



Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps

Marco Brocanelli

Department of Electrical and Computer Engineering
The Ohio State University
brocanelli.1@osu.edu

Xiaorui Wang

Department of Electrical and Computer Engineering
The Ohio State University
wang.3596@osu.edu

ABSTRACT

A critical quality factor for smartphone apps is responsiveness, which indicates how fast an app reacts to user actions. A *soft hang* occurs when the app's response time of handling a certain user action is longer than a user-perceivable delay. Soft hangs can be caused by normal User Interface (UI) rendering or some blocking operations that should not be conducted on the app's main thread (i.e., *soft hang bugs*). Existing solutions on soft hang bug detection focus mainly on offline app code examination to find previously known blocking operations and then move them off the main thread. Unfortunately, such offline solutions can fail to identify blocking operations that are previously unknown or hidden in libraries.

In this paper, we present Hang Doctor, a runtime methodology that supplements the existing offline algorithms by detecting and diagnosing soft hangs caused by previously unknown blocking operations. Hang Doctor features a two-phase algorithm that first checks response time and performance event counters for detecting possible soft hang bugs with small overheads, and then performs stack trace analysis when diagnosis is necessary. A novel soft hang filter based on correlation analysis is designed to minimize false positives and negatives for high detection performance and low overhead. We have implemented a prototype of Hang Doctor and tested it with the latest releases of 114 real-world apps. Hang Doctor has identified 34 new soft hang bugs that are previously unknown to their developers, among which 62%, so far, have been confirmed by the developers, and 68% are missed by offline algorithms.

CCS CONCEPTS

• **Software and its engineering** → **Dynamic analysis**; **Software performance**; **Operating systems**;

KEYWORDS

Soft Hang Bug, Mobile Apps, Performance Counters

ACM Reference Format:

Marco Brocanelli and Xiaorui Wang. 2018. Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps. In *EuroSys '18: Thirteenth EuroSys Conference 2018, April 23–26, 2018, Porto, Portugal*. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3190508.3190525>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '18, April 23–26, 2018, Porto, Portugal

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5584-1/18/04...\$15.00

<https://doi.org/10.1145/3190508.3190525>

1 INTRODUCTION

There can be a variety of reasons for software to have responsiveness problems and programming issues are among the major ones. Correctness bugs such as deadlocks or infinite loops [26, 27, 53] may cause an app to become unresponsive for an unlimited period of time or until the app is killed. *Soft hang bugs* instead, which is our focus in this paper, are programming issues that may cause the app to have *soft hangs*, i.e., the app becomes unresponsive for a limited but perceivable period of time. A soft hang bug is some blocking operation¹ on the app's main thread that can be executed on a separate worker thread, such that the main thread can become more responsive [44]. For example, a soft hang may occur when the main thread is blocked by some lengthy I/O APIs (e.g., file read and write). Different from server/desktop software, the development of mobile apps is more accessible even to inexperienced developers who can easily have soft hang bugs in the released version of their app. Therefore, it is important to help those smartphone developers detect and diagnose soft hangs in their apps.

Existing studies [30, 44, 50] propose offline detection algorithms that try to find soft hang bugs by searching for calls to well-known blocking APIs on the app's main thread. Unfortunately, offline algorithms can fail for three main reasons. First, the exponential growth of new APIs [46] makes it almost impossible to have full knowledge of their processing time, thus new blocking APIs (i.e., potential soft hang bugs) may be unknown to offline detection algorithms and developers (e.g., *K9-mail* bug #1007 in Table 5). Second, some segments of the app code, e.g., closed-source third-party libraries, may have a soft hang bug but may not be directly accessible. Thus, offline solutions may not be able to analyze the source code of those libraries and may miss soft hang bugs. For example, one out of the three *SageMath* bugs (#84 in Table 5) is caused by a well-known blocking database API hidden within a third-party library. This API can be detected only if the offline algorithm has a chance to examine the library code. Third, a self-developed lengthy operation (e.g., a heavy loop) on the main thread cannot be detected by offline algorithms that try to search for the names of well-known blocking APIs. Some studies [11, 33] optimize loops to improve app performance, but they do not focus on soft hang bugs. As a result, an app may still have bugs that can cause soft hangs at runtime, even after offline detection tools have already been applied.

Given the limitations of offline detection, it is desirable to have a runtime hang detection algorithm that catches a soft hang on the fly

¹We adopt the terminology used in [44] and consider any operation blocking if there exists a worst-case scenario that prevents the calling thread from making progress until timeout (e.g., 100ms perceivable delay [14]).

and finds which blocking operation is causing it, so that the developer can get sufficient diagnosis information to fix the problem. An important challenge for runtime soft hang detection is to diagnose if a soft hang is indeed caused by soft hang bugs, instead of lengthy User Interface (UI) operations that must execute on the main thread. If a UI operation is mistakenly diagnosed as a soft hang bug, we say it is a false positive. Some proposed runtime algorithms for server/desktop software [35, 53] monitor the resource utilization of the software (e.g., CPU time or memory access) and detect potential hangs when static resource utilization thresholds are violated. Unfortunately, those algorithms are mainly designed for correctness bugs rather than soft hang bugs. Correctness bugs, different from soft hang bugs, cause the app to become unresponsive for an unlimited period of time and thus can be detected by monitoring the coarse-grained resource utilization of apps. However, soft hang bugs can last as little as 100ms and need a more fine-grained monitoring of the app execution. As a result, as we show in this paper, resource utilizations used for soft hang bug detection may cause large numbers of false positives and negatives. Some recent studies [36] propose in-lab test case generation to detect a sequence of actions whose execution cost gradually increases with time, but they are not designed to work in the wild to detect soft hang bugs. Some practical tools have been developed for smartphones in the wild. For example, Android OS incorporates an Application Not Responding (ANR) tool [20] to detect hangs that last longer than 5 seconds, which is much longer than the 100ms perceivable delay [14]. Thus, it can miss many soft hangs. However, as shown in Section 2.2, simply reducing the timeout to 100ms, as proposed in [28], would lead to a large number of false positives.

In this paper, we propose Hang Doctor, a runtime soft hang detection and diagnosis methodology that runs in the wild on user devices. Hang Doctor helps developers track the responsiveness performance of their apps and provides diagnosis information for them to fix soft hangs. Hang Doctor is not meant to replace offline detection, which is the primary approach because it can detect known soft hang bugs before the app is released in the wild. Instead, it can supplement offline detection by identifying new blocking APIs that are previously unknown.

Hang Doctor features a two-phase algorithm to achieve high detection performance with small overheads. The first phase is a lightweight soft hang bug symptom checker (S-Checker) that is invoked upon the execution of each user action to label only those actions that have the symptoms of a soft hang bug. We define the symptoms of a soft hang bug by profiling performance event counters during soft hangs. Then, we use correlation analysis, which identifies the performance events that are more suitable for soft hang bug detection. Based on this analysis, we design a soft hang filter that reads the selected performance events and compares them with their thresholds to find soft hang bugs and minimize the numbers of false positives and negatives. Compared to resource utilizations [35, 45, 53], monitoring and accessing performance event counters is more lightweight and provides a wider variety of low-level hardware metrics. Compared to just monitoring the response time [28], using both response time and performance events allows to minimize the number of false positives, thus improving the detection performance. The second phase is a Diagnoser that is invoked only for those labeled actions that have the soft hang bug

symptoms. Diagnoser monitors the response time of an executing action. If its response time exceeds 100ms again, Diagnoser collects stack traces until the end of the soft hang for in-depth diagnosis. Then, Diagnoser analyzes the collected stack traces to determine if there is indeed a soft hang bug. Upon the detection of a soft hang bug, the collected information is reported to the app developer. If it is caused by a previously unknown blocking API, Hang Doctor adds it to the database of known blocking APIs, so that the offline algorithms can detect them.

Hang Doctor addresses the limitations of offline detection solutions because it is a runtime solution that can detect soft hang bugs caused by 1) new blocking APIs, 2) known blocking APIs called in third-party libraries (without the need of source code), and 3) self-developed lengthy operations, as long as those soft hang bugs manifest themselves at runtime. Therefore, with Hang Doctor, developers can track the responsiveness performance of their apps in the wild and get diagnosis information about the soft hang bugs to be fixed.

Specifically, this paper makes three contributions:

- We propose Hang Doctor, a runtime methodology to detect soft hang bugs that can be missed by offline detection algorithms.
- Hang Doctor features a two-phase detection algorithm to achieve small runtime overheads. A novel soft hang filter is designed based on response time and performance event counters for high detection performance. To our best knowledge, this is the first work that leverages performance event counters for soft hang bug detection.
- We have implemented Hang Doctor and tested it with the latest releases of 114 real-world apps. Hang Doctor has found 34 new soft hang bugs previously unknown to their developers. So far, 62% of the bugs have already been confirmed by the developers (see our website for details [3]) and 68% are missed by offline detection algorithms.

The rest of this paper is organized as follows. Section 2 motivates our study and Section 3 describes the design of Hang Doctor. Section 4 evaluates our solution. Section 5 discusses the related work. Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATION

In this section, we first introduce some background information on soft hangs. We then use real-world examples and traces to demonstrate the limitations of existing soft hang bug detection algorithms as our motivation.

2.1 Background

For mobile apps (e.g., Android, iOS, Windows), the only app thread that is designed to receive and execute user actions from the User Interface (UI) is the main thread [14]. Here, we briefly introduce how apps handle user actions and why blocking operations cause soft hangs. Note, in this paper we mainly focus on Android OS for its open-source nature, but similar considerations can be applied to other mobile OSs.

User actions performed through the touchscreen of the smartphone are recognized and forwarded by the OS to the main thread of the foreground app as *input events*. An input event is a message

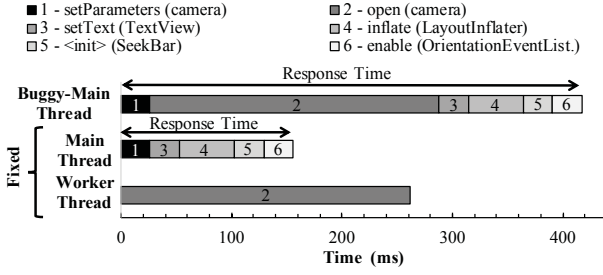


Figure 1: The Main Thread of the app *A Better Camera* [18] executes UI-related APIs (e.g., `setText`, `inflate`, `init`, `enable`) and camera APIs (e.g., `setParameters`, `open`). Moving blocking APIs (e.g., `open`) to a worker thread makes the app more responsive to user actions.

containing some information that allows the main thread to determine what code to execute for that event (e.g., listeners, handlers). New events are put into an event queue upon their arrival and the events are executed, one by one, in their queue order. Therefore, if the code related to an input event includes the execution of a blocking operation on the main thread, a delay longer than 100ms may be perceived by users [20, 30]. These responsiveness bugs are well known in the literature as soft hang bugs [44, 50]. In order to avoid soft hangs, as suggested by the Android guideline [20], blocking operations should be moved to a worker thread, so that the main thread can execute new input events in a timely manner without the user perceiving delay.

Practical Example. Figure 1 shows the sequence of operations executed by an input event on the main thread of the app *A Better Camera* [16, 18] when the user *Resumes* the main activity (labeled *Buggy Main Thread* in figure). The main activity of the app is composed of 1) camera images loaded through two camera APIs (e.g., `setParameters`, `open` [19]) and 2) a UI interface loaded through four UI-APIs (e.g., `setText`, `inflate`, `<init>`, `enable`) to allow the user to interact with the app. The input event has a response time of 423ms, which is perceivable by the user, to execute those operations. As shown in Figure 1, the camera API `open` is the one that takes the longest to execute. This API connects the app’s UI with the camera, thus it may take a long time to establish the connection. The response time shown in Figure 1 can be reduced to the less perceivable 160ms by moving the execution of this API to a worker thread (labeled *Fixed* in Figure 1), so that it can be executed asynchronously and return the necessary data to the main thread (e.g., with the `onPostExecute` method of `AsyncTask`) without affecting the user experience. Note, while the responsiveness can be further improved by also moving other APIs (e.g., `setParameters`), the UI-APIs must be executed on the main thread because they manipulate the UI and may inevitably introduce a perceivable delay. Thus, UI-APIs are not soft hang bugs and should not be reported by Hang Doctor. It is our future work to study the responsiveness of UI-APIs.

Some soft hang bugs may need a more sophisticated fix, because the subsequent operations on the main thread may depend on the data generated by the blocking API. As a result, such a blocking API cannot be simply moved to a worker thread. Note that the focus of

App Name	DroidWall	FrostWire	Ushaidi	WebSMS
Commit #	3e2b654	55427ef	59fbb533d0	1f596fbd29
App Name	cgeo	Seadroid	FBReaderJ	A Better Camera
Commit #	6e4a8d4ba8	5a7531d	0f02d4e923	9f8e3b0

Table 1: Apps with well-known soft hang bugs tested in the motivation study. The commit number refers to the app version that has the bug.

Hang Doctor is to detect and report such a blocking operation as a soft hang bug. It is up to the developer on deciding how to fix it.

2.2 Motivation

In this section, we first discuss the limitations of offline detection algorithms with several real-world examples. We then conduct experiments with some open-source apps and an LG V10 smartphone to test the performance of existing runtime algorithms. We select these apps (summarized in Table 1) from recent studies [30] and from online repositories [16] by searching the keywords “freeze”, “hang”, or “ANR” (Application Not Responding) in the apps change logs.

Offline Soft Hang Detection. Offline detection algorithms [30, 44, 50] find soft hang bugs by scanning the app code to look for *well-known* blocking APIs on the main thread. However, these offline approaches fail to detect those soft hangs caused by APIs that are *not known* as blocking.

There are cases that some APIs were not known as blocking in the past but caused soft hangs. For example, the camera API `open` is available since 2008 but has been marked as blocking only after 2011 [19, 21]. Similarly, other APIs, such as `mediaplayer.prepare`, `bitmap.decode`, and `bluetooth.accept`, are all available since 2009 but have been clearly marked as blocking only after 2012. As a result, any offline algorithm would not have been able to detect the soft hang bugs caused by those APIs before 2011/2012. Thus, soft hangs may occur at runtime even after using those offline algorithms. Our hypothesis is that there can be other blocking APIs that remain unknown to offline tools and can cause soft hangs at runtime. To our knowledge, currently, there is no established way to automatically determine if a certain API is an unknown soft hang bug. As also stated in some related work, e.g., [44], an API becomes a soft hang bug mainly based on expert knowledge, which often entails manually diagnosing performance data and/or stack traces collected in the wild. Therefore, for these unknown blocking APIs, only a runtime detection algorithm would be able to detect the incurred soft hang in a timely manner and report to the developer. For example, the *K9-mail* bug (#1007 in Table 5) that we have found at runtime is caused by an unknown blocking API `clean`, which is missed by a state-of-the-art detection tool [30]. In addition, self-developed lengthy operations and well-known blocking APIs nested into closed-source third-party libraries may also be missed. We discuss various examples of such cases in Section 4.2.

Runtime Detection. The most representative state-of-the-art runtime method used to catch responsiveness problems is the *Timeout-based* (e.g., [12, 20, 28]). It detects a responsiveness problem when the response time of a user action is longer than a timeout. The main thread may execute several input events for each action. Each input event may cause a soft hang. We refer to the user action

App Name	True Positives				False Positives			
	5s	1s	500ms	100ms	5s	1s	500ms	100ms
DroidWall	0	0	0	1	0	0	1	3
FrostWire	0	0	1	1	0	0	0	5
Ushaidi	0	0	0	2	0	0	1	4
SeaDroid	0	1	1	1	0	0	2	6
WebSMS	0	0	0	1	0	0	0	3
cgeo	0	0	0	5	0	0	2	5
FBReaderJ	0	0	0	6	0	0	2	4
A Better Camera	0	0	0	2	0	0	0	4
TOTAL	0/19	1/19	2/19	19/19	0	0	8	33

Table 2: The timeout value influences the performance of Timeout-based runtime detection algorithms. The numbers report the average numbers of true positives and false positives detected for the apps in Table 1.

response time as the maximum response time of the input events executed.

The choice of the timeout value determines the detection quality of the Timeout-based method. As Table 2 shows, a long timeout (e.g., 5 seconds used by Android’s ANR tool [20]) misses most of the soft hang bugs. A shorter timeout (e.g., 100ms) leads to many false positives caused by UI operations. As we show in Section 4.5, collecting stack traces for every soft hang longer than 100ms may lead to an unnecessarily high overhead. Thus, Timeout-based methods alone are not sufficient for soft hang bug detection. Hang Doctor achieves better detection performance and lower overhead by using response time and performance event counters.

3 DESIGN OF HANG DOCTOR

In this section, we first describe the goals and challenges of Hang Doctor. We then introduce its two-phase algorithm at high level. Finally, we discuss the details of each phase.

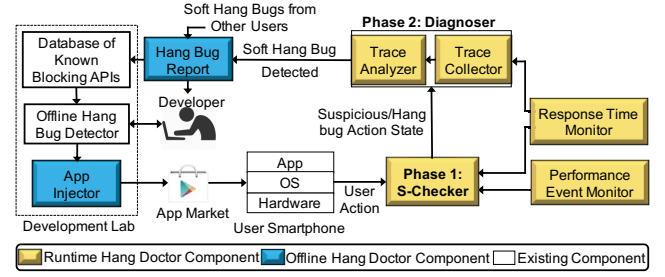
3.1 Goals and Challenges

The target of Hang Doctor is to help app developers fix soft hang bugs that can be missed by offline algorithms (e.g., [30]). Hang Doctor runs at runtime on the users’ devices and has three main goals: 1) understand whether an app is affected by soft hang bugs, 2) diagnose which blocking operation causes each soft hang, and 3) update the database of known blocking APIs used by the offline algorithms. Soft hang bugs caused by self-developed lengthy operations are communicated only to the app developer.

There are three major challenges for Hang Doctor:

- (1) **Finding the root cause:** Hang Doctor should be able to detect soft hang bugs caused by APIs previously unknown as blocking or nested within libraries.
- (2) **High detection performance:** Hang Doctor should ensure high-quality detection, i.e., *all* and *only* the manifested soft hang bugs are detected and analyzed.
- (3) **Low overhead:** Analyzing every soft hang could lead to a high overhead due to a large number of false positives.

In order to achieve the goals and meet the challenges described above, Hang Doctor is designed as a two-phase algorithm that is activated for every user action. The first phase is a lightweight soft hang bug symptom checker (i.e., S-Checker) and the second phase is a soft-hang Diagnoser.



(a) High-level architecture

File Name	Subroutine	Blocking Op.	Code Line	Occurrence
AttachedImageDrawable.java	DrawableFromPath	decodeFile	74	75%
Xstl.java	toHtmlString	transform	67	15%
Xstl.java	toHtmlString	newTransformer	64	10%

(b) Hang Bug Report Example

Figure 2: (a) High-level architecture of Hang Doctor. It is designed as a two-phase algorithm that is activated for every user action. The detected soft hang bugs are communicated to the developer through the Hang Bug Report. (b) Example entries of the app *AndStatus* in Hang Bug Report.

3.2 Design Overview

Figure 2(a) shows the high-level architecture design of Hang Doctor. Because soft hang bugs occur only for some user actions, Hang Doctor dynamically transitions each action among several states. Based on the current action state, Hang Doctor performs either a lightweight analysis with the first-phase S-checker or a deep analysis with the second-phase Diagnoser. Hang Doctor has five runtime components (yellow boxes on the right side of Figure 2(a)), i.e., the *Response Time Monitor*, the *Performance Event Monitor*, the first-phase *S-Checker*, and the second-phase *Diagnoser*, which is composed of *Trace Collector* and *Trace Analyzer*. Hang Doctor includes also two offline components (blue boxes in Figure 2(a)), i.e., *Hang Bug Report* and *App Injector*.

S-Checker. The main approach of Hang Doctor to balance performance and overhead, is to first analyze an executing action with the lightweight first-phase S-Checker. Figure 3 shows a state machine that represents how Hang Doctor manages an action’s state over time. Each node is a state and the solid black arrows represent the transition of an action from one state to another. The labels on these arrows specify the Hang Doctor component that causes the transition (in bold) and the condition. There are three possible paths for an action to go through, starting from state *Uncategorized*, which means the action has never caused a soft hang before.

Path A: Upon the execution of an uncategorized action, if the response time of this action is longer than 100ms, the performance event counters are examined by S-Checker. If the performance event values are *low* (see Section 3.3 for more details), the action is determined as a UI operation and transitioned by the S-Checker to the *Normal* state, which means it does not have a soft hang bug.

Paths B and C: If the uncategorized action has the symptoms of a soft hang bug, i.e., response time longer than 100ms and *high* performance event values, the action transitions to the *Suspicious*

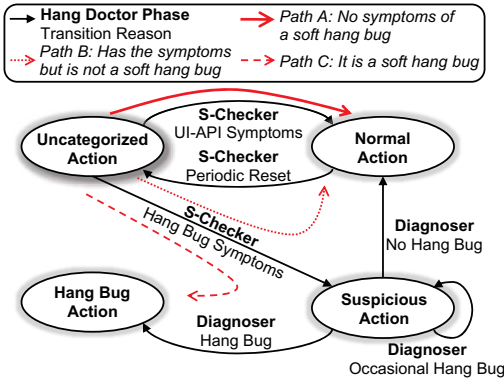


Figure 3: The first-phase S-Checker and the second-phase Diagnoser transition the state of each individual action based on their analysis result to improve the detection performance and lower the overhead. S-Checker monitors the performance event counters and the response time of actions in the *Uncategorized* state to filter out soft hangs caused by UI operations. Diagnoser collects stack traces during the soft hangs caused by actions in the *Suspicious* and *Hang Bug* states to determine the root cause blocking operation.

state. Diagnoser is then triggered to determine (see below) if this action indeed contains a soft hang bug. If not, the action follows Path B to *Normal*. Otherwise, the action transitions to *Hang Bug* through Path C. For those actions in the *Normal* state, to account for soft hang bugs that may manifest after a long time, S-Checker periodically resets them back to *Uncategorized*, so that they can be analyzed again. The period can be configurable (e.g., every 20 executions of the action [36]).

Diagnoser. As Figures 2(a) and 3 show, actions in the *Suspicious* state are analyzed by Diagnoser to determine if the executing action has a soft hang bug. Diagnoser checks if the action currently executing violates the 100ms timeout again and generates a soft hang. If the timeout is not violated (i.e., there is no soft hang), the action may have a soft hang bug that manifests only occasionally because a soft hang was previously detected by S-Checker for this action. In such cases, the Diagnoser leaves the action in the *Suspicious* state, so that it can be traced and analyzed as soon as it causes another soft hang. On the other hand, if the timeout is violated again, Trace Collector collects the main thread’s stack traces until the end of the soft hang, which are then analyzed by Trace Analyzer to determine whether the soft hang is caused by a UI operation or a real soft hang bug. In the former case, Diagnoser transitions the action to *Normal* through Path B in Figure 3. On the other hand, when a soft hang is determined to be a soft hang bug (Path C), Diagnoser transitions the action to the *Hang Bug* state so that it is always analyzed by Diagnoser during future executions. Note, we could avoid collecting other stack traces during soft hangs of actions in the *Hang Bug* state to further reduce the overhead. However, doing so may lead to misdiagnose the root cause of some soft hangs: some actions (e.g., *Andstatus* bug 303, *K9-mail* bug 1007 [3]) may include multiple soft hang bugs that cause soft hangs in different executions.

Developer Feedback and Implementation. Hang Doctor maintains the Hang Bug Report for the developer, which allows to view statistical information about the app responsiveness performance in the wild. It includes a table of detected soft hang bugs ordered by the percentage of occurrences across user devices. Figure 2(b) shows an example of report entries for the three new soft hang bugs of the app *AndStatus* (see Section 4.2). As Figure 2(a) shows, Hang Doctor adds the detected unknown soft hang bugs in the list of known blocking APIs used by offline algorithms, so that also developers of other apps can be warned about the possible new soft hang bugs and fix them before they may cause problems in the wild. We consider Hang Doctor as a supplementary runtime solution to offline algorithms for two main reasons. First, it is desired to detect soft hang bugs offline to avoid poor user ratings and runtime overhead. However, as we have discussed in Section 1, there are unknown soft hang bugs, e.g., *transform* in Figure 2(b), that can be missed by offline solutions, thus a runtime solution is also needed. Second, the user privacy, which is a concern of runtime solutions, is not violated by Hang Doctor because all the anonymized data sent out from the user devices only include those blocking operations that have caused a soft hang. Hang Doctor can be embedded into an app by the developer who wants to improve the app performance. It runs as an additional, separate, and lightweight thread within the app, but it does not need any OS modification to work (see discussion in Section 3.5).

3.3 First Phase: S-Checker

Uncategorized actions are analyzed by S-Checker, which performs a light-weight analysis of their execution to filter out soft hangs caused by UI-APIs. The filtering is based on soft hang bug symptoms, which we define by using correlation analysis of performance event counters with soft hang bugs. We first provide some background information about the performance event counters and the methodology used for the analysis. Second, we explain which app threads to select for the analysis. Third, we examine the results of the correlation analysis and discuss their generality across platforms and training sets. Finally, based on these results, we determine how many performance events are needed to detect soft hang bugs and define the soft hang bug symptoms.

3.3.1 Soft Hang Bug Detection with Performance Event Counters.

Performance Event Counters. Performance event counters can provide low-level information about how well an app is performing. In general, there are two main types of performance event counters: performance events generated and counted at kernel level and performance events generated by the performance monitoring unit (PMU) of the CPU, which are counted using a limited number of special registers. The main advantages of using performance events are the low monitoring overhead and the high customizability, e.g., the user can select which performance events to monitor and the target process or thread. However, using all the available performance events for soft hang bug detection can have two main drawbacks. First, the counting accuracy may decrease because the number of PMU-generated events available is usually much greater than the number of registers (e.g., 37 events vs 6 registers in the LG V10). Second, the soft hang bug detection performance may degrade because some of the available performance events may not

be able to distinguish soft hang bugs from UI operations, which may lead to high numbers of false positives and false negatives. To solve these two problems, we design the S-Checker based on correlation analysis, which identifies few performance events that lead to high soft hang bug detection performance.

Methodology. Here, we briefly describe the methodology used for the correlation analysis. We mainly present the results with the LG V10 smartphone, but we have obtained similar results with other devices (e.g., Nexus 5, Galaxy S3). We collect performance event samples (46 performance events are available in total) to use in the analysis from the apps listed in Table 5. We use 1) a training set of soft hang bugs and UI-APIs to find the performance events and their thresholds that allow to detect soft hang bugs with low numbers of false positives and false negatives and 2) a validation set of soft hang bugs and UI-APIs to demonstrate the efficacy of our solution. For the training set, we have used 10 different well-known hang bugs in Table 5 that are also detected by offline tools and 11 UI-APIs. Note, due to the limited number of different known soft hang bugs that can be analyzed, the training set size is limited. For the validation set (see results in Section 4.4), we have used the previously unknown soft hang bugs in Table 5 that are missed by existing offline solutions. None of the soft hang bugs in the training set is included in the validation set.

To train S-Checker, we sample the performance events during the execution of user actions that have soft hangs caused by the soft hangs bugs and UI-APIs in the training set. For each performance event sample, we perform the correlation analysis by calculating the *Pearson correlation coefficient* [24] between the samples collected during each action and a vector that identifies each sample as soft hang bug or UI operation. Each coefficient ranges from -1 (negative correlation) to 1 (positive correlation): the higher is the correlation coefficient of a performance event, the better it diagnoses the soft hang cause. Note that here we test the linear correlation of performance events with soft hang bugs. We leave as future work studying the non-linear correlation.

Thread Selection. In order to perform the analysis, we first need to choose which app threads to monitor. In general, there are three types of threads for each app. Several background worker threads, a main thread, and a render thread. Background threads are not involved, in most of the cases, in soft hangs and thus they should not be monitored. The main thread, as explained in Section 2.1, is the thread that may have soft hang bugs and thus it should be included in the analysis. It handles user actions and performs some UI update operations. These UI changes are then communicated to the render thread, which performs the heavier job of generating and communicating every frame (e.g., button color change) to the Graphics Processing Unit (GPU). Thus, when there is no soft hang bug, the main thread executes mostly UI-related jobs and generates a lot of work for the render thread. *Intuitively, it may be possible to recognize soft hang bugs when the main thread does not generate much work for the render thread.* Therefore, for each action, we consider two cases for the correlation analysis. First, we monitor all the performance events of both main thread and render thread, i.e., each performance event has one sample that is the difference between the recorded performance event values of main thread and render thread. Second, we monitor only the main thread. Note, we

Performance Event	Corr. Coeff.	Performance Event	Corr. Coeff.
context-switches	0.658	minor-faults	0.601
task-clock	0.632	page-faults	0.601
cpu-clock	0.632	L1-dcache-loads	0.469
page-faults	0.561	L1-dcache-stores	0.454
minor-faults	0.557	instructions	0.451
cpu-migrations	0.548	cache-misses	0.440
cache-misses	0.472	task-clock	0.431
instructions	0.466	cpu-clock	0.431
cache-references	0.466	cache-references	0.428
raw-l1-dcache-refill	0.459	branch-loads	0.416
Average	0.545	Average	0.472

(a) Main Thread - Render Thread

(b) Only Main Thread

Table 3: Correlation analysis results used for the design of S-Checker. Top-10 most correlated performance events for soft hang diagnosis. (a) Monitoring main thread and render thread increases the correlation of about 14% on average compared to (b) monitoring only the main thread.

do not consider the case of only render thread because soft hang bugs are located only in the main thread.

Correlation Analysis. Table 3(a) shows the top-10 most correlated performance events for the first case: 30% of the events have a coefficient higher than 0.6, 30% between 0.5 and 0.6, and 40% between 0.4 and 0.5. Table 3(b) shows the top-10 performance events for the second case: 20% of the coefficients are just above 0.6 while 80% are between 0.4 and 0.5. These results confirm that monitoring both main thread and render thread allows to have better soft hang bug detection performance than monitoring only the main thread. Thus, we use both threads to identify which performance events are necessary to minimize false positives and negatives. On the other hand, S-Checker could be designed based only on the main thread for smartphones running older versions of Android (i.e., below 5.0) that do not have the render thread.

Some events in Table 3(a), e.g., context-switch, task-clock, and page-fault, have a high correlation coefficient because their increase is dictated by OS decisions on thread scheduling rather than the particular source code of a soft hang bug. The context-switch count of a thread increases whenever this thread is executing but is preempted by another thread. The task-clock count of a thread increases to keep track of the CPU time received by this thread. The page-fault count of a thread increases whenever this thread is executing and tries to access a page that is not currently mapped in memory. When there is a soft hang bug, the main thread has heavy work to execute and does not provide much work to the render thread. As a result, during soft hang bugs, the context-switch, task-clock, and page-fault counters are generally high for the main thread and low for the render thread, i.e., the difference between main thread and render thread for each event counter is usually high. During a UI-API, the main thread provides much more (UI-related) work to the render thread, i.e., the difference between main thread and render thread for each event counter is usually low. As a result, these event counters are likely to be useful for the detection of soft hang bugs. Other event counters have a lower correlation coefficient because their increase may depend more on the specific source code of a soft hang bug. For example, the instruction count increases every

Performance Event	Corr. Coeff.	Performance Event	Corr. Coeff.
context-switches	0.707	context-switches	0.817
task-clock	0.678	task-clock	0.778
cpu-clock	0.678	cpu-clock	0.777
page-faults	0.563	page-faults	0.734
minor-faults	0.560	minor-faults	0.733
cache-misses	0.467	raw-l1-dcache-refill	0.548
L1-dcache-stores	0.463	cache-misses	0.540
cpu-migrations	0.457	instructions	0.467
raw-l1-dcache-refill	0.449	raw-l1-itlb-refill	0.464
cache-references	0.443	cache-references	0.461

(a) 75% training set

(b) 50% training set

Table 4: Sensitivity analysis for the correlation analysis of Table 3(a). The correlation analyses with (a) 75% and (b) 50% of the data points used in Table 3(a) have similar results, thus the results do not depend on the training set.

time a thread executes an instruction. Each soft hang bug may have more or less instructions compared to UI-APIs and thus it is more difficult to use those event counters to distinguish soft hang bugs from UI-APIs. As a result, it is unlikely that they can be used for soft hang bug detection. Note, these observations hold for heavy APIs and also self-developed lengthy operations².

Generality of the Analysis. It could be argued that the above analysis may depend on the platform and the training set used. As we have verified by testing various devices (LG V10, Nexus 5, Galaxy S3), the proposed correlation analysis has little to do with the particular platform used because these performance events are mostly related to OS scheduling decisions at the kernel level. In addition, the first six most-correlated event counters in Table 3(a) are generated by the kernel and thus they are available independently from the particular CPU and architecture. The rest of the events in Table 3(a) are generated by the PMU of the CPU but most of them, e.g., cache-misses, instructions, cache-references, are present in most CPUs. This is why different platforms have similar correlation analysis results. Next, we perform a sensitivity analysis to demonstrate that the proposed correlation analysis is not affected by the particular training set.

In order to perform the sensitivity analysis, we change the training set used to execute the correlation analysis. Due to the limited number of known data points, we perform the sensitivity analysis by 1) reducing the size of the training set to generate new training sets, 2) executing the correlation analysis on the new training sets, and 3) comparing the most correlated performance events for each training set. The correlation analysis summarized in Table 3(a) is robust to training set changes if the top-correlated performance events remain the same across all the training sets. We randomly remove data points from the full training set and generate two new training sets that have 75% and 50% of the data points used in the full training set. Tables 4(a) and 4(b) show the correlation analysis with the 75% and 50% training sets, respectively. The top-5 performance events, i.e., the context-switch, task-clock, cpu-clock, page-faults,

and minor-faults, have the same ranking positions in all the training sets, which means that the correlation of these performance events to the soft hang bugs is not affected by the training set used. Note that, with smaller training sets, the correlation coefficients may increase because it is easier to separate hang bugs from UI-APIs. In addition, in some cases, low-correlated performance events may change in ranking position because their correlation may be more dependent on the particular data points in the training set. This sensitivity analysis demonstrates that our correlation analysis is robust to different training sets and that the correlation of the top-5 performance events is not affected by the training set used.

Hang Bug Symptoms and Filter Details. In order to find which performance events are necessary to detect the soft hang bugs, we use the following procedure. First, starting from the most correlated event counter (i.e., *context-switches* in Table 3(a)), we find the best threshold that distinguishes soft hang bugs from UI-APIs by minimizing false positives and false negatives. Second, in case of false negatives, we include another performance event (as ordered in Table 3(a)) until all the soft hang bugs in the training set can be detected by at least one performance event. Note, the primary target of Hang Doctor is to detect soft hang bugs, thus it is important that we minimize or eliminate (if possible) the number of false negatives. The minimization of false positives is a secondary target to reduce the overhead that we address by properly choosing the thresholds. Using this procedure (see next paragraph), we find that just three performance events are necessary: two performance events that track the CPU activity, i.e., *context-switches* and *task-clock*,³ and one that tracks the memory activity, i.e., *page-faults*.

As explained before, a higher event counter difference indicates a higher main thread activity. Figure 4 shows the soft hang samples (HB for soft hang bug and UI-API for UI operations) in descending order for the three event counters. As the figure shows, most of the soft hang bugs have a high performance event difference. For each performance event, we identify the soft hang bug symptoms that distinguish most of the soft hang bugs from most of the UI-APIs:

- *Positive context-switch difference.* As Figure 4(a) shows, 90% of the UI-API samples have a negative *context-switch* difference while 90% of the soft hang bugs have a positive difference.
- *Task-clock difference above 1.7e8.* As Figure 4(b) shows, 80% of the soft hang bug samples have a *task-clock* difference greater than 1.7e8, which is more than twice larger than 80% of the UI-API samples.
- *Page-fault difference above 500.* As Figure 4(c) shows, 90% of the soft hang bug samples have a *page-fault* difference greater than 500, which is more than twice larger than 80% of the UI-API samples.

When an *Uncategorized* user action has a soft hang, S-Checker monitors the above three performance events and reads their accumulated values at the end of the action: if at least one of the above three conditions is verified, S-Checker transitions the action to the *Suspicious* state, so that it can be further diagnosed with the Diagnoser. Otherwise, if none of the conditions are verified, the action is transitioned to the *Normal* state, which does not do any data collection for minimized overhead. If a soft hang does not occur, in order to account for soft hang bugs that may occasionally

²We do not discuss network-related operations because 1) they are well-known hang bugs that can be detected by offline tools and 2) they would generate an exception during the build operation of the app, thus it is unlikely to find them in the wild. Hang Doctor can be easily extended to detect also these bugs by monitoring the network activity of the main thread.

³The *cpu-clock* is omitted because it is similar to the *task-clock*.

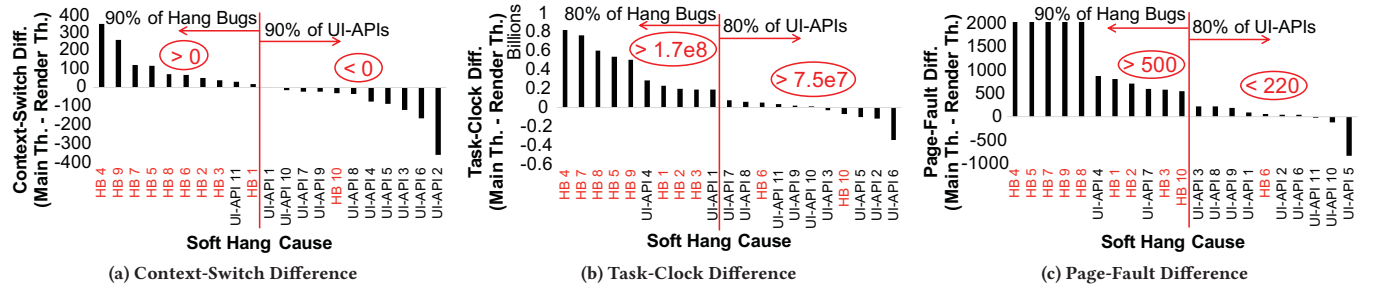


Figure 4: Analysis of three top-correlated performance events in our training set. Using these three performance events allows to distinguish soft hangs caused by soft hang bugs (HB) from those caused by UI operations (i.e., UI-API). Most of the soft hang bugs have a high performance event difference while most of the UI-APIs have a low performance event difference. This is because soft hang bugs, different from UI-APIs, cause more work for the main thread and less work for the render thread.

manifest with soft hangs, the action is left in the *Uncategorized* state so that it is monitored again in future executions. This filter recognizes 100% of the soft hang bugs and prunes 64% of the false positives in the training set (81% overall accuracy).

Automatic Adaptation of the Filter. As explained before, the effect of soft hang bugs on the execution behavior of main thread and render thread is mainly software dependent rather than platform dependent. Thus, as we verify by testing the designed filter with various devices (e.g., LG V10, Nexus 5, Galaxy S3), the selected thresholds and events are generally good also for other platforms. In addition, our validation results in Section 4.4 show that the above conditions ensure good detection performance even with a set of soft hang bugs and UI-APIs not used for the design of S-Checker.

On the other hand, because unfortunately we could not test all the possible existing soft hang bugs and platforms, we cannot completely exclude the possibility that there could be cases of soft hang bugs that need more event counters to be used or a slightly different threshold on a particular device. In order to address this concern, Hang Doctor could automatically adapt the thresholds or even the selected event counters. For example, Hang Doctor could perform a periodic data collection of performance event counters (e.g., top-ten counters in Table 3(a)) and stack traces during the execution of user actions. This data collection would be performed as an extra task for Hang doctor and would be independent from the activities of S-Checker and Diagnoser. The data collection period could be adjusted and set long enough so that this extra data collection overhead can become negligible. Using the collected data, Hang Doctor may verify whether to execute a light adaptation or a heavier adaptation algorithm. The light adaptation is executed on the user device and has a low computational overhead. It is executed if the data collected includes false positives or false negatives that can be eliminated by simply increasing or decreasing, respectively, some of the thresholds of the selected performance event counters. The heavy adaptation, which may lead to a higher computational overhead and thus it could run on a server, is executed if the light adaptation is not sufficient or leads to poor detection performance. The heavy adaptation uses the collected data to execute an automated version of the above described algorithm to select the performance events and their thresholds. The selected new performance event counters

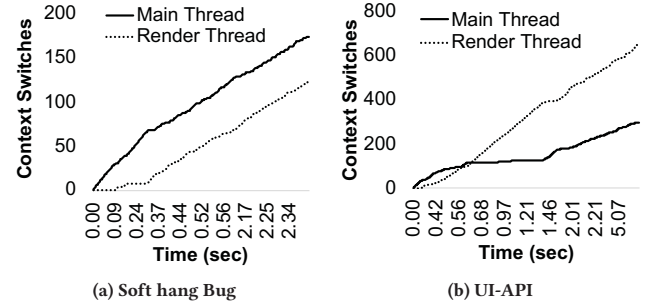


Figure 5: Context-switch traces of main thread and render thread for two actions with a soft hang caused by the (a) soft hang bug 2 and (b) UI-API 2 in Figure 4(a). Using only a few samples collected at the beginning of the action execution may lead to false positives (e.g., from time 0s to 0.6s in (b)).

and their new thresholds could then be sent as upgrades to the device for improved detection performance.

Discussion. In order to minimize the overhead of S-Checker, we could run the above filter based only on a few performance event samples collected at the beginning of an action execution. Unfortunately, this strategy may lead to many false positives. Figures 5(a) and 5(b) show the context-switch count of two actions that lead to the soft hang bug number 2 and the UI-API number 2 in Figure 4(a). While the action with the soft hang bug shows soft hang bug symptoms during the whole execution, i.e., positive difference, the UI-API in Figure 5(b) has soft hang bug symptoms between time 0s to 0.6s, even though the soft hang is caused by a UI operation (similar results for most of the other UI-APIs and performance events). This behavior is common at the beginning of an action execution because the main thread has to execute some developer-defined code (e.g., in the *onClick* method for buttons) and some UI-related operations (e.g., calculate UI elements positions) before doing any UI changes that involve the render thread. Therefore, the above soft hang bug symptoms may not work through the whole action execution. As a result, S-Checker conservatively counts the performance events until the end of the action execution,

i.e., none of the two threads execute or a new action is detected. Then, it checks the above conditions.

3.4 Second Phase: Diagnoser

Suspicious actions are analyzed by the Diagnoser, which performs a deep analysis of their execution to determine the root cause blocking operations that cause soft hangs.

3.4.1 Trace Collection and Analysis. During a user action execution, the main thread may execute several input events in sequence, which are analyzed by Hang Doctor according to the action's current state. If any of these input events has a response time longer than the minimum human-perceivable delay (i.e., 100ms), Diagnoser starts collecting stack traces of the main thread until the end of the soft hang to find the root cause blocking operation (i.e., Diagnoser does not monitor performance events). A stack trace shows which operation and code line a thread is executing at a certain time point. Therefore, by collecting stack traces during a soft hang, we can understand which operations the main thread executes over time. In particular, an operation that executes for long time and causes a soft hang appears in most of the collected stack traces. For example, in Figure 1, the *camera.open* API, which is the root cause of the soft hang, appears in about 60% of the stack traces collected during the soft hang. Different from other tracking methods, e.g., code injection to log when certain operations are executed [38], this technique allows to track the execution of any operations executed on the main thread of the app, even those blocking APIs nested in third-party libraries. Stack traces are also useful in detecting soft hangs caused by self-developed lengthy operations, such as heavy loops, because they include which file, subroutine, and code line number in the app code has the heavy loop. At the end of the soft hang, Trace Analyzer analyzes the stack traces collected to find the root cause blocking operation.

Trace Analyzer determines the root cause of a soft hang by analyzing the *occurrence factor* of the API that appears the most across the stack traces collected. The occurrence factor is defined as the percentage of stack traces that include a certain API. If the occurrence factor is high (the exact threshold can be adjusted), as the example in Figure 1, the probable cause of the soft hang is a single heavy API (e.g., *camera.open*). However, there could be cases of self-developed operations executing many light APIs that cause a soft hang. In these situations, moving just one of these APIs would likely not fix the soft hang. In order to fix this type of soft hang, the whole self-developed operation should be moved to a background thread. Trace Analyzer recognizes these cases when the occurrence factor is low: first it finds what is the most common caller function (i.e., the self-developed operation that executes those APIs) across the stack traces collected that has a high occurrence factor and then indicates this caller function as the probable cause of the soft hang.

Next, Trace Analyzer determines if the root cause is a soft hang bug or a UI-API. To our best knowledge, any operation that does not involve the UI can be moved off the main thread to improve the app responsiveness. This analysis can be automated because UI-APIs are well known as they are grouped in a few classes (e.g., View and Widget Classes [21]) and thus they can be easily recognized by analyzing the stack traces. Trace Analyzer can recognize even new UI-APIs from their class name because, different from new soft

hang bugs, new UI-APIs are expected to be part of those classes. Note, self-developed lengthy operations are reported as possible soft hang bugs to the app developer if the collected stack traces do not include only UI-APIs.

3.5 Hang doctor Implementation

Android apps handle user actions by implementing special listeners, handlers, and callback functions such as *onClick* when buttons are clicked, *onScroll* when the user scrolls lists of items, and so on. To distinguish the various actions, App Injector assigns a Unique ID (UID) to every action. Then, at runtime, a look-up table is created to save various information about the actions, including UIDs and current states. When the user executes an action, Hang Doctor reads the UID and looks up the current state of that action to eventually activate S-Checker or Diagnoser.

Hang Doctor measures the response time of each input event executed on the main thread by exploiting the *setMessageLogging* API of Android's *Looper* class, which is invoked in two cases: 1) when an input event is dequeued for execution and 2) when this input event finishes the execution. As a result, the response time is measured as the difference between these two invocations. The performance events are accessed and monitored using *Simpleperf* [22], which is an executable that accepts a wide range of parameters to customize which threads and which performance events to collect. Currently, Hang Doctor exploits this executable to start and stop the monitoring of performance events during a user action. SimplePerf can be easily included with the app as an additional lightweight executable (i.e., less than 1% of extra space) or directly integrated into Hang Doctor's source code.

We integrate Hang Doctor in the app so that developers do not need any OS modification to track the responsiveness performance of their apps. However, the methodology of Hang Doctor could be generalized and integrated into the OS as a more general framework that improves the currently used ANR tool [20]. We plan to do this in our future work.

4 EVALUATION

In this section, we first introduce our baseline and evaluation metrics. Second, we summarize all the real-world soft hang bugs detected by Hang Doctor. Third, we give a concrete example on how Hang Doctor works. Fourth, we evaluate its detection performance and overhead. Finally, we discuss alternative design approaches and the current limitations of Hang Doctor.

4.1 Baselines and Performance Metrics

Baselines. We consider three baselines for comparison:

- (1) **Timeout-based (TI)** detects a potential soft hang bug when the action's response time becomes longer than the human-perceivable delay of 100ms. TI is similar to solutions adopted in Android OS [20] and proposed by various studies such as Jovic et al. [28].
- (2) **Utilization-based (UT)** monitors periodically (every 100ms) the resource utilizations of the main thread (e.g., CPU time, memory traffic, network usage). It detects a potential soft hang bug when at least one of the resource utilizations is above its static utilization threshold. UT is similar to the

solutions proposed by various studies such as Pelleg et al. [35] and Zhu et al. [53].

- (3) **UT+TI** is a simple combination of Utilization-based and Timeout-based but collects resource utilizations only during soft hangs. UT+TI detects potential soft hang bugs when 1) the response time becomes longer than 100ms and 2) at least one of the resource utilizations monitored is above its static threshold. Different from Hang Doctor, it uses coarse-grain resource utilizations rather than low-level performance events to diagnose soft hang bugs.

Like Hang Doctor, all the baselines collect stack traces when they detect a potential soft hang bug to find out the causing operation. For the baselines that use the resource utilizations, to test the impacts of different thresholds, we consider two possible thresholds for each app: 1) a low threshold (i.e., UTL, UTL+TI), which is the minimum resource utilization observed during soft hang bugs, 2) a high threshold (i.e., UTH, UTH+TI), which is set as the 90% peak value of the resource usage observed during soft hang bugs. In addition, we compare the detection performance of Hang Doctor with *PerfChecker* [30], which is the state-of-the-art offline detection tool. We do not test a phase-2-only baseline because it would be similar to the Timeout baseline, thus we omit it.

Performance Metrics. In order to compare the detection performance of the baselines with Hang Doctor, we manually review the source code of apps to find *well-known* soft hang bugs (e.g. database operations). Then, for each baseline, we count the *true positives*, i.e., real soft hang bugs that are also detected by the baseline, *false positives*, i.e., bugs detected by the baseline that are not real soft hang bugs, and *false negatives*, i.e., real soft hang bugs that are missed by the algorithm. When an *unknown* bug is detected by any of the baselines, we revise the app code to determine whether it is a true positive, i.e., we fix the bug and verify that the app does not have any more soft hangs, or a false positive.

A problem we have is how to count the false negatives for the unknown soft hang bugs. For those new bugs that manifest with a soft hang, we can use the baseline TI because it reports all the stack traces collected during all the soft hang occurrences. Therefore, in order to count the false negatives in such cases, we run Hang Doctor (or a baseline) and TI at the same time to get two separate detection traces. After manually reviewing each trace as described above, we compare these two traces to count the numbers of false negatives for Hang Doctor (or for a baseline). On the other hand, some unknown soft hang bugs may never manifest with a soft hang during our experiments. Unfortunately, there is no manageable way to find and study these *hidden* unknown bugs and count them as false negatives. However, in this study we consider a wide variety of soft hang bugs and thus we believe that Hang Doctor, whenever those hidden bugs actually manifest, would be able to correctly diagnose them.

4.2 Result Summary and Developers' Response

App Selection and Testing. The apps used in our tests are all open-source and available in the Google Play Store [18] and in the GitHub [16]. We have started testing those apps that are still receiving regular updates from the developers, have high counts of downloads, and ensure a large variety of categories (e.g., Social,

App Name (# Downloads)	Commit #	Category	Issue ID	BD (MO)
AndStatus (1K+)	49ef41c	Social	303	3 (2)
DashClock (1M+)	7e248f7	Personalization	874	1 (0)
CycleStreets (50K+)	2d8d550	Travel & Local	117	4 (3)
K9-mail (5M+)	ac131a2	Communication	1007	2 (2)
Omni-Notes (50K+)	8ffde3a	Productivity	253	3 (3)
OwnTracks (1K+)	1514d4a	Travel & Local	303	1 (0)
QKSMS (100K+)	2a80947	Communication	382	3 (3)
StickerCamera (5K+)	6fc41b1	Photography	29	3 (0)
AntennaPod (100K+)	c3808e2	Media & Video	1921	3 (2)
Merchant (10K+)	c87d69a	Business	17	1 (1)
UOITDC Booking (100+)	5d18c26	Tools	3	2 (2)
Sage Math (10K+)	3198106	Education	84	3 (2)
RadioDroid (10+)	0108e8b	Music & Audio	29	2 (1)
Git@OSC (10K+)	bb80ea95	Tools	89	1 (1)
Lens-Launcher (100K+)	e41e6c6	Personalization	15	1 (0)
SkyTube (5K+)	3da671c	Video Players	88	1 (1)
Total				34 (23)

Table 5: Apps tested with Hang Doctor that have shown soft hang problems (114 apps tested in total). The “Commit Number” refers to the master version at the time of our experiments. BD is the number of Bugs Detected by Hang Doctor and MO shows how many of them are Missed by a state-of-the-art Offline detection tool [30]. The web-links to the “Issue IDs” and more details are available on our website [3].

Productivity). Using these criteria, we have tested about 114 apps so far and asked 20 users to test them with Hang Doctor on their own devices for 60 days. Due to space limitations, we summarize only those tested apps that have shown soft hang problems in Table 5. More details about all the tested apps are available on our Hang Doctor website [3]. The reported commit number is the latest version of each app at the time of the tests.

The apps in Table 5 represent typical usage cases of smartphone users: for example, AndStatus is used to scroll social posts in a timeline and is similar to more widespread apps such as Facebook or Twitter; K9-mail is an email client similar to Outlook or Gmail; AntennaPod is used to listen podcasts and is similar to the more popular Podcast Player. In general, the likelihood to find soft hang bugs (or any other type of software bugs) in apps that are not well tested is higher than well-tested and experienced apps. Among the apps in Table 5, 50% have less than 10,000 downloads, 37% have between 50,000 and 100,000 downloads, and 13% have more than 1,000,000 downloads. Thus, the majority of the apps where we have found soft hang bugs are not well-tested. In fact, Hang Doctor can be a powerful tool for the inexperienced developers of apps that need more testing and improvements, so that they can have higher chances of success.

As summarized in Table 5, Hang Doctor has identified 34 new soft hang bugs that were previously unknown to their developers: 68% of the soft hang bugs found by Hang Doctor are missed by *PerfChecker*, i.e., the state-of-the-art offline detection algorithm [30], because the root causes were new unknown blocking APIs. In addition, all the known soft hang bugs detected by *PerfChecker* that manifest with a soft hang are diagnosed by Hang Doctor (see discussion in Section 4.6 for the bugs that did not cause soft hangs). When Hang Doctor detected a soft hang bug, we opened an issue with the app’s developers on GitHub (*Issue ID* in Table 5). For all those issues that have received a reply (62% of the detected soft hang

bugs), the developers have confirmed the problem and, in many cases, have fixed it with a new release of the app (e.g., *AndStatus* bug 303, *K9-mail* bug 1007). Some of the opened issues (38% of the cases) did not receive feedback from the developers (e.g., *Cyclestreets* bug 117). However, we verify the correctness of the detected soft hang bug by fixing it ourselves and testing the modified app. In all the cases, the modified app did not show any more soft hangs.

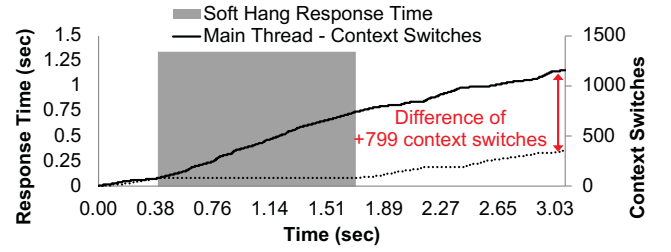
Detected New Blocking APIs. Hang Doctor supplements existing offline detection tools by identifying APIs that are not known as blocking (68% of the cases). These soft hang bugs can be missed by the offline algorithms and thus may cause soft hangs at runtime. For example, one of the soft hang bugs detected in *K9-mail* manifests for an API named *clean* from a third-party library named *org.HtmlCleaner*. This API is used to parse HTML content when some emails are opened by users. For particularly heavy pages, this operation causes the app to have a response time longer than 1.3s. Two of the three detected soft hang bugs in *Sage Math* manifest for an API called *toJson* from the library *com.google.gson*, which is used to serialize a specified object into its equivalent Json (JavaScript Object Notation) representation. The serialization lasts about one second for particularly large objects. All these previously unknown blocking APIs detected with Hang Doctor can be added to the database of known blocking APIs, so that they can improve the detection performance of offline algorithms.

Other Detected Bugs. In a few other cases, i.e., 11 out of 34 (32% of the cases), the soft hang bugs detected by Hang Doctor are caused by well-known blocking APIs as *bitmap decode* or *database operations*, thus they can be detected by the offline algorithm. However, Hang Doctor can be useful also in these cases in two ways. First, some developers may simply choose to ignore soft hang bugs detected offline because they underestimate their impact at runtime. For example, the developer of *AndStatus* (issue 303) has initially argued that the blocking API *BitmapFactory.decodeFile* would not cause many problems since it would be rarely executed. However, Hang Doctor has reported that this blocking API has frequently caused soft hangs of 600ms every time the timeline of *AndStatus* is scrolled. As a result, the developer has promptly fixed the issue and released a more responsive version of the app. Second, in 3 out of these 11 cases (*Owntracks*, *Sage Math*, *Lens-Launcher*), the call to the well-known blocking API is nested within a library API used on the main thread. For example, one of the three soft hang bugs detected in *Sage Math* has a call to the API *get* from a third-party library named *cupboard*, which is not known as blocking. However, this library API hides the execution of a database operation (*insertWithOnConflict*). As discussed in Section 1, the source code of some libraries may be unavailable or encrypted, and thus soft hang bugs could be missed by offline tools. By detecting soft hang bugs while they occur at runtime, Hang Doctor is able to detect the root causes of any soft hangs, even when the bug is nested within a library API whose code cannot be analyzed offline.

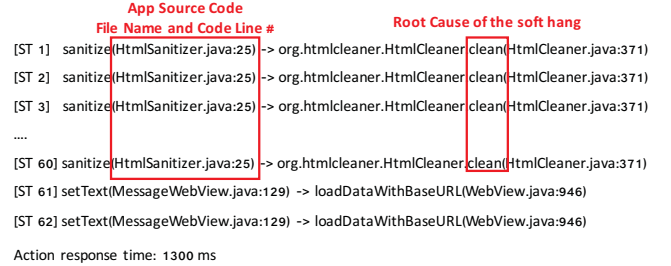
These results confirm that Hang Doctor can effectively help developers improve the responsiveness of their apps.

4.3 Example Runtime Hang Bug Detection

In this section, we show how Hang Doctor detects soft hang bugs with an example app: *K9-Mail*. First, we focus on a particular user



(a) S-Checker detects a possible soft hang bug.



(b) Diagnoser collects Stack Traces (ST) for a deeper analysis.

Figure 6: (a) Execution trace of a user action with *K9-mail*. One of the input events related to the action has a soft hang (shadowed area). S-Checker, at the end of the action execution (i.e., at time 3.1s), finds a positive context-switch difference, i.e., there may be a soft hang bug. (b) At the next execution of the same action, Diagnoser collects the Stack Traces (ST) during the soft hang to find the root cause operation: *clean* API, code line 25 of *HtmlSanitizer.java*.

action to explain how Hang Doctor finds the root cause of a soft hang. Then, we show how Hang Doctor changes the action state when multiple actions are executed.

Finding Root Cause of the Soft Hang. Figure 6 shows how Hang Doctor detects a soft hang bug. Specifically, Figure 6(a) shows the activity of S-Checker for the user action *Open email*. It shows 1) a shadowed area that highlights the response time and time period when an input event of this action has a soft hang and 2) the context-switch of main thread and render thread during the action execution. Note, the other two performance events collected are not shown because they are less meaningful for this specific case.

The user action *Open Email* has never caused a soft hang before, thus it has an initial state of *Uncategorized* and is analyzed by S-Checker. One of the input events executed for this action has a response time of 1.3s (i.e., from time 0.45s to 1.75s), which is much longer than the 100ms human-perceivable delay. At the end of the action execution (i.e., at time 3.1s), S-Checker reads the performance event counters and finds a positive context-switch difference. Thus, S-Checker determines that there could be a potential soft hang bug and transitions that action to *Suspicious* for further diagnosis. When this action causes another soft hang (similar to the soft hang shown in Figure 6(a)), Diagnoser collects stack traces during the soft hang manifestation. Figure 6(b) shows an extract of the collected stack traces. Diagnoser examines them and determines 1) the root cause API, 2) the file name and the code line in the app source code

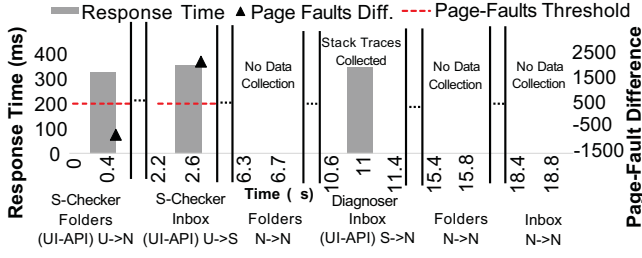


Figure 7: S-Checker and Diagnoser use action states (*U* for *Uncategorized*, *S* for *Suspicious*, *H* for *Hang bug*, *N* for *Normal*) to minimize the overhead of collecting stack traces for soft hangs caused by UI-APIs.

containing the bug. The API *clean* has a high occurrence factor (i.e., 96%, see Section 3.4.1) and is not a UI-API, thus it is determined to be a soft hang bug. As a result, Diagnoser transitions the action to the *Hang Bug* state.

Action State Transitioning. Hang Doctor transitions actions to several states to minimize the stack trace collection overhead during soft hangs caused by UI operations. Figure 7 shows an example trace of *K9-mail* with two different actions (Folders and Inbox) that lead to soft hangs caused by UI-APIs, i.e., they are not soft hang bugs. The figure shows the response time of the actions (shadowed areas), and, when collected, the page-fault difference between the main thread and render thread (the other two event counters are less meaningful for this specific case, thus we do not show them). The bottom of the figure describes 1) Hang Doctor’s component examining each action execution, 2) the action name, 3) the root cause of the soft hang (e.g., UI-API) and the action state update decision (*U* for *Uncategorized*, *S* for *Suspicious*, *H* for *Hang bug*, *N* for *Normal*).

As Figure 7 shows, when the user opens for the first time the *Folders* menu at time 0.2s, a soft hang of 305ms occurs. However, S-Checker finds that the page-fault difference is negative, which is lower than its threshold (dashed red line) and correctly determines that this soft hang is caused by a UI-API. As a result, it transitions the action to the *Normal* state so that Hang Doctor does not check this action in future executions, e.g., 6.3s and 15.4s in Figure 7. In contrast, when the user opens the *Inbox* at 2.3s, S-Checker finds a soft hang of 350ms and a page-fault difference above the threshold, i.e., it is a false positive. Thus, S-Checker transitions this action to the *Suspicious* state for a deeper diagnosis. When the user executes again this action at 10.7s, Diagnoser collects the stack traces and finds that the root cause is indeed a UI-API. As a result, Diagnoser transitions this action to *Normal*, so that it does not cause unnecessarily high overhead for stack trace collection in future executions, e.g., at 18.4s (see Section 4.5 for more results).

4.4 Detection Performance Comparison

Ideally, to ensure best detection performance and lowest overhead, all and only the soft hangs with soft hang bugs are traced with stack traces. Thus, here we count true positives, false positives, and false negatives by counting the soft hangs caused by soft hang bugs and UI operations that are actually traced. Note, based on the apps

App Name	New Bugs	# of Bugs Detected with		
		Context-Switches	Task-Clock	Page-Faults
AndStatus	2	1	-	1
CycleStreets	3	3	-	-
K9-Mail	2	2	2	2
Omni-Notes	3	-	-	3
QKSMS	3	3	3	-
AntennaPod	2	2	2	-
Merchant	1	1	-	-
UOITDC Booking	2	2	2	2
SageMath	2	2	2	2
RadioDroid	1	-	-	1
GIT@OSC	1	1	-	-
SkyTube	1	1	1	1
Total	23	18	12	12

Table 6: S-Checker uses three performance events, i.e., context-switches, task-clock, and page-faults, to find soft hang bugs. The 23 *New Bugs* are those from Table 5 that were previously unknown to be soft hang bugs, i.e., missed offline. All the new soft hang bugs are correctly recognized by at least one of the three event counters.

in Table 5, in Section 3.3.1 we have designed Hang Doctor with a training set, which includes only the well-known soft hang bugs that are *not missed offline*. Here, we test Hang Doctor using the validation set, which includes a different set of soft hang bugs that are *missed offline*. First, we study how S-Checker detects the new soft hang bugs. Then, we compare the detection performance of Hang Doctor with the baselines.

Table 6 lists all the soft hang bugs in the validation set. Hang Doctor monitors performance events to detect previously unknown soft hang bugs. For each app, we report how many bugs in the validation set are detected with each one of the three performance events monitored, i.e., context-switches, task-clock, and page-faults. As Table 6 shows, Hang Doctor correctly recognizes all the 23 unknown soft hang bugs. In particular, 18 bugs out of 23 are recognized with the context-switch counter, and 12 out of 23 with the task-clock and page-fault counters. Thus, similar to the results observed in Section 3.3.1, the context-switch counter is the most correlated with the soft hang bugs. However, using only this event counter would miss 5 new soft hang bugs in these tests, i.e., 1 bug in *AndStatus*, 3 in *Omni-Notes*, and 1 in *RadioDroid*, which are detected with the page-fault counter. These results, demonstrate 1) the effectiveness of Hang Doctor in recognizing soft hang bugs not included in the training set and 2) the importance of using several performance event counters in S-Checker.

Figures 8(a) and 8(b) summarize the comparison with the baselines of true positives and false positives, respectively. The results are normalized to the TI baseline, which collects stack traces for all the soft hangs, thus it does not have false negatives (see Section 4.6 for a discussion about the false negatives that have never manifested with a soft hang). In order to ensure a fair comparison, we use the same app user traces to test Hang Doctor and the baselines. Due to space limitation, we report only the results of some representative apps. Similar results are obtained with the rest of the apps listed in Table 5. *CycleStreets*, as we verify by comparing the number of true positives for all the apps in Figure 8(a), has the lowest number of true positives. This is because this app includes map loading

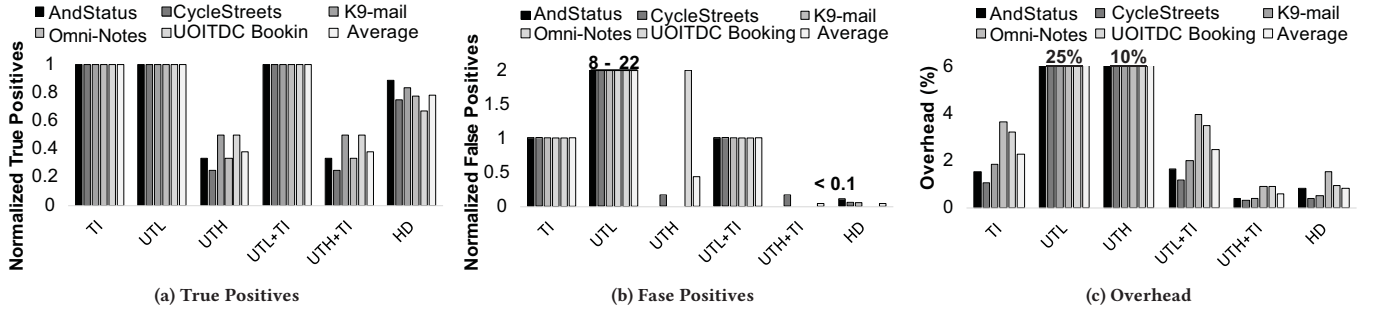


Figure 8: Detection performance normalized to the Timeout-based (TI) baseline, which does not have false negatives. Hang Doctor (HD), different from the baselines, (a) traces most of the real soft hang bugs every time they manifest (the few false negatives are only due to the initial filtering activity of S-Checker) while (b) pruning most of the false positives. As a result, (c) Hang Doctor achieves low overheads, while having high detection performance at the same time.

operations that may cause high resource utilization on the main thread. Thus, the UT baselines may not be able to distinguish soft hang bugs from UI operations. As Figure 8(a) shows, Hang Doctor increases the true positives by relaying on performance events, e.g., 66% more true positives compared to UTH+TI. On average, across the various apps, Hang Doctor traces 80% of the true positive soft hangs and, as Figure 8(b) shows, less than 10% of the false positive soft hangs. The false negatives for Hang Doctor are due to the initial filtering activity of S-Checker. However, all the soft hang bugs are correctly traced with Diagnoser in the subsequent executions of those actions. None of the other baselines achieve a high true positive count and a low false positive count at the same time for all the apps. For example, UTL detects all the soft hang bugs but traces from 8 to 22 times more false positives compared to TI. UTH, has a near zero false positive count but misses 62% of the soft hang bugs. The combinations UTL+TI and UTH+TI achieve a lower false positive count compared to UTL and UTH, respectively. However, they cannot achieve the high detection performance of Hang Doctor because they do not use performance events and do not transition actions across states to lower the false positives.

4.5 Overhead Analysis

Here, we compare the resource usage overhead of Hang Doctor with that of the baselines. Specifically, for each trace, we measure the CPU and memory access (from the *stat* and *io* files, respectively, available in the *proc/PID* filesystem) before and after the execution of a trace without Hang Doctor (or a baseline). Then, we repeat the measurements when Hang Doctor (or the baseline) executes and calculate the percentages of CPU and memory increase. The resource usage overhead is calculated as the average between the percentage CPU overhead and the percentage memory overhead.

Figure 8(c) shows the overhead comparison between the baselines and Hang Doctor. UTL and UTH have about 25% and 10% overhead on average, respectively, because they need to periodically sample the resource utilizations. In addition, UTL frequently collects stack traces because it has many more false positives than UTH, which further increases the overhead. TI instead, on average, has more false positives than UTH but collects stack traces only when the app has a response time longer than 100ms without need

of periodically measuring the resource utilizations. Thus, TI has a lower overhead, i.e., 2.26% on average. UTH+TI is the algorithm with the lowest overhead (about 0.58%) but, as discussed in Section 4.4, it misses most of the soft hang bugs. Hang Doctor has a high detection performance and a 0.83% overhead, which is slightly higher than that of UTH+TI, but 63% lower than that of TI. These results demonstrate the efficiency of collecting performance-event data rather than resource utilizations to prune false positives and reduce the overall overhead. Hang Doctor has also a negligible impact on apps' code size, energy consumption, and responsiveness.

4.6 Alternative Approaches and Limitations

Hang Doctor finds soft hang bugs in the wild while users interact with the apps. An alternative approach would be to run Hang Doctor on a test bed of smartphones where user inputs are automatically generated by tools such as Android's Monkey and MonkeyRunner. The main advantage of this approach is that soft hang bugs could be detected before they cause problems on user devices. In addition, in a test bed environment, smartphones can be easily connected to external power, thus the overhead of Hang Doctor would not be an important concern. As a result, the second phase of Hang Doctor may be sufficient in a test bed because Trace Analyzer can discard most of the false positives by reading the stack traces collected during all the soft hangs. However, note that such test beds often cannot completely recreate the real environment of apps in the wild, which may cause some soft hang bugs to never manifest. As a result, soft hang bugs could still be missed in the test bed and Hang Doctor would still need to run in the wild.

Hang Doctor has four possible limitations.

First, under special conditions, e.g., a soft hang bug within an action that has some heavy render thread operations, none of the conditions described in Section 3.3.1 may be verified, which leads to possible false negatives. However, in our experiments, we have not yet encountered such cases. We plan to address this issue in our future work.

Second, some soft hang bugs may never manifest at runtime with a soft hang. Due to its runtime detection nature, Hang Doctor will miss these soft hang bugs. However, the user would also not experience any responsiveness problems in such cases. Thus, we

can consider these missed bugs as benign false negatives. Note that the false negatives due to unknown soft hang bugs are challenging to identify if they never cause a soft hang. We plan to address this issue in our future work.

Third, Hang Doctor may miss occasional hang bugs in user actions that have previously caused a false positive and thus are in the *Normal* state. Although in our experiments all the known occasional soft hang bugs were diagnosed as soon as they manifest with a soft hang, in order to handle such situations, Hang Doctor periodically resets *Normal* events to *Uncategorized*, so that they can be analyzed again.

Fourth, the training set size used for the correlation and sensitivity analyses in Section 3.3.1 is limited due to the limited number of known soft hang bugs. We plan to repeat the analysis with a larger training set when more soft hang bugs are reported.

5 RELATED WORK

Recent research has proposed a variety of strategies to improve apps' performance [37, 39, 40, 43]. For example, some studies [6, 17, 49] propose to offload the computation-intensive tasks of an app to the cloud. A few other studies help developers improve their apps' performance by identifying the critical path in user transactions [38, 52] or by estimating apps' execution time for given inputs [29]. Different from these studies, we focus on detecting soft hang bugs.

Offline Detection. A widely adopted approach to diagnosing programming issues in software is the offline analysis of source code. For example, many studies [11, 25, 33, 34, 48] focus on helping developers find the app performance bottleneck (e.g., inefficient loops). Huang et al. [23] propose to help developers identify programming issues across different app commits. The most closely related work is offline soft hang bug detection [30, 44, 50], which proposes offline algorithms to automatically detect soft hang bugs by searching the app code for well-known blocking APIs. In contrast, Hang Doctor detect and diagnoses soft hangs at runtime in order to address the limitations of offline detection discussed in Section 1.

Runtime Detection. A variety of runtime approaches have also been proposed to address responsiveness problems. Many proposed runtime algorithms for server/desktop software [8–10, 13, 15, 41, 42, 47] are not suitable for smartphone apps mainly because of their relatively high overheads. Some studies [35, 53] profile various resource utilizations (e.g., CPU time, Memory Access) during bug-free runs of the application and use static thresholds to detect responsiveness problems caused by correctness bugs. Other solutions detect software failure due to concurrency bugs [2] or diagnose synchronization bugs [1]. Different from these approaches, Hang Doctor is designed to detect a different type of bug, i.e., soft hang bugs, on smartphone apps.

Some research [4, 5, 7, 28, 32, 51], similar to the ANR detection tool of Android [20], detects soft hangs in software by monitoring the response time of user actions. The main limitation of these timeout-based approaches is that they can lead to large numbers of false positives and negatives. Pradel et al. [36] propose in-lab test case generation to detect a sequence of actions whose execution cost gradually increases with time, but this solution is not designed to work in the wild to detect soft hang bugs. In addition to their offline

solutions, Wang et al. [44] propose to allow users to force-terminate the currently executing job during a soft hang, but, different from Hang Doctor, they do not diagnose its root cause.

Another important feature of Hang Doctor is its two-phase algorithm that balances detection performance and overheads. Some proposed approaches [12, 31] also attempt to balance monitoring performance and logging overhead in the wild. However, different from Hang Doctor, they either do not detect soft hang bugs [31] or perform only timeout-based detection, without pinpointing the exact blocking operation that causes the soft hang [12].

6 CONCLUSIONS

In this paper, we have presented Hang Doctor, a runtime methodology that supplements the existing offline algorithms by detecting and diagnosing soft hangs caused by previously unknown blocking operations. Hang Doctor features a two-phase algorithm that first checks response time and performance event counters for detecting possible soft hang bugs with small overheads, and then performs stack trace analysis when diagnosis is necessary. A novel soft hang filter based on correlation analysis is designed to minimize false positives and negatives for high detection performance and low overhead. Our results have shown that Hang Doctor has identified 34 new soft hang bugs that were previously unknown to their developers, among which 62%, so far, have already been confirmed by the developers, and 68% are missed by offline detection algorithms.

ACKNOWLEDGMENTS

We thank all the anonymous EuroSys reviewers for their detailed feedback. We would also like to thank our shepherd Dr. Cristiano Giuffrida for helping us shape the final version of our paper. We thank the undergraduate and master students of The Ohio State University who helped us test Hang Doctor. In particular, we would like to thank Yuxiang Liu for his help in the software implementation of our solution. Finally, we would like to thank all the Ph.D. students of our Power-Aware Computer Systems (PACS) laboratory at The Ohio State University for all the invaluable time spent discussing research ideas, which has highly contributed to the successful publication of this work.

REFERENCES

- [1] Mohammad Mejbah ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. SyncPerf: Categorizing, Detecting, and Diagnosing Synchronization Performance Bugs. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*.
- [2] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run Software Failure Diagnosis via Hardware Performance Counters. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*.
- [3] Marco Brocanelli and Xiaorui Wang. 2017. Hang Doctor: Runtime Detection and Diagnosis of Soft Hangs for Smartphone Apps. <https://sites.google.com/site/hangdoctorhome/>. (2017).
- [4] Benjamin Elliott Canning and Thomas Scott Coon. 2008. Method, system, and apparatus for identifying unresponsive portions of a computer program. (2008). Microsoft Corporation, US Patent.
- [5] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C. Rinard. 2011. Detecting and Escaping Infinite Loops with Jolt. In *25th European Conference on Object-Oriented Programming (ECOOP '11)*.
- [6] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. 2011. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*.
- [7] Domenico Cotroneo, Roberto Natella, and Stefano Russo. 2009. Assessment and improvement of hang detection in the Linux operating system. In *28th IEEE*

- International Symposium on Reliable Distributed Systems (SRDS '09).*
- [8] Daniel Joseph Dean, Hiep Nguyen, and Xiaohui Gu. 2012. UBL: Unsupervised Behavior Learning for Predicting Performance Anomalies in Virtualized Cloud Systems. In *Proceedings of the 9th International Conference on Autonomic Computing (ICAC '12)*.
 - [9] Daniel J. Dean, Hiep Nguyen, Xiaohui Gu, Hui Zhang, Junghwan Rhee, Nipun Arora, and Geoff Jiang. 2014. PerfScope: Practical Online Server Performance Bug Inference in Production Cloud Computing Infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*.
 - [10] Daniel J Dean, Hiep Nguyen, Peipei Wang, Xiaohui Gu, Anca Sailer, and Andrzej Kochut. 2016. PerfCompass: Online Performance Anomaly Fault Localization and Inference in Infrastructure-as-a-Service Clouds. *IEEE Transactions on Parallel and Distributed Systems* vol. 27, no. 6 (2016), pp. 1742–1755.
 - [11] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '15)*.
 - [12] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. 2015. Log2: A Cost-aware Logging Mechanism for Performance Diagnosis. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*.
 - [13] Xiaoning Ding, Hai Huang, Yaoping Ruan, Anees Shaikh, and Xiaodong Zhang. 2008. Automatic Software Fault Diagnosis by Exploiting Application Signatures. In *Proceedings of the 22nd Conference on Large Installation System Administration Conference (LISA '08)*.
 - [14] Brad Fitzpatrick. 2010. Writing zippy Android apps. In *Google I/O Developers Conference*.
 - [15] Pierre-Marc Fournier and Michel R. Dagenais. 2010. Analyzing Blocking to Debug Performance Problems on Multi-core Systems. *SIGOPS Oper. Syst. Rev.* vol. 44, no. 2 (2010), pp. 77–87.
 - [16] GitHub. 2018. Home Page. <https://github.com/>. (2018).
 - [17] Ioana Giurgiu, Oriana Riva, and Gustavo Alonso. 2012. Dynamic Software Deployment from Clouds to Mobile Devices. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*.
 - [18] Google. 2018. Google Play Store. <https://play.google.com/store>. (2018).
 - [19] Android Developers Guide. 2018. Controlling the Camera. <https://developer.android.com/training/camera/cameradirect.html>. (2018).
 - [20] Android Developers Guide. 2018. Keeping Your App Responsive. <https://developer.android.com/training/articles/perf-anr.html>. (2018).
 - [21] Android Developers Guide. 2018. Package Index. <https://developer.android.com/reference/packages.html>. (2018).
 - [22] Android Developers Guide. 2018. Simpleperf. <https://developer.android.com/ndk/guides/simpleperf.html>. (2018).
 - [23] Peng Huang, Xiao Ma, Dongcai Shen, and Yuanyuan Zhou. 2014. Performance Regression Testing Target Prioritization via Performance Risk Analysis. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*.
 - [24] S.L. Jackson. 2012. *Research Methods and Statistics: A Critical Thinking Approach*. Cengage Learning.
 - [25] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*.
 - [26] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. 2011. Automated Atomicity-violation Fixing. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*.
 - [27] Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu. 2012. Automated Concurrency-bug Fixing. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI '12)*.
 - [28] Milan Jovic, Andrea Adamoli, and Matthias Hauswirth. 2011. Catch me if you can: performance bug detection in the wild. *ACM SIGPLAN Notices* vol. 46, no. 10 (2011), pp. 155–170.
 - [29] Yongin Kwon, Sangmin Lee, Hayoon Yi, Donghyun Kwon, Seungjun Yang, Byung-Gon Chun, Ling Huang, Petros Maniatis, Mayur Naik, and Yunheung Paek. 2015. Mantis: Efficient Predictions of Execution Time, Energy Usage, Memory Usage and Network Usage on Smart Mobile Devices. *IEEE Transactions on Mobile Computing* vol. 14, no. 10 (2015), pp. 2059–2072.
 - [30] Yeping Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and Detecting Performance Bugs for Smartphone Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*.
 - [31] Priya Nagpurkar, Hussam Mousa, Chandra Krintz, and Timothy Sherwood. 2006. Efficient Remote Profiling for Resource-constrained Devices. *ACM Transactions on Architecture and Code Optimization (TACO)* vol. 3, no. 1 (2006), pp. 35–66.
 - [32] Nithin Nakka, Giacinto Paolo Saggese, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2005. An Architectural Framework for Detecting Process Hangs/Crashes. In *5th European Dependable Computing Conference (EDCC '05)*.
 - [33] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*.
 - [34] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. 2013. Toddler: Detecting Performance Problems via Similar Memory-access Patterns. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*.
 - [35] Dan Pelleg, Muli Ben-Yehuda, Rick Harper, Lisa Spainhower, and Tokunbo Adeshiyun. 2008. Vigilant: out-of-band detection of failures in virtual machines. *ACM SIGOPS Operating Systems Review* vol. 42, no. 1 (2008), pp. 26–31.
 - [36] Michael Pradel, Parker Schuh, George Necula, and Koushik Sen. 2014. EventBreak: Analyzing the Responsiveness of User Interfaces Through Performance-guided Test Generation. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*.
 - [37] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P. Pipe, Thomas F. Wenisch, and Milo M. K. Martin. 2012. Computational Sprinting. In *Proceedings of the IEEE 18th International Symposium on High-Performance Computer Architecture (HPCA '12)*.
 - [38] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. AppInsight: Mobile App Performance Monitoring in the Wild. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*.
 - [39] Lenin S. Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. 2013. Timercard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *The 24th ACM Symposium on Operating Systems Principles (SOSP '13)*.
 - [40] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. 2016. EbbRT: A Framework for Building Per-Application Library Operating Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*.
 - [41] Abhishek B Sharma, Haifeng Chen, Min Ding, Kenji Yoshihira, and Guofei Jiang. 2013. Fault detection and localization in distributed systems using invariant relationships. In *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*.
 - [42] Kai Shen, Christopher Stewart, Chuanpeng Li, and Xin Li. 2009. Reference-driven Performance Anomaly Identification. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '09)*.
 - [43] Wook Song, Nosub Sung, Byung-Gon Chun, and Jihong Kim. 2014. Reducing Energy Consumption of Smartphones Using User-perceived Response Time Analysis. In *Proceedings of the 15th Workshop on Mobile Computing Systems and Applications (HotMobile '14)*.
 - [44] Xi Wang, Zhenyu Guo, Xuezheng Liu, and Zhilei Xu. 2008. Hang Analysis: Fighting responsiveness Bugs. In *Proceedings of the 3rd Conference on Computer Systems (Eurosys '08)*.
 - [45] Long Wangand, Zbigniew Kalbarczyk, Weining Guand, and Ravishankar K Iyer. 2007. Reliability microkernel: Providing application-aware reliability in the OS. *IEEE Transactions on Reliability* vol. 56, no. 4 (2007), pp. 597–614.
 - [46] Programmable Web Research Center. 2014. Growth in Web APIs From 2005 to 2013. <http://www.programmableweb.com/api-research>. (2014).
 - [47] Zilong Wen, Weiqi Dai, Deqing Zou, and Hai Jin. 2016. PerfDoc: Automatic Performance Bug Diagnosis in Production Cloud Computing Infrastructures. In *Trustcom/BigDataSE/I SPA*.
 - [48] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-sensitive Delta Inference for Identifying Workload-dependent Performance Bottlenecks. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '13)*.
 - [49] Lei Yang, Jiannong Cao, Shaojie Tang, Di Han, and Neeraj Suri. 2016. Run Time Application Repartitioning in Dynamic Mobile Cloud Environments. *IEEE Transactions on Cloud Computing* vol. 4, no. 3 (2016), pp. 336–348.
 - [50] Shengqian Yang, Dacon Yan, and Atanas Rountev. 2013. Testing for Poor Responsiveness in Android Applications. In *1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS '13)*.
 - [51] Andrew Zeigler, Shawn M Woods, David M Ruzyski, John H Lueders, Jon R Berry, and Daniel James Plaster. 2012. Hang recovery in software applications. (2012). Microsoft Corporation, US Patent.
 - [52] Lide Zhang, David R Bild, Robert P Dick, Z Morley Mao, and Peter Dinda. 2013. Panappticon: event-based tracing to measure mobile application and platform performance. In *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS '13)*.
 - [53] Yian Zhu, Yue Li, Jingling Xue, Tian Tan, Jialong Shi, Yang Shen, and Chunyan Ma. 2012. What Is System Hang and How to Handle It. In *IEEE 23rd International Symposium on Software Reliability Engineering (ISSRE '12)*.