 38 programs across 10 domains. *SafeExit* finds 133 types of cleanup behaviors from 36 programs and detects 2861 missing behaviors from 292 interrupted executions. To predict missing behaviors for unseen input scenarios, *SafeExit* trains prediction models using a set of sampled input scenarios. The results show that *SafeExit* is accurate with an average F-measure of 92.5%.

CCS CONCEPTS

• **Software and its engineering** → **Software reliability**; *Software testing and debugging*; Dynamic analysis.

KEYWORDS

Ungraceful exit, Software signal, Missing cleanup

ACM Reference Format:
Zhouyang Jia, Shanshan Li, Tingting Yu, Xiangke Liao, and Ji Wang. 2019. Automatically Detecting Missing Cleanup for Ungraceful Exits. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338938

*Also with Department of Computer Science, University of Kentucky, USA.

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338938

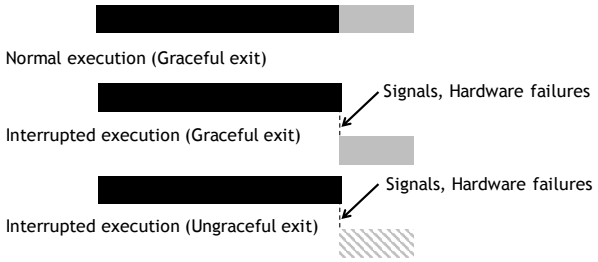

Figure 1: Different types of executions and exits.

1 INTRODUCTION

An *interrupted execution* is defined as a program execution interrupted by a software signal (e.g., a *sigterm* signal issued by a user or a *sigsegv* signal triggered by a program bug) or a hardware failure (e.g., power off). This is different from a *normal execution*, where the program terminates naturally without experiencing an internal or external abnormal event. A *graceful exit* is the ability to terminate the program at the end of interrupted execution, leaving the system in a consistent state. To achieve this, programs usually perform some cleanup behaviors at the exit stage, such as terminating children processes, releasing resources, and writing state files. Graceful exit is advocated in many operating systems, because if it is not correctly done, the consequence would be data corruption of program and operating system files, which can negatively impact the stability or the correctness of the system. In contrast, an *ungraceful exit* represents a program exit under an interrupted execution and that the program does not perform any or part of the cleanup behaviors. Figure 1 illustrates the types of executions and exits.

An ungraceful exit can be caused by 1) software bugs in signal handlers, in which developers do not correctly clean up the system state; 2) developers intentionally retain certain system resources (e.g., processes) after program terminations in order to attach a debugger or collect important information, such as a core dump or stack trace, for diagnosing the software failure [43]; 3) hardware failure, which means the program has no chance to clean up. In any case, users may suffer from "weird" problems due to the ungraceful exits. Taking *Nginx* [28], a server program, as an example, if a fatal error (e.g., dereference of a null pointer) happens in its main process, *Nginx* will exit ungracefully because the children processes are not cleaned up and thus become orphan processes. As a result, users may encounter the following problems: 1) *Nginx* and any other web server using the same networking port fail to start because the port is occupied by an orphan process; 2) *Nginx* fails to stop because the main process does not exist anymore. We refer to the problems

caused by ungraceful exits (UE) as UE problems. To fix UE problems, the most common advice from technical supporters is rebooting the computer [27, 35, 36]. This is because a reboot wipes away the stale state of the software [20, 21]. However, sometimes simply rebooting the computer may still not fix the problems, such as when files are left inconsistent in the computer's hard drive during an ungraceful exit [17]. In these cases, a "hard reset" is required — users have to reset the computer to its factory default state or even reinstall the operating system.

The above solutions (e.g., reboot, reset) to UE problems are often heavyweight and thus lead to poor user experience, because other normal processes have to restart at the same time, especially for the server programs, such as web servers or database servers [32]. The main reason for using the heavyweight solutions is that users do not understand the cause of the problems and thus not able to fix them specifically. For example, to fix the UE problems of *Nginx*, instead of rebooting the operating system, users can manually kill the orphan processes if they know where the root cause is. Therefore, techniques on identifying the causes of UE problems are needed.

There has been much research on addressing problems involving software interruptions or hardware failures. For examples, many techniques are targeted at studying and detecting exception handling bugs [3, 5, 8, 10, 14, 18, 29–31, 41, 42], such as checking if an exception handler is correctly implemented with respect to the specification. However, these techniques cannot detect UE problems that may or may not be handled by exception handlers. There have been some works focusing on preventing and detecting persistency bugs in storage systems, such as file systems and database systems [1, 6, 7, 11, 16, 38, 45, 46]. For example, if a storage system crashes during the process of writing data, the system has to carefully roll back the operation to avoid data corruption. However, rollback aims to recover the data in storage system but does not target at cleaning up the system state in the case of ungraceful exit. Other works have studied the crash recovery mechanism [12, 13, 15, 39], especially in distributed systems, such as recovering failed nodes. These techniques target at addressing problems during system recovery, but not the exit of programs.

In this paper, we design and implement *SafeExit*, a tool that can automatically detect and pinpoint the root causes of UE problems, which can help users to fix the problems with lightweight solutions without having to reboot the system. *SafeExit* focuses on detecting and localizing UE problems instead of recovering the system or rollback. The key insight of *SafeExit* is to check the program exit behaviors in the case of an interrupted execution against *expected* exit behaviors. A *behavior* consists of a sequence of system calls because a program interacts with the operating system through system calls, which can change the state of the system and influence other programs. We assume that the expected exit behaviors can be obtained by monitoring the behaviors of the program exit in a *normal execution* (i.e., an execution without interrupts). The intuitions are that 1) developers often carefully test the program under normal execution, and 2) it does not require reserving system resources for debugging under normal execution. Thus, we assume all necessary cleanup operations are performed when a program exits normally. For example, when *Nginx* exits normally, the main process will notify its children processes, who will stop listening to the 80 port and exit, then the main process deletes the *nginx.pid* file and exits. Finally, any inconsistencies between the actual and expected behaviors are reported as the causes of UE problems.

SafeExit involves two phases: *offline learning* and *online monitoring*. Given a program, the offline learning phase first generates a diverse set of input scenarios (i.e., combinations of configuration options and workload values). For each input scenario, *SafeExit* obtains one normal execution and multiple interrupted executions by simulating different interruption triggers (e.g., a particular software signal). From each normal and interrupted execution, *SafeExit* traces its exit state and extracts the cleanup behaviors from the trace. After that, *SafeExit* selects the expected behaviors for each interrupted execution from the behaviors of the normal execution. It then compares the actual and expected behaviors of the interrupted execution to determine if any missing behaviors are detected. The output of the comparison is a *behavior vector* in a binary format, where each element in the vector indicates a normal behavior. The behavior vector is initialized with all zero values and whenever a missing behavior for a particular interrupted execution is detected, the corresponding behavior is set to 1. Therefore, *SafeExit* learns a behavior vector for each interrupted execution. Finally, *SafeExit* builds a prediction model, which can predict the missing cleanup behaviors for an unseen input scenario and an interruption trigger.

In the online monitoring phase, *SafeExit* is deployed in the production environment, such that given a real input scenario. *SafeExit* monitors the state of the target program, and once an interruption is detected, it predicts if an ungraceful exit occurs and reports the missing cleanup behaviors to users. Based on the reported behaviors, users can manually clean up the system state (e.g., delete a file, change an option, or kill a process) without rebooting the system.

To evaluate *SafeExit*, we conduct experiments on 38 widely used programs from 10 software domains. All programs are open source, mature, active and written in C/C++. The results showed that *SafeExit* identifies 133 types of cleanup behaviors from the normal executions of 36 programs under default configurations, while 2 programs do not perform any cleanup behavior when exiting normally. Next, *SafeExit* simulates 12 interruption triggers for the 36 programs (i.e., 432 interrupted executions), and detects 2861 missing behaviors from 292 interrupted executions. The results also showed that *SafeExit* is effective and efficient in training prediction models for detecting missing cleanup behaviors in ungraceful exits given unseen input scenarios. On average, it samples only 93.2 input scenarios in the learning phase and achieves 92.5% accuracy in terms of F-measure.

In summary, the contributions of this paper are as follows:

- We conduct the first research to define and study ungraceful exit (UE) problems. The study builds a taxonomy of cleanup behaviors, which helps people better understand UE problems and design tools to handle them.
- We design and develop *SafeExit*, an automated tool to detect ungraceful exits and pinpoint the root causes of the UE problems. *SafeExit* provides lightweight solutions for users to fix UE problems.
- We evaluate *SafeExit* on 38 widely-used programs from 10 software domains. The results show that *SafeExit* is effective and efficient in detecting and predicting missing behaviors in ungraceful exits.

```
$ sudo nginx # try to start nginx
nginx: [emerg] bind() to *ip*:80 failed (98: Address already in use)

$ sudo nginx -s quit # try to stop nginx
nginx: [alert] kill(*pid*, 3) failed (3: No such process)
```

Figure 2: Users could neither start nor stop Nginx, after a fatal signal is issued to the main process.

2 UNGRACEFUL EXIT PROBLEM

In this section, we first present two illustrative examples, then introduce preliminaries and definitions of UE problems.

2.1 Illustrative Examples

The first example is *Nginx* [28], a popular open-source web server. When a fatal-bug signal (e.g. *sigill*, *sigbus*, *sigfpe*, *sigsegv*) is issued to the main process, it causes an ungraceful exit. Specifically, a few orphan processes are left when the program terminates. As a consequence, users could neither start nor stop *Nginx*. As shown in Figure 2, *Nginx* fails to start since the port is used by the orphan processes and fails to stop since the main process does not exist anymore. At the same time, any other web server using the same port will fail to start too. To fix these problems, users need to restart the operating system, or kill the orphan processes manually.

Another example is an industrial proxy program. The program turns on a system configuration option, *i.e.*, SOCKS Proxy, at the startup stage. In a normal execution, the program will turn off this option at the exit stage. But when the program is killed by users through the system monitor (this often happens when users suffer from performance issues), the option will still be "on", and the operating system will be in a corrupted state. As a result, users cannot access the Internet anymore. To fix this problem, users can either set the configuration option back to "off", or reset the computer to its factory default state. The problem is with the files on hard drive, thus can not be fixed by restarting.

For problems in both examples, users can solve them by restarting the computer or resetting the computer to its factory default state. At the same time, however, the problems can also be solved by simply killing some processes or changing a configuration option. The latter solutions are obviously more lightweight for users compared with the former ones. Motivated by these examples, we design an automated tool that pinpoints the root causes of the problems caused by ungraceful exits, and helps users to fix the problems with lightweight solutions.

***SafeExit*'s solution.** In the case of *Nginx*, *SafeExit* first generates a set of input scenarios for *Nginx*. Each scenario contains a combination of configuration options (e.g., *worker_processes*) and workload values (e.g., http request). For each input scenario, *SafeExit* uses the command "*nginx -s quit*" to get the normal execution, and uses software signals (e.g., *sigsegv*, *sigterm*, *sigkill*) to trigger the interrupted executions. *SafeExit* then extracts exit behaviors from the normal execution (e.g., deleting pid file and terminating children processes) as the expected behaviors. For each interrupted execution, *SafeExit* selects the expected behaviors from the normal behaviors according to the interruption trigger (i.e., software signal). After that, *SafeExit* extracts exit behaviors from the interrupted execution, and compares them against the expected ones. For each input scenario and each interruption trigger, *SafeExit* outputs a vector of missing behaviors. Finally, *SafeExit* builds a prediction model, in which the features are the input scenarios and the interruption triggers and the labels are the missing behavior vectors.

Table 1: Category of interruption triggers.

Category	Interruption triggers
Fatal bug	SIGILL, SIGABRT, SIGBUS, SIGFPE, SIGSEGV, SIGPIPE, SIGXCPU, SIGXFSZ
User termination	SIGHUP, SIGINT, SIGQUIT, SIGTERM
Non-term. signal	SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOT, SIGCHLD
Hard exit	SIGKILL, kernel panic, hard restart, hardware failure, power failure
Other signal	SIGTRAP, SIGUSR1, SIGUSR2, SIGALRM, SIGVTALRM, SIGPROF, SIGPOLL

Challenges. There are three main challenges in the design of *SafeExit*. First, it is non-trivial to extract the cleanup behaviors from a system-call trace, since one behavior may consist of multiple system calls, and there can be a number of noisy system calls. To address this challenge, we use a combination of static and dynamic approach. Specifically, *SafeExit* first extracts dynamic system-call sequences according to the resources used by the calls then clusters sequences according to static source-code semantics.

The second challenge is that program behaviors are often affected by environments, such as configurations and workloads. *SafeExit* needs to predict the missing behaviors in production environments. In this regard, we apply machine learning techniques to predict the missing behaviors for an unseen input scenario.

Third, the expected behaviors of an interrupted execution may not be exactly the same as the ones in a normal execution. *SafeExit* has to select the necessary behaviors in interrupted executions. To achieve this, we study real-world cleanup behaviors from 38 programs and build a taxonomy (Section 3), which helps to design heuristic rules for selecting expected behaviors.

2.2 Triggers of Program Interruptions

An interruption can be triggered by various reasons, which often require different cleanup behaviors at the exit stage. These triggers can be summarized into five categories: 1) Fatal bug, e.g., dereference of a null pointer. The operating system notifies programs of the bugs in the form of signals; 2) User termination, e.g., an user input *ctrl+c* will lead to a *sigint* signal; 3) Non-termination signal. The signals that do not terminate the programs, e.g., *sigstop*; 4) Hard exit, which means the exit situation can not be handled by programs, e.g., hard restart or *sigkill* signal; 5) Other signal, signals for specific purposes, e.g., *sigtrap* is often used for debug, while *sigusr1* is used for program-specific purpose. Table 1 shows different types of triggers. *SafeExit* focuses on the types of *user termination*, *fatal bug* and *hard exit*. The other types of triggers do not require cleanup because the *other signals* are issued for specific purposes, while the *non-termination signals* will not terminate the programs.

For interruptions caused by user terminations and hard exits, a program can use the exit behaviors of its normal execution to clean up the stale states. Suppose the stale state set at the time of interruption is S, and the exit behavior set of the normal execution is

Table 2: Taxonomy of cleanup behaviors.

ID	Behavior type	Abbreviation	Impact of missing behavior	Fix advise
1	Delete PID file†	D-PID	The users or other programs believe the program is still running	Delete the *.pid* file
2	Delete sock/fifo files	D-SOCK	The next startup may fail since failing to create the file	Delete the files
3	Delete temp file†	D-TMP	The silent consuming of disk resource	Delete the temp file
4	Delete shared memory†	D-SHM	The silent consuming of memory resource	Delete the shared memory
5	Delete lock file†	D-LCK	The other programs may fail since failing to obtain the lock	Delete the lock file
6	Unlock lock file	U-LCK	The next startup may fail since failing to obtain the lock	Unlock the lock file
7	Kill child process†	K-CHLD	The child process does not properly exit and consumes resources	Kill the child process
8	Write program config file	W-PCFG	The program config is in corruption state	Change the program config
9	Write system config file†	W-SCFG	The system config is in corruption state	Change the system config
10	Write program state file	W-PSTAT	The next startup may encounter feature failures	Delete the state file
11	Write system state file†	W-SSTAT	The other programs may encounter feature failures	Warning
12	Write log file†	W-LOG	The log file is incomplete, affecting users or log analyzing tools	Warning
13	Sendmsg to child process	S-CHLD	The target process may fail	Warning
14	Sendmsg to other program†	S-PROG	The target program may fail	Warning
15	Sendmsg to network socket†	S-NET	The target socket may fail	Warning

† Cleanup behaviors of this type can potentially affect other programs or the operating system.

B. If we assume that a normal exit command is issued at the time of interruption, the behavior set *B* is enough to clean up the state set *S*. Therefore, we regard the exit behaviors of the normal execution as expected behaviors of user-termination and hard-exit interruptions. For user-termination interruptions, we also need to extract their actual behaviors, which are used to compare against the expected behaviors. For hard-exit interruptions, the expected behaviors are regarded as missing behaviors directly, since the program can not handle these interruptions.

For interruptions caused by fatal bugs, we can not regard the exit behaviors of the normal execution as expected behaviors, since the program is in an error and unknown state at the time of interruption. For example, a database program may sync its data in the exit stage of a normal execution. When fatal bugs happen, the data may be corrupted and should not be synced. In this regard, we need to select necessary behaviors from the exit behaviors of the normal execution as expected behaviors of fatal-bug interruptions.

2.3 Definitions

For the ease of presenting the design of *SafeExit*, we introduce some key concepts that will be used throughout the rest of the paper.

Normal/interrupted execution. A *normal execution* (*NE*) is a program execution ended with a normal exit, such as clicking an exit button of a graphical user interface (GUI) program, or issuing an exit command of a command line interface (CLI) program. In contrast, an *interrupted execution* (*InE*) is ended by one of the trigger listed in Table 1. An *InE* can be either graceful or ungraceful.

Normal/interrupted trace. A *trace* is the sequence of system calls of a program execution. A *normal trace* (*NT*) is from a *NE*, while an *interrupted trace* (*InT*) is from an *InE*.

Normal/interrupted behavior. A *behavior* is a sequence of syscall calls that are together to perform a high-level task. A *normal behavior* (*NB*) is a sub-sequence of a *NT*, while a *interrupted behavior* (*InB*) is a sub-sequence of an *InT*.

3 EXPECTED CLEANUP BEHAVIORS

To detect ungraceful exits and localize their root causes, we need to obtain the *expected behaviors* at the exit states of interrupted executions. As discussed in Section 2.2, interrupted executions may require different cleanup behaviors at the exit stages according to the categories of the interruption triggers. For interrupted executions triggered by user terminations and hard exits, *SafeExit* uses all cleanup behaviors of the normal execution as expected behaviors. For interrupted executions triggered by fatal bugs, *SafeExit* select the expected behaviors from the cleanup behaviors of the normal execution. We use the following heuristic rule to determine the expected behaviors — in fatal-bug cases, the program at least cannot affect other programs or the operating system. According to this rule, *SafeExit* needs to select the behaviors that can potentially affect other programs or operating system as the expected behaviors.

To achieve this, we study the real-world cleanup behaviors and classify them into different types. These behavior types are defined once and broadly applicable to any interrupted executions. Then, we manually select the behavior types that can potentially affect other programs or operating system. Finally, *SafeExit* automatically classifies each normal behavior into one of the types. The behaviors belong to the selected types will be regarded as expected behaviors for fatal-bug executions.

The target programs of the study include 38 widely used real-world programs across 10 software domains. The programs have different types, i.e., GUI and CLI, server and client. All programs are open source, mature (about 10 years or longer development history), active (released in the last year, committed in the last month), and written in C/C++. The full list is shown in Table 4. The process to automatically extract cleanup behaviors is in Section 4.1.3.

We totally summarize 15 behavior types according to their system calls and resource types. The result is shown in Table 2, as well as their impact and fix suggestions. For example, a program may delete a PID file (Type 1) or send a message to the *Xorg* daemon (Type 14) at its exit stage. We find 10 out of the 15 types can potentially affect other programs or operating system. For example, if a program fails to delete the PID file when exits, users or other programs may believe the program is still running (Type 1). If a program does not kill its children processes properly, the processes would still occupy the system resources (Type 7).

Algorithm 1 Pseudo-code of learning prediction models.

Require: input scenario variables V, interruption triggers I, taxonomy of cleanup behaviors T
Ensure: M (MsB) $\leftarrow S \times I$
1: S = SamplesInputScenario(V)
2: **for** each i in [1, $|S|$] **do**
3: NT_i = TraceSystemCall(S_i, NULL)
4: NB_i = ExtractBehaviorsFromTrace(NT_i)
5: **for** each j in [1, $|I|$] **do**
6: $InT_{i,j}$ = TraceSystemCall(S_i, I_j)
7: $InB_{i,j}$ = ExtractBehaviorsFromTrace($InT_{i,j}$)
8: **end for**
9: **end for**
10: $UNB = NB_1 \cup NB_2 \cup \ldots \cup NB_{|S|}$
11: **for** each i in [1, $|S|$] **do**
12: **for** each j in [1, $|I|$] **do**
13: $ExB_{i,j}$ = FilterExpectedBehaviors(NB_i, I_j, T)
14: $MsB_{i,j}$ = [0, 0, ..., 0], where $|MsB_{i,j}| = |UNB|$
15: **for** each k in [1, $|UNB|$] **do**
16: **if** $UNB_k \in ExB_{i,j}$ and $UNB_k \notin InB_{i,j}$ **then**
17: $MsB_{i,j}[k] = 1$
18: **end if**
19: **end for**
20: **end for**
21: **end for**
22: M = TrainPredictionModel($S \times I$, $MsB_{i,j}$)

4 *SAFEEXIT* APPROACH

SafeExit contains two phases: offline learning, and online monitoring. The first phase trains a missing behavior predictor that given an input scenario, it can predict if an ungraceful exit occurs, as well as the missing behaviors. In the second phase, the predictor is deployed in the production environment to monitor the exit state of an interrupted execution for detecting ungraceful exit and its root cause (i.e., the missing behaviors).

4.1 Learning Prediction Models

In the first phase, *SafeExit* trains prediction models to predict ungraceful exit and the associated missing behaviors with respect to an input scenario and a particular trigger of interrupted execution (e.g., SIGINT). The input scenario is modeled as <C, W, U>, where C is a list of configuration options, W contains the workload values, and U includes user input parameters.

Algorithm 1 summarizes the main steps of training prediction models. To build the dataset for training, *SafeExit* first generates a set of input scenarios S using a well-known N-wise sampling method [44] (Line 1). It then runs each input scenario S_i against the program to collect a system call trace NT_i at the exit stage of the normal execution and extracts the normal cleanup behaviors NB_i (Lines 3–4). Next, for each interrupted execution with respect to a signal type I_j (e.g., SIGINT), *SafeExit* collects a system call trace $InT_{i,j}$ at the exit state and extracts its interrupted cleanup behaviors $InB_{i,j}$ (Lines 5–8). *SafeExit* obtains a list of union cleanup behaviors UNB from normal executions for all input scenarios.

After that, by using the taxonomy of cleanup behaviors (Section 3), *SafeExit* extracts the expect behaviors (ExB) for each interrupted execution with respect to an input scenario and a trigger

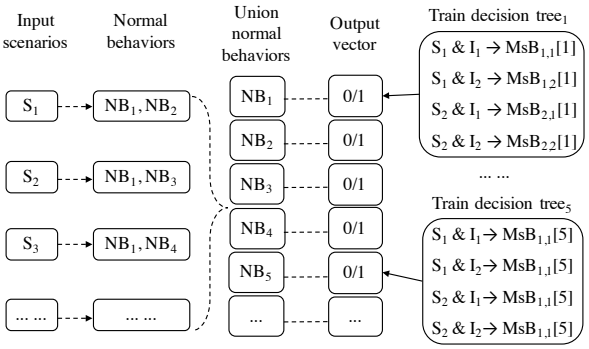

Figure 3: Overview of the prediction model.

(Line 13). For each behavior in UNB, it will be regarded as a missing behavior (MsB) for the input scenario and the trigger, if it is their expected behavior, and does not appear in the corresponding interrupted behaviors (Lines 14–17). Finally, *SafeExit* collects the variables of the input scenario and interruption triggers as features, and the missing behavior vector as labels. The dataset is fed to decision trees implemented by scikit-learn [33] (Line 22). The overview of the prediction model is shown in Figure 3. Each bit of the output vector indicates a normal behavior, and *SafeExit* will train $|UNB|$ decision trees in total.

4.1.1 Input Sampling. The input scenario of a program contains configuration options, workloads values and user input parameters. An exhaustive search of all combinations of these variables will lead to an exponential explosion problem. To avoid this, we sample the combinations of the input variables. There are two factors that will affect the sampling process: 1) the sampling method; 2) the sampling density of one variable.

Sampling method. In combinatorial sampling, a well-known method is pair-wise (or 2-wise) sampling. For each pair of input parameters, the method uses carefully chosen samples to test all possible combinations of those parameters. Similarly, N-wise sampling can be considered as the generalized form of pair-wise sampling [44]. We will compare different sampling methods in Section 5.2, and apply the optimal method in *SafeExit*.

Sampling density of one variable. For integer-type variables, *SafeExit* needs to sample certain values instead of enumerating all possible values (e.g., the number of HTTP request may range from 1 to 50000). The sampling density can also affect the predicting accuracy and training efficiency. In this regard, we evaluate different sampling density in Section 5.2, and apply the optimal parameter in *SafeExit*.

The sampling process is implemented by the *pict* [26] tool. Users need to provide the input variables and their value ranges. Users can also input the constraints across variables, and *SafeExit* can avoid the combinations that break the constraints. All the user inputs are optional. In the case that no variable is provided, *SafeExit* only generates one input scenario including the default configuration and an empty workload.

4.1.2 Exit Stage Tracing. An ungraceful exit potentially affects the system state, which can in turn affect the execution of other program processes. Since the interaction between programs and the system is often performed by system calls, *SafeExit* traces system

calls of the target program to extract cleanup behaviors at the exit stage for both the normal execution and the interrupted executions, and only focuses on the system calls happening after issuing the exit command.

There are two problems in obtaining the exit traces. First, an exit trace can be long, such as when the system is performing certain repetitive actions (e.g., delete a large number of files). Therefore, tracing all system calls can be expensive. The second is determining when to issue an interruption event (i.e., signals) to obtain an interrupted execution, since exiting before the program reaches a stable state (i.e., after the workload is processed) and after that may significantly affect exit behaviors.

Tracing relevant system calls. To reduce the overhead of tracing, *SafeExit* traces only the system calls that may affect the state of the operating system, which is the root cause of UE problems. Specifically, a *relevant system call* satisfies the following two properties: 1) the call has side effects beyond the current process, and 2) the side effects are still effective after the process exits. For example, the system call, *read*, can affect the variables inside the process, while *write* can affect files or other processes, and the effects are still effective after the process exits. Therefore, only *write* is a system call of interest and thus traced. Another example is *close*. *close* will delete a file descriptor, which will be deleted when the process exits anyway. Failing to *close* cannot affect the operating system or other processes after the process exits. *SafeExit* also considers the arguments within a system call. For example, when calling *open* with the argument "O_CREAT", it will create a new file, which is a long-term side effect. Otherwise, *open* only creates a file descriptor, which will be closed when the process exits. Another example is *futex*, it will change the lock state when calling with "WAKE", and do nothing when calling with "WAIT".

Toward this end, we use *strace* [37] to collect the system calls of the 38 programs in Table 4, and find 147 system calls are used in their exit stages. Finally, we summarize 37 out of 147 system calls that as relevant system calls. *SafeExit* only traces the relevant system calls with relevant arguments.

The points of exit. In general, there are three options for programs when exiting during workload. The simplest option is to exit immediately, which means the program tries to quit quickly without any operation on the current workload. Another option is to exit after serving. In this situation, the program will first finish the workload, then exit. For example, a web server will finish serving the current request before exiting. The last option is to exit after rollback. Typical examples are database servers, which can rollback the current operation for some problem situations.

When exiting immediately, programs only perform some necessary cleanup behaviors. When exiting after serving, programs perform cleanup behaviors as well as workload behaviors. When exiting after rollback, programs perform rollback behaviors before cleanup behaviors. Among the above behaviors, the rollback behaviors have been well studied by existing works that focus on persistency bugs. The task of *SafeExit* is to filter out the workload behaviors. This problem can be solved by simply issuing the exit commands after the workload, so the workload behaviors are wiped out naturally. In this case, all exit behaviors can be regarded as cleanup behaviors for *SafeExit*.

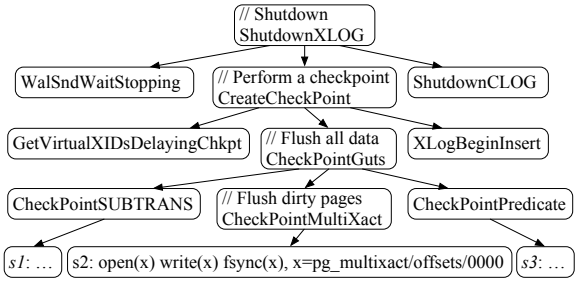

Figure 4: Example of the task tree in PostgreSQL.

4.1.3 Extracting Cleanup Behaviors. Reporting missing system calls is too low level to help users understand the root cause of UE problems. *SafeExit* extracts high-level behaviors from the system calls that perform the same task. For example, in the exit stage of *MySQL*, there is the following sequence:

1. *open (ib_buffer_pool.incomplete, O_CREAT...)*;
2. *write (ib_buffer_pool.incomplete, ...)*;
3. *unlink (ib_buffer_pool)*;
4. *rename (ib_buffer_pool.incomplete, ib_buffer_pool).*

The system call *open* creates a temp file, which will be written by *write*. After that, *unlink* deletes the file *ib_buffer_pool*, to which the temp file will be renamed. This is a common practice to write an important file, since writing the file directly may lead to data loss if the writing operation fails. There are two steps to extract behaviors. First, *SafeExit* extracts the system-call sequence that have related resources, *e.g.*, the file names in above example. Second, multiple system-call sequences may perform a higher level task, and *SafeExit* needs to cluster the sequences.

Extract system-call sequence. Programs may have multiple threads, of which the system calls may interleave. *SafeExit* first classifies the system calls according to their threads identifiers (TIDs). Second, for the trace of each thread, *SafeExit* splits it into segments, each of which contains the same resources, *e.g.*, file name. Third, *SafeExit* scans the trace again and detect related resources, e.g., *ib_buffer_pool.incomplete* and *ib_buffer_pool* and related since they both are arguments of *rename*. Fourth, *SafeExit* joints the serial segments that have related sources as a sequence. For above example, *SafeExit* will get the sequence "(open x) (write x) (unlink y) (rename x y)", where x=*ib_buffer_pool.incomplete*, y=*ib_buffer_pool*. Finally, *SafeExit* removes loops in the sequences. For example, the sequence "(creat x) (write x) (fsync x) (write x) (fsync x) (rename x y)" will become "(creat x) (write x) (fsync x) (rename x y)"

Cluster sequences (optional). *SafeExit* clusters sequences that perform a high-level task if provided the source code. For example, in Figure 4, the system calls in the sequence *s2* are invoked by *CheckPointMultiXact*, which is further invoked by *CheckPointGuts*. At the same time, both *CheckPointSUBTRANS* and *CheckPointPredicate* are invoked by *CheckPointGuts*, and each of them invokes its own sequence, *i.e.*, *s1* and *s3*. In this situation, *SafeExit* clusters sequences for different levels of tasks, and finally build a task tree. For example, the higher level task of *s2* is *CheckPointMultiXact* which flushes dirty pages, while the higher level task of *s1*, *s2* and *s3* is *CheckPointGuts* which flushes all data.

To build the task tree, *SafeExit* first collects the call stack of each system call, and get the longest common call-stack prefix for

Table 3: Examples of execution-specific information.

Before normalization	After normalization
open(Chromium.vIwTTl...) open(Chromium.rweDBR...)	open(Chromium.******...)
writev(unix:[314384->314385]>...) writev(unix:[428732->428733]>...)	writev(.X11-unix/X0...)
kill(2391...) kill(4891...)	kill(PID_1...)

each sequence. Second, *SafeExit* finds the longest common prefix for different sequences, and clusters the sequences that have a common prefix. This process will be repeated recursively until the top task. Third, for each node of the task tree, *SafeExit* records its function name and comments as the semantics of this task.

4.1.4 Detecting Missing Cleanup Behaviors. *SafeExit* detects the missing behaviors in an interrupted execution *InE* by comparing the extracted cleanup from *InE* with the expected behaviors (Section 3) in the normal execution *NE*. There are two main tasks to achieve this. First, a program may not exit when an interruption trigger happens. For example, *MySQL* will ignore the *sigint* signal. In this case, *MySQL* will not cause any UE problem even all expected behaviors are missing, since *MySQL* does not exit at all. *SafeExit* needs to recognize the reactions of the programs for interruption triggers. Second, the traces of two executions may be different even under the same input scenario. For example, in the first example of Table 3, the file names have random suffixes, which are different in two executions. *SafeExit* needs to eliminate the execution-specific information when comparing behaviors from two executions.

Program reactions of interruptions. We classify program reactions of interruptions into two categories: good practices and bad practices. In good practices, the program may *survive* from the situation, in this case, the main process of the program will still exist after the trigger happens. Also, the program can *recover* from the situation, which means all processes exit and new processes will be generated instead. Another good practice is *graceful exit*, which means all necessary behaviors have been performed successfully before the program exits. On the other hand, the bad practice is *ungraceful exit*, which means any or part of cleanup behaviors are missing after the program exits. For the cases of *survive* and *recover*, *SafeExit* does not report any missing behavior, which means all bits of the missing behavior vector are zeroes.

System call normalization. *SafeExit* normalizes the following execution-specific information when comparing the system calls of interrupted behaviors and expected behaviors: (a) File name. *SafeExit* keeps the common substring of the file names and changes the random suffix into a string of '*' with the same length. (b) Output target. In the second example of Table 3, the first argument of *writev* is a Unix-type socket, whose target is an inode number. *SafeExit* uses the utility *lsof* to get the responding file of the inode number and replaces the number with a real file name. (c) Process identifier. The PIDs always change in different executions. *SafeExit* sorts the PIDs of each execution in ascending order, and normalizes the PIDs to their ranks. In the last example, the PID of the first argument in *kill* is normalized to 'PID_1', which means the process with the smallest PID in the current program.

4.2 Online Monitoring and Predicting

The online-monitoring component is a system-level monitor, which is implemented by the *auditd* [2] tool (used to record system calls and signals). *SafeExit* does not do any form of program instrument, thus the overhead is very limited. When an interruption is detected, *SafeExit* predicts the missing behaviors by using the real input scenario and detected interruption trigger. If the input scenario variables are not provided in the offline training phase, *SafeExit* will report the missing behaviors of default configuration and empty workload. Otherwise, *SafeExit* can report the predicted missing behaviors to users.

5 EVALUATION

To evaluate *SafeExit*, we consider three research questions:

RQ1: What behaviors are done when programs exit normally?

RQ2: What are the root causes of ungraceful exits?

RQ3: How accurate does *SafeExit* predict missing behaviors?

In *SafeExit*, there are two main functions. The first is detecting missing behaviors for a given input scenario. This includes two steps: recognize normal behaviors (RQ1) and detect missing behaviors (RQ2). The second is predicting missing behaviors for an unseen input scenario (RQ3). In this section, we evaluate *SafeExit* on 38 mature and active programs across 10 domains shown in Table 4.[1]

5.1 Missing Behavior of a Given Input Scenario

5.1.1 RQ1: Exit Behaviors in Normal Executions. The insight of *SafeExit* is to extract behaviors that are done during the exit stage of normal execution and use these behaviors as oracles for detecting the missing behaviors in the interrupted executions so as to localize the root causes of ungraceful exits. Therefore, we need to know what have been done when programs exit normally. We run the benchmark programs under the default configuration and empty workload, then obtain the exit behaviors from the normal executions using the approach described in Section 4.1.3.

The results are shown in the fourth column of Table 4. For example, *Apache* performs four types of behaviors at the exit stage when exiting normally, *i.e.*, deleting the PID file, killing children processes, writing log files and sending messages to its children processes. One program may perform multiple behaviors that belong to the same behavior type. For example, *Apache* writes to different log files (i.e., *access_log* and *error_log*).

In total, *SafeExit* detects 133 types of behaviors from the 38 programs (about 3.5 for each program). Besides, every program will stop its main process at the exit stage, for space reason, we do not list the behavior. Two out of the 38 programs do not perform any cleanup behaviors at their exit stages, i.e., *Bftpd* and *MPV*. Most programs with GUI will send messages to other daemons including *Xorg* (23/23), *dbus* (15/23), and *ibus* (13/23), while most servers will delete PID files (9/11).

These results indicate that most programs (36/38) have cleanup behaviors at their exit stages, supporting our assumption that the expected exit behaviors can be obtained from the behavior of the

[1] For ease of reproducing, the *SafeExit* source code and all data will be available in https://github.com/ZhouyangJia/SafeExit.

Table 4: Exit behaviors of normal execution and reactions of interrupted executions.

Domain	Program-version	Type	Exit behaviors of normal executions	Program reactions[1]	# Missing behaviors
Web Server	Apache-2.4.37	CLI/Server	D-PID, K-CHLD, W-LOG, S-CHLD	S/G/B/G B/B/B/B/B/S/B/S	58
	Lighttpd-1.4.45	CLI/Server	D-PID, W-LOG	S/G/B/G B/B/B/B/B/S/B/B	16
	Nginx-1.15.6	CLI/Server	D-PID, S-CHLD	S/G/G/G B/B/B/B/B/S/B/B	14
Broswer	Brave-0.56.15	GUI/Client	D-SHM, D-LCK, U-LCK, K-CHLD, W-PCFG, W-PSTAT, S-CHLD, S-PROG	G/G/B/G B/B/B/B/B/S/B/B	493
	Chromium-65.0.3325	GUI/Client	D-SHM, D-LCK, U-LCK, K-CHLD, W-PCFG, W-PSTAT, S-CHLD, S-PROG	G/G/B/G B/B/B/B/B/S/B/B	347
	Firefox-59.0.2	GUI/Client	D-TMP, D-SHM, D-LCK, U-LCK, W-PCFG, W-PSTAT, S-PROG, S-NET	B/B/B/B R/R/R/R/R/S/B/B	230
	GnomeWeb-3.28.1	GUI/Client	D-TMP, D-SHM, U-LCK, W-PCFG, W-SCFG, W-PSTAT, S-PROG, S-NET	B/G/B/G B/B/B/B/B/S/B/B	126
Database	MonetDB-1.7	CLI/Server	D-PID, D-SOCK, U-LCK, W-LOG	S/G/G/G B/B/B/B/B/S/B/B	48
	MySQL-8.0.13	CLI/Server	D-PID, D-SOCK, D-TMP, D-LCK, W-PSTAT	S/S/G/G B/B/B/B/B/S/B/B	182
	PostgreSQL-11rc1	CLI/Server	D-PID, D-SOCK, D-TMP, D-SHM, D-LCK, K-CHLD, W-PSTAT	S/G/B/G B/B/B/B/B/S/B/S	149
	SQLite3-3.22.0	CLI/Client	W-PSTAT	B/S/B/B B/B/B/B/B/B/B/B	11
Email	Geary-0.12-dev	GUI/Client	S-PROG	B/B/B/B B/B/B/B/B/S/B/B	11
	OpenSMTPD-6.0.3	CLI/Server	D-PID, D-SOCK, S-PROG	B/G/B/G B/B/B/B/B/S/B/B	25
	Postfix-3.3.0-1	CLI/Server	S-PROG	S/G/G/G G/B/G/B/G/S/B/B	4
	Thunderbird-52.7.0	GUI/Client	D-TMP, D-LCK, U-LCK, W-PCFG, W-PSTAT, S-PROG, S-NET	B/B/B/B R/R/R/R/R/S/B/B	119
FTP	Bftpd-4.4	CLI/Server	-	-	-
	FileZilla-3.28.0	GUI/Client	D-TMP, U-LCK, W-PCFG, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	44
	gCommander-1.4.8	GUI/Client	W-PCFG, S-PROG	B/S/B/B B/B/B/B/B/S/B/S	54
	ProFTPD-1.3.5e	CLI/Server	D-PID, D-LCK, U-LCK, W-LOG, W-PSTAT	S/G/G/G G/G/G/G/G/S/G/B	5
	Pure-FTPd-1.0.46	CLI/Server	D-PID	G/G/G/G B/B/B/B/B/S/G/B	6
Text Editor	Emacs-25.2.2	GUI/Client	D-TMP, W-PSTAT, S-PROG	G/G/G/G G/G/G/G/G/S/G/G	0
	Gedit-3.28.1	GUI/Client	W-PCFG, W-SSTAT, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	41
	Vim-8.0.1453	CLI/Client	D-TMP, W-PSTAT	B/B/B/B G/G/G/G/G/S/G/G	5
Image Editor	Darktable-2.4.2	GUI/Client	D-LCK, W-PCFG, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	55
	GraphicMagick-1.3.28	GUI/Client	S-PROG	B/B/B/B B/B/B/B/B/B/B/B	12
	gThumb-3.6.1	GUI/Client	W-PCFG, W-SCFG, W-PSTAT, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	55
	KolourPaint-17.12.3	GUI/Client	S-PROG	B/B/B/B B/B/B/B/B/S/B/B	11
	SynfigStudio-1.2.1	GUI/Client	D-SOCK, W-PCFG, W-PSTAT, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	66
Player	Audacious-3.9	GUI/Client	W-PSTAT, S-PROG	G/G/G/G B/B/B/B/B/S/B/B	30
	MPV-0.27.2	CLI/Client	-	-	-
	Rhythmbox-3.4.2	GUI/Client	W-PCFG, W-PSTAT, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	66
	SMPlayer-18.2.2	GUI/Client	D-TMP, D-LCK, U-LCK, W-PCFG, W-PSTAT, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	204
	Totem-3.26.0	GUI/Client	W-PSTAT, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	37
	VLC-3.0.1	GUI/Client	U-LCK, W-PCFG, W-PSTAT, S-PROG	G/G/G/G B/B/B/B/B/S/B/B	41
Network Monitor	Netsniff-ng-0.6.4	CLI/Client	W-SCFG	S/G/G/G B/B/B/B/B/B/B/B	40
	Wireshark-2.4.5	GUI/Client	W-PSTAT, S-PROG	B/B/B/B B/B/B/B/B/S/B/B	43
Office Suite[2]	LibreOffice-6.0.3.2	GUI/Client	D-TMP, D-LCK, W-PCFG, W-PSTAT, W-LOG, S-PROG	S/B/B/B B/R/R/R/R/S/B/B	168
	OpenOffice-4.1.6	GUI/Client	D-TMP, D-LCK, W-PCFG, W-LOG, S-PROG	S/B/B/B B/R/R/R/R/S/B/B	45

[1] "S/R/G/B" mean the different reactions, where "S" for survival, "R" for recovery, "G" for graceful exit, and "B" for missing behaviors. The reactions of interrupted executions are listed in the order of "sighup/sigint/sigquit/sigterm" (user terminations) and "sigill/sigabrt/sigbus/sigfpe/sigsegv/sigpipe/sigxcpu/sigxfsz" (fatal bugs).

[2] Each office suit contains multiple programs, e.g., writer, calc, impress, whose exit behaviors are exactly the same. Therefore, we regard them as one program.

program exit in a normal execution. As for the other two programs, *SafeExit* cannot determine whether there is an ungraceful exit. There is no false positive or false negative with regard to detecting exit behaviors, since the behaviors of a program are fixed for a given input scenario (we do not consider the order of behaviors in case of concurrency programs).

5.1.2 RQ2: Root Causes of Ungraceful Exits. The goal of RQ2 is to evaluate whether *SafeExit* can detect ungraceful exits and the root causes (i.e., missing behaviors) of the ungraceful exits. There are four reactions when programs encounter interruptions (i.e., survive, recover, graceful exit and ungraceful exit) according to Section 4.1.4. *SafeExit* first detects the reactions of the programs, then reports missing behaviors for ungraceful exits.

The results are shown in the rightmost two columns of Table 4. In the table, "S/R/G/B" indicate the reactions of the program for different interruption triggers, where "S" for survival, "R" for recovery, "G" for graceful exit, and "B" for missing behaviors (i.e., ungraceful exit). The reactions are listed in the order of user terminations (*sighup*, *sigint*, *sigquit*, *sigterm*) and fatal bugs (*sigill*, *sigabrt*, *sigbus*, *sigfpe*, *sigsegv*, *sigpipe*, *sigxcpu*, *sigxfsz*). For hard exits, e.g., *sigkill*, programs always exit ungraceful, thus we do not list them.

For example, *Apache* will read the configuration files and restart the server (children processes) when receiving *sighup*. *SafeExit* reports that *Apache* can survive from *sighup*, since the main process still exist after the signal. For *sigint* and *sigterm*, all processes will exit, and *SafeExit* detects all necessary behaviors are successfully performed. As for some fatal signals and *sigquit*, *SafeExit* finds *Apache* exit ungracefully and reports missing behaviors. For example, the main process will send termination signals to its children processes (K-CHLD) during normal exit. In interruptions triggered by bugs like SIGSEGV, the behavior is missing and the children processes become orphan processes. For space reason, we do not list the behaviors.

In the 36 programs that perform cleanup behaviors, *SafeExit* evaluates their reactions for 12 interruption triggers, i.e., 432 interrupted executions. *SafeExit* finds 292 out of the 432 (or 67.6%) executions exit ungracefully (about 8.1 on average for each program). Among the ungraceful exits, *SafeExit* detects 2861 missing behaviors in total (i.e., 9.8 for each ungraceful exit). Specifically, servers (i.e., web server, database server, FTP server, and email server) handle user terminations (5/40=12.5%) better than other situations, including clients for user terminations (76/104=73.1%), servers for fatal

Table 5: Target programs for the accuracy evaluation.

Program	# Var.	Dep.	Program	# Var.	Dep.
Apache	11	Yes	Gedit	12	No
Chromium	35	Yes	gThumb	15	No
MySQL	29	Yes	SMPlayer	38	Yes
Thunderbird	40	Yes	Wireshark	29	No
FileZilla	13	No	LibreOffice	36	No

# Var. = number of input-scenario variables: config options and workload variables.
Dep. = dependence: whether the exit behaviors are dependent on its input scenario.

bugs (58/80=72.5%), and clients for fatal bugs (153/208=73.4%). This is because many server functions (e.g, start, stop, restart or update config) are implemented by user-termination signals. There is one program (i.e., *Emacs*) that handles all interruptions well, and one program (i.e., *GraphicMagick*) that exits ungracefully for any interruption. Most programs (33/36) survive a *sigpipe* signal.

These results suggest that ungraceful exit is the most common reaction (67.6%) when programs encounter interruptions. *SafeExit* can be widely used to help users to solve problems caused by ungraceful exits. In *SafeExit*, the accuracy of detecting missing behaviors is dependent on the quality of code in the normal exit. For example, if developers perform an unnecessary behavior in normal exit code, *SafeExit* will report a false positive when the behavior is missing. If developers miss a necessary behavior in normal exit code, *SafeExit* will report a corresponding false negative.

5.2 RQ3: Accuracy of Behavior Prediction

Program behaviors are affected by input scenarios like configurations and workloads. For default configuration and empty workload, *SafeExit* can precisely report the missing behaviors when ungraceful exit happens, since the program behaviors are fixed. As for other scenarios, *SafeExit* needs to sample input scenarios, build a prediction model, and predict the missing behaviors. To achieve this, *SafeExit* implements a script for each target program to execute its input scenarios automatically. In general, for programs from one domain, their input scenarios are similar. Therefore, we select one highly configurable program from each domain to evaluate.

Table 5 shows the target programs. For each program, due to the large input space, we manually choose some configuration options and workload variables. During this process, we avoid choosing configuration options that will not affect the exit behaviors of the program. For example, in *Apache*, the configuration *ServerName* gives the name and port that the server uses to identify itself, which can not affect the exit behaviors. Nevertheless, the exit behaviors of five programs are still independent of the chosen variables. In these cases, *SafeExit* always achieves 100% prediction accuracy. As for the programs whose exit behaviors are dependent on their input scenarios, we evaluate the predicting accuracy and training efficiency when using different sampling methods and sampling densities as discussed in Section 4.1.1.

For each program, we randomly sample 100 input scenarios and 100 interruption triggers as the test set. Input scenarios in the test set are different from the ones in the training phase. Then *SafeExit* outputs a vector of missing behaviors for each test case, i.e., an input scenario and an interruption trigger. We use two widely-used metrics, *i.e.*, *precision* and *recall*, to evaluate the prediction accuracy.

Table 6: Accuracy and efficiency of different sampling methods and sampling densities.

Program	Sampling method			Sampling density		
	1-wise	2-wise	3-wise	D=2	D=4	D=8
Apache	66.8%†	85.7%	85.7%	76.2%	85.7%	98.0%
	6‡	33	174	14	33	112
Chromium	57.6%	67.6%	70.7%	66.5%	67.6%	75.1%
	4	20	67	13	20	94
MySQL	88.6%	92.9%	94.8%	91.8%	92.9%	95.2%
	5	36	197	20	36	115
Thunderbird	95.7%	96.1%	97.1%	94.1%	96.1%	97.6%
	4	20	89	13	20	72
SMPlayer	82.1%	95.8%	95.8%	95.8%	95.8%	96.6%
	4	21	84	13	21	73
Average	78.2%	87.6%	88.8%	84.9%	87.6%	92.5%
	4.6	26	122.2	14.6	26	93.2

† F_{score} of missing-behavior prediction. ‡ Number of training samples.

Suppose there are N behaviors in the vector, *i.e.*, $b_1, b_2, ..., b_N$. Let $X(i)$ denote the total number of test cases that miss b_i, $Y(i)$ denote the number of test cases that *SafeExit* predicts b_i will miss, and $Y'(i)$ denote the number of test cases that b_i is truly missed. The predicting precision of b_i, $P(i)$, is the ratio of $Y'(i)$ to $Y(i)$, while the predicting recall, $R(i)$, is the ratio of $Y'(i)$ to $X(i)$. We further calculate the averaged precision $\bar{P}$ and recall $\bar{R}$ of all behaviors. If a cleanup behavior is missed in the test set, but never happens in the training stage, its precision and recall rates are regarded as zeroes. For the ease of comparison, we finally calculate F_{score}, which is the harmonic mean of $\bar{P}$ and $\bar{R}$.

As shown in Table 6, we evaluate three sampling methods, i.e., 1-wise, 2-wise and 3-wise, and three sampling densities, i.e., D=2, D=4, D=8. For an integer variable, *SafeExit* samples D values:

$$\lfloor \frac{Max - Min}{D - 1} * (i - 1) \rfloor + Min, i \in [1, D]$$

where Max and Min are the ranges of the variable. For example, given a variable $v \in [1, 10]$, the sampled values are 1, 4, 7, 10 when D=4. In Table 6, we first evaluate the sampling methods (using D=4). For each method, we provide the F_{score} and the number of training samples. In term of the averaged accuracy, the 2-wise (87.6%) is much higher than 1-wise (78.2%), and the 3-wise (88.8%) is close to 2-wise (87.6%). While the numbers of samples are increasing linearly, thus the 3-wise takes much more time. Therefore, *SafeExit* chooses 2-wise as the optimal method. Next, we evaluate different sampling densities (using 2-wise). The accuracy improvements from D=2 to D=4 and from D=4 to D=8 are similar. *SafeExit* uses D=8 as the default density since its high accuracy, while users can choose custom values according to their preference on predicting accuracy or training efficiency.

These results show that the exit behaviors of a program may or may not be dependent on its input scenarios. For programs whose exit behaviors are dependent on their input scenarios, different parameters (sampling method and sampling density) can affect the predicting accuracy and training efficiency. Under the optimal parameters (2-wise and D=8), *SafeExit* can achieve a high accuracy (92.5%) on predicting missing behaviors while using limited input-scenario samples (93.2 on average) in the training phase.

6 DISCUSSION

Bugs during normal exiting. In *SafeExit*, the key insight is to learn the expected behaviors of ungraceful exits from normal exits. We assume that developers carefully designed the workflow of normal exit, and the behaviors during normal exiting can server as oracles. In the cases that there are bugs during normal exit, *SafeExit* cannot eliminate the effects of the bugs, resulting in false positives or false negatives. In Section 5.1, we summarized several patterns that what behaviors should be performed at the exit stage for a certain type of programs. Inspired by these patterns, we can detect the bugs of normal exit by mining exit rules across programs. We leave this feature to further work.

Recover at the next startup. Some programs may exit ungracefully when fatal situations happen and recover the corrupted states at the next startup. For example, if there is a fatal bug in *PostgreSQL*, the cleanup behaviors will be performed at the next startup (after version 8.3). Another example is the office suite, which can recover both user terminations and fatal bugs at the next startup. *SafeExit* does not analyze the program behaviors of the startup stage, thus still reports the missing behaviors in this situation. The results of *SafeExit* are valid until the next startup, since the operating system contains corrupted states between the time of the crash and the next startup.

Long-term running programs only. *SafeExit* needs to learn the behaviors that should be performed at the exit stage. The beginning of the exit stage is the time when clicking an exit button of GUI program, or issuing an exit command of a CLI program. This exit mechanism is used in long-term running programs, meaning there is an explicit exit action performed by users, the operating system or other programs. For some programs, however, there is no such exit action. Taking the compiler program as an example, a compiler starts when issuing the start command, and exits when finish compiling. In this case, there is no explicit exit action, thus *SafeExit* cannot recognize the exit stage of a compiler.

7 RELATED WORK

Exception-handling bugs. There is a long line of research focusing on exception-handling mechanisms [3–5, 8, 10, 14, 18, 19, 22, 25, 29–31, 41, 42]. For example, Ebert *et al*. [8] conducted a survey of 154 developers and an analysis of 220 exception handling bugs. Oliveira *et al*. [29] presented an empirical study on the relationship between the usage of Android abstractions and uncaught exceptions. Filho *et al*. [10] presented a study of the adequacy of the AspectJ language for modularizing exception handling code. Jakobus *et al*. [18] contrasted exception handling code across languages from 50 open source projects. Rahman *et al*. [31] proposed a context-aware code recommendation approach that recommends exception handling code examples. Barbosa *et al*. [3] presented a tool to recommend repairs of exception handling violations with aware of the global context. Weimer *et al*. [41, 42] presented a data-flow analysis for finding exception-handling bugs in Java programs. Exception handling does not always end up with an exit, for example, programs may create a new file when the library function *fopen* cannot find the target file. Therefore, existing work does not focus on cleanup behaviors on the exit stage. *SafeExit* is a complementary tool to the existing exception-handling techniques.

Data corruption and crash recovery. Prior efforts on persistent data consistency have looked at finding storage or distributed system errors [1, 6, 7, 11–13, 15, 16, 38, 39, 45, 46]. Gao *et al*. [13] presented a comprehensive study on 103 crash recovery bugs from four distributed systems. Gunawi *et al*. [15] proposed a testing framework for cloud recovery. Subramanian *et al*. [38] injected faults into the MySQL DBMS, and found corruption can greatly harm the system. Yang *et al*. [46] built a system, FiSC, for model checking file systems. Ganesan *et al*. [12] analyzed how modern distributed storage systems behave in the presence of file-system faults. Wang *et al*. [39] presented a comprehensive study on 138 real-world data corruption incidents reported in Hadoop bug repositories. Some works focus on persistency bugs in storage systems, which help to clean data corruption caused by problem situations. While *SafeExit* is designed to clean general states under ungraceful exit conditions. Other works are targeting errors during recovery, but can not help when programs try to exit. While we find the majority programs will exit ungracefully in problem situations.

Other related work. There has been some other research related to *SafeExit*. Similar to *SafeExit*, many works [9, 23, 24] mined software traces for various purposes, but few work focuses on missing behaviors of ungraceful exits. Wang *et al*. [40] presented an automated framework that can detect and validate race conditions hardware interrupt. While *SafeExit* focuses on the ungraceful exit problems caused by software interrupts. Shan *et al*. [34] addressed the problems of data loss, failure to resume/restart or resuming/restarting in the wrong state in Android applications. These problems are caused by incorrect handling of instance data and are easily triggered by just pressing the 'Home' or 'Back' buttons. While *SafeExit* is targeting the problems caused by ungraceful exits in PC programs.

8 CONCLUSIONS

This paper presented *SafeExit* a tool that can automatically detect and pinpoint the root causes of UE problems, which can help users to fix the problems with lightweight solutions without having to reboot the system. *SafeExit* learned the missing behaviors of interrupted executions under different input scenarios and interruption triggers, and built a prediction model to predict the missing behaviors of an unseen input scenario. During the learning phase, we studied the real-world cleanup behaviors to help *SafeExit* select the expected behaviors for different interrupted executions. Finally, we evaluated *SafeExit* by using 38 widely used programs from 10 domains. The experimental results showed that *SafeExit* can effectively extract the cleanup behaviors for normal executions, and detect the missing behaviors for interrupted executions. We also evaluated the effects of sampling method and sampling density for the training efficiency and predicting accuracy, helping *SafeExit* to choose the optimal parameters.

ACKNOWLEDGMENTS

This research was supported by National Key R&D Program of China No. 2017YFB1001802; NSFC No. 61872373 and 61872375; NSF grant CCF-1652149; High-End Generic Chips and Basic Software under grants No. 2017ZX01038104-002; China Scholarship Council.

REFERENCES

[1] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 331–346. https://doi.org/10.1145/2723372.2723711

[2] Auditd. 2018. The Linux Audit daemon. https://linux.die.net/man/8/auditd.

[3] Eiji Adachi Barbosa and Alessandro Garcia. 2018. Global-aware Recommendations for Repairing Violations in Exception Handling. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. ACM, New York, NY, USA, 858–858. https://doi.org/10.1145/3180155.3182539

[4] Eiji Adachi Barbosa, Alessandro Garcia, and Simone Diniz Junqueira Barbosa. 2014. Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *2014 Brazilian Symposium on Software Engineering*. IEEE Computer Society, Washington, DC, USA, 11–20. https://doi.org/10.1109/SBES.2014.19

[5] Eiji Adachi Barbosa, Alessandro Garcia, Martin P. Robillard, and Benjamin Jakobus. 2016. Enforcing Exception Handling Policies with a Domain-Specific Language. *IEEE Trans. Softw. Eng.* 42, 6 (June 2016), 559–584. https://doi.org/10.1109/TSE.2015.2506164

[6] Philip Bohannon, Rajeev Rastogi, S. Seshadri, Avi Silberschatz, and S. Sudarshan. 2003. Detection and Recovery Techniques for Database Corruption. *IEEE Trans. on Knowl. and Data Eng.* 15, 5 (Sept. 2003), 1120–1136. https://doi.org/10.1109/TKDE.2003.1232268

[7] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. ACM, New York, NY, USA, 18–37. https://doi.org/10.1145/2815400.2815402

[8] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An exploratory study on exception handling bugs in Java programs. *Journal of Systems and Software* 106 (2015), 82 – 101. https://doi.org/10.1016/j.jss.2015.04.066

[9] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.* 69, 1-3 (Dec. 2007), 35–45. https://doi.org/10.1016/j.scico.2007.01.015

[10] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília Mary F. Rubira. 2006. Exceptions and Aspects: The Devil is in the Details. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 152–162. https://doi.org/10.1145/1181775.1181794

[11] Daniel Fryer, Kuei Sun, Rahat Mahmood, Tinghao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying File System Consistency at Runtime. *ACM Trans. Storage* 8, 4, Article 15 (Dec. 2012), 29 pages. https://doi.org/10.1145/2385603.2385608

[12] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to File-System Faults. *ACM Trans. Storage* 13, 3, Article 20 (Sept. 2017), 33 pages. https://doi.org/10.1145/3125497

[13] Yu Gao, Wensheng Dou, Feng Qin, Chushu Gao, Dong Wang, Jun Wei, Ruirui Huang, Li Zhou, and Yongming Wu. 2018. An Empirical Study on Crash Recovery Bugs in Large-scale Distributed Systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM, New York, NY, USA, 539–550. https://doi.org/10.1145/3236024.3236030

[14] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Jian Lü, and Zhendong Su. 2016. Automatic Runtime Recovery via Error Handler Synthesis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 684–695. https://doi.org/10.1145/2970276.2970360

[15] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. 2011. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI'11)*. USENIX Association, Berkeley, CA, USA, 238–252. http://dl.acm.org/citation.cfm?id=1972457.1972482

[16] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2008. SQCK: A Declarative File System Checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, Berkeley, CA, USA, 131–146. http://dl.acm.org/citation.cfm?id=1855741.1855751

[17] Chris Hoffman. 2018. Why Does Rebooting a Computer Fix So Many Problems? https://www.howtogeek.com/173760/htg-explains-why-does-rebooting-a-computer-fix-so-many-problems/.

[18] Benjamin Jakobus, Eiji Adachi Barbosa, Alessandro Garcia, and Carlos Jose Pereira de Lucena. 2015. Contrasting exception handling code across languages: An experience report involving 50 open source projects. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE Computer Society, Washington, DC, USA, 183–193. https://doi.org/10.1109/ISSRE.2015.7381812

[19] Zhouyang Jia, Shanshan Li, Xiaodong Liu, Xiangke Liao, and Yunhuai Liu. 2018. SMARTLOG: Place error log statement by deep understanding of log intention. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, Washington, DC, USA, 61–71. https://doi.org/10.1109/SANER.2018.8330197

[20] Dennis Kennedy. 2013. Why Rebooting Works So Well. https://www.lawtechnologytoday.org/2013/11/why-rebooting-works-so-well/.

[21] Thorin Klosowski. 2013. Why Rebooting Your Computer Fixes Problems. https://lifehacker.com/why-rebooting-your-computer-fixes-problems-1445670330.

[22] Yan Lei, Chengnian Sun, Xiaoguang Mao, and Zhendong Su. 2018. How test suites impact fault localisation starting from the size. *IET Software* 12, 3 (2018), 190–205. https://doi.org/10.1049/iet-sen.2017.0026

[23] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '09)*. ACM, New York, NY, USA, 557–566. https://doi.org/10.1145/1557019.1557083

[24] David Lo, Siau-Cheng Khoo, and Chao Liu. 2007. Efficient Mining of Iterative Patterns for Software Specification Discovery. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '07)*. ACM, New York, NY, USA, 460–469. https://doi.org/10.1145/1281192.1281243

[25] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based Statistical Fault Localization. *J. Syst. Softw.* 89, C (March 2014), 51–62. https://doi.org/10.1016/j.jss.2013.08.031

[26] Microsoft. 2018. pict. https://github.com/Microsoft/pict/.

[27] Rob Miles. 2016. Explained: why a reboot is the go-to computer fix. http://theconversation.com/explained-why-a-reboot-is-the-go-to-computer-fix-65261.

[28] Nginx. 2019. Nginx. https://www.nginx.com/.

[29] Juliana Oliveira, Deise Borges, Thaisa Silva, Nelio Cacho, and Fernando Castor. 2018. Do android developers neglect error handling? a maintenance-Centric study on the relationship between android abstractions and uncaught exceptions. *Journal of Systems and Software* 136 (2018), 1 – 18. https://doi.org/10.1016/j.jss.2017.10.032

[30] Juliana Oliveira, Nelio Cacho, Deise Borges, Thaisa Silva, and Fernando Castor. 2016. An Exploratory Study of Exception Handling Behavior in Evolving Android and Java Applications. In *Proceedings of the 30th Brazilian Symposium on Software Engineering (SBES '16)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2973839.2973843

[31] Mohammad Masudur Rahman and Chanchal K. Roy. 2014. On the Use of Context in Recommending Exception Handling Code Examples. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation (SCAM '14)*. IEEE Computer Society, Washington, DC, USA, 285–294. https://doi.org/10.1109/SCAM.2014.15

[32] Paul Randal. 2011. Survey results on rebooting - is it good or bad? https://www.sqlskills.com/blogs/paul/survey-results-on-rebooting-is-it-good-or-bad/.

[33] Scikit-learn. 2018. Scikit-learn: Machine Learning in Python. https://scikit-learn.org/stable/.

[34] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. 2016. Finding Resume and Restart Errors in Android Applications. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 864–880. https://doi.org/10.1145/2983990.2984011

[35] Tina Sieber. 2014. Why Does Rebooting Your Computer Fix So Many Issues? https://www.makeuseof.com/tag/rebooting-computer-fix-many-issues/.

[36] Tina Sieber. 2018. Why Does Restarting Seem to Fix Most Computer Problems? https://www.lifewire.com/why-does-restarting-seem-to-fix-most-computer-problems-2624569.

[37] Strace. 2018. Strace: linux syscall tracer. https://strace.io.

[38] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jeffrey F Naughton. 2010. Impact of disk corruption on open-source DBMS. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE Computer Society, Washington, DC, USA, 509–520. https://doi.org/10.1109/ICDE.2010.5447821

[39] Peipei Wang, Daniel J. Dean, and Xiaohui Gu. 2015. Understanding Real World Data Corruptions in Cloud Systems. In *Proceedings of the 2015 IEEE International Conference on Cloud Engineering (IC2E '15)*. IEEE Computer Society, Washington, DC, USA, 116–125. https://doi.org/10.1109/IC2E.2015.41

[40] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao, and Xuandong Li. 2017. Automatic Detection and Validation of Race Conditions in Interrupt-driven Embedded Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*. ACM, New York, NY, USA, 113–124. https://doi.org/10.1145/3092703.3092724

[41] Westley Weimer and George C. Necula. 2004. Finding and Preventing Run-time Error Handling Mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, New York, NY, USA, 419–431. https://doi.org/10.1145/

1028976.1029011

[42] Westley Weimer and George C. Necula. 2008. Exceptional Situations and Program Reliability. *ACM Trans. Program. Lang. Syst.* 30, 2, Article 8 (March 2008), 51 pages. https://doi.org/10.1145/1330017.1330019

[43] Wikipedia. 2017. Graceful exit. https://en.wikipedia.org/wiki/Graceful_exit.

[44] Wikipedia. 2019. All-pairs testing. https://en.wikipedia.org/wiki/All-pairs_testing.

[45] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 131–146. http://dl.acm.org/citation.cfm?id=1298455.1298469

[46] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 393–423. https://doi.org/10.1145/1189256.1189259