Abstract—Nowadays, many applications do not exist independently but rely on various frameworks or libraries. The frequent evolution and the complex implementation of framework APIs induce lots of unexpected post-release crashes. Starting from the crash stack traces, existing approaches either perform application-level call graph (CG) tracing or construct datasets with similar crash-fixing records to locate buggy methods. However, these approaches are limited by the completeness of CG or dependent on historical fixing records, and some of them only focus on specific manually modeled exception types.

To achieve effective debugging on complex framework-specific crashes, we propose a code-separation-based locating approach that weakly relies on CG tracing and does not require any prior knowledge. Our key insight is that one crash trace with the description message can be mapped to a definite exception-thrown point in the framework, the semantics analysis of which can help to figure out the root causes of the crash-triggering procedure. Thus, we can pre-construct reusable summaries for all the framework-specific exceptions to support fault localization in application code. Based on that idea, we design the exception-thrown summary (ETS) that describes both the key variables and key APIs related to the exception triggering. Then, we perform static analysis to automatically compute such summaries and make a data-tracking of key variables and APIs in the application code to get the ranked buggy candidates. In the scenario of locating Android framework-specific crashing faults, our tool CrashTracker exhibited an overall MRR value of 0.91 and outperforms the state-of-the-art tool Anchor with higher precision. It only provides a compact candidate set and gives user-friendly reports with explainable reasons for each candidate.

I. INTRODUCTION

With the increasing size and rapid updating requirement of software, it is hard to eliminate all the bugs before the code release. To deal with these post-release bugs, developers need to analyze massive crash reports to find out the real buggy methods. Usually, one crash report is composed of three key elements, the type of exception, the crash description message, and the crash stack trace, in which the stack trace presents parts of the executed methods before the crash is triggered and is helpful for the developer’s code debugging process [34]. However, for precise fault localization, only using trace information is not enough. The real buggy method may not be the last executed one in the stack trace, or even not appear in the stack [25], i.e., the buggy method may not lead to the crash-triggering directly. Instead, after the buggy method is executed, its execution results affect the subsequent code execution and finally lead to an exception being thrown.

To quickly fix bugs with execution results, several fault localization approaches have been proposed. The spectrum-based methods [28]–[31], [38], [41] rank suspicious statements by computing the ratio of failed and passed test cases that execute the statements. These techniques impose high requirements on test cases, which are not suited for post-released crashes. For these crashes, both the test cases and the runtime coverage are usually unknown. Recent years, learning-based approaches [25], [37], [42] are widely adopted to solve this problem, for which higher precision relies on spending more effort on dataset labeling. Despite working well when similar crash-fixing records are collected, they are not good at handling unfamiliar new crash reports. Considering the generalizability, users can choose the analysis-based approaches [22], [24], [27], [35] to recover the real execution trace of the target application by backward tracking on the CG. Some of them [24], [35] improve the precision by manually modeling specific exceptions, and Kong et al. [27] retrieve similar crashes from the collected dataset before static analysis to precisely locate the out-of-stack buggy methods.

Nowadays, many apps are developed based on specific frameworks or libraries, e.g., the Android framework [1], google-map SDK [8], zxing library [16], etc. These framework- and library-specific exceptions account for the majority of app crashes [22]. However, developers have difficulty debugging and fixing them, especially in the understanding of method call ordering between application code and framework code [20]. To locate and understand the non-application crash faults, there are three obstacles when adopting the existing analysis-based approaches. First is the high modeling cost. As there are thousands of or even more rapidly evolving exceptions in one framework, e.g., Android, it is not cost-effective to manually model all the framework-level exceptions. Second is the deep call depth. The buggy point may be far away from the stack trace, but the number of candidates increases quickly when tracing deeper along the CG, e.g., developers may need to examine hundreds of candidates to find a newly discovered buggy point [25]. And if the analysis starts from the exception-triggering point in the framework or library-level code, the
candidate size will suffer more from long call traces and turn out to be extremely large. Finally, the unlinked call edges. Even though many candidates can be traced by CG analysis, the statically constructed CG is incomplete for reasons like UI callbacks, asynchronous methods, etc. So, the buggy method may be missed even with numerous candidates.

To cope with these problems, we propose a code-separation-based framework-specific fault localization approach that weakly relies on the CG edges and does not require extra prior knowledge. First, we separate the whole program code into two parts, the application-level and the framework-level. The application-level code contains a set of buggy methods to be located and fixed, which may misuse the APIs provided by the bottom-level code. The framework-level code provides APIs and throws exceptions when APIs are misused. By investigating the public framework-specific crash reports on GitHub [4], we note that one crash report with crash stack, exception type, and crash message can be mapped to a definite exception-thrown point in the framework. Thus, we can perform semantics analysis to pre-construct reusable summaries for framework-specific exceptions. To achieve that, the main challenges are which key information should be extracted from frameworks to understand the exception-triggering procedure, and how to use the exception-related information to precisely locate application-level buggy methods.

In this paper, we propose a specification technique, called exception-thrown summary (ETS), for framework-level methods, which describes the fault-inducing elements that lead to exception-triggering from the framework users’ point of view. We first perform static analysis to automatically compute ETSs for all the framework methods and apply the matched ETS on the given crash stack trace. The target ETS points out the parameters (keyVars) that may be wrongly input to the framework code, or the framework APIs (keyAPIs) whose invocation may influence the status-checking of the corresponding exception. With that information, we can perform more targeted application-level code analysis for buggy localization and generate a compact candidate set. Also, it helps users quickly figure out the complete path from the buggy methods to deep exception-thrown points for better explainability.

Android apps are typical framework-based applications, whose frameworks are much more complex than their upper-level apps. It has over ten million lines of code [23] and millions of call edges. With rapid evolution, the number of exceptions increases eight times more (1.643 to 13,717) from API 8 to 32. The high complexity of Android frameworks brings difficulties in bug debugging. In this paper, we take the Android crash fault localization as a typical scenario and implement our approach into a tool CrashTracker. In the evaluation, we collect 580 instances (569 Android apps and 11 third-party SDKs) and ten versions of Android framework SDKs. The results show that CrashTracker exhibited an overall mean reciprocal rank (MRR) metric value of 0.91. It also outperforms the state-of-the-art tool Anchor [27] with 7.4%, 11.5%, and 12.6% improvement on finding real buggy in the top 1, 5, and 10 sorted candidates. On average, only 6.35 explainable candidates are reported for one crash.

Contributions. The contributions of this work are threefold:

- We propose a code-separation-based analysis approach and apply it to Android framework-specific fault localization.
- We propose a novel specification, ETS, and construct it for 76,247 exceptions thrown in multiple Android frameworks.
- We implement our crash fault localization approach in tool CrashTracker [17], which can precisely locate buggy methods within a compact and explainable candidate set.

II. PRELIMINARY

A. Application-level and Framework-level Code

Among all the Android crashes, over 50% are framework-specific or library-specific ones [22], which indicates that the understanding of bottom-level code will influence code quality on the upper level. To avoid repetitive analysis of the large-scale and complex bottom-level code, we take the methods that may be debugged or fixed by developers as the application-level methods, and the methods that provide APIs to application-level methods as the framework-level ones. If we take all the methods in the whole program as $M_{\text{whole}}$, both the application- and framework-level methods are its subsets ($M_{\text{whole}} = M_{\text{app}} \cup M_{\text{frame}}$). There is no fixed division between two parts, i.e., the separation line could be totally customized by users under different scenarios. In this paper, to debug the Android framework-specific crashes, we add all the application and library code contained in the Android Packages (APKs) and the third-party Android software development kits (SDKs) into $M_{\text{app}}$, while adding the official Android framework methods in $M_{\text{frame}}$.

B. From Crash Report to Buggy Method

On the bug-tracking systems (e.g., GitHub [7], StackOverflow [13]), the crash reports provided by users mainly contain three elements, an exception type, a crash message, and a stack trace snapshot that reflects the callee-caller method chain when the exception is triggered. In the stack trace, the application- and framework-level methods may show up alternately. Table I displays one crash issue [3] of the Android app cgeo, which has 1.2k stars. In this example, an IllegalStateException is thrown out with a crash message “attempt to re-open · · ·”. We label signaler, crashAPI, crashMethod, and entry tags beside methods in the trace, and give the fixed buggy method in the last row, which is out of the stack.

Actually, when a crash is triggered, the buggy method must be located in the execution trace, i.e., it should be a

<table>
<thead>
<tr>
<th>Type</th>
<th>java.lang.IllegalArgumentException</th>
</tr>
</thead>
<tbody>
<tr>
<td>Msg</td>
<td>attempt to re-open an already-closed object: SQLiteDatabase: SELECT count(*) FROM cg_caches WHERE reason = &quot;1&quot;</td>
</tr>
<tr>
<td>Crash</td>
<td>android.database.sqlite.SQLiteDatabase.acquireLockReference, [signaler]</td>
</tr>
<tr>
<td>Stack</td>
<td>android.database.sqlite.SQLiteDatabase.acquireLockReference, [signaler]</td>
</tr>
<tr>
<td>Trace</td>
<td>cgeo.geocaching.DataStoresPreparedStmts.simpleQuery, [crashAPI]</td>
</tr>
<tr>
<td>Buggy</td>
<td>cgeo.geocaching.DataStoresPreparedStmts.clearPreparedStmts</td>
</tr>
</tbody>
</table>
method that has been executed already. The execution trace of application app can be denoted as

\[ T_{\text{execute}} = \langle f_0, \ldots, f_i, f_{i+1}, \ldots, f_n \rangle \]

in which \( f_i \) is a method. For Java programs that have a single entry method entry, we have \( entry = f_0 \). And the pair \( (f_i, f_{i+1}) \) is a call edge in the CG, i.e., \( f_i \) invokes \( f_{i+1} \).

For event-driven Android programs that have multiple callback entries, like component lifecycle and user/system callbacks, we have \( f_0 \in S_{\text{entry}} \). In this case, the method \( f_{i+1} \) may be a callee of \( f_i \), or be a callback method \( f_{i+1} \in S_{\text{entry}} \). As \( T_{\text{execute}} \) is a crash-triggering execution trace, its last method \( f_n \) is the signaler method that directly throws the exception out. The method \( f_i \) in \( T_{\text{execute}} \) could be either an application-level or a framework-level method. As the Android framework-specific exceptions denote the exceptions thrown in the Android framework [22], here we focus on the crashes whose signaler \( \in M_{\text{frame}} \).

The target of the fault localization is to find out the real buggyMethod to be fixed, which must be one of the executed methods. That is to say, we can find an integer \( b \) so that \( f_b \) in \( T_{\text{execute}} \) and buggyMethod = \( f_b \).

During execution, methods in \( T_{\text{execute}} \) will be pushed into a stack \( st \) in order and be popped out when finished. The crash trace \( T_{\text{crash}} \) records the unpopped methods in the stack \( st \) when the crash is triggered, whose set of elements is a subset of the complete execution trace, i.e., \( \text{Set}(T_{\text{crash}}) \subseteq \text{Set}(T_{\text{execute}}) \). We can denote \( T_{\text{crash}} \) as a sequence \( T_{\text{crash}} = \langle f_n, \ldots, f_j, f_{j-p}, \ldots, f_e \rangle \), \( (0 < p < j, 0 \leq e < j-p) \), where each method in it also exists in \( T_{\text{execute}} \). The signaler method \( f_n \) is the top element in stack \( st \), and entry is the bottom one. Sometimes, the app developers do not invoke the signaler directly, instead, they invoke the crashAPI \( f_{ca} \) to indirectly invoke the signaler, where for all \( j, ca \leq j \leq n, f_j \in M_{\text{frame}} \). We call the last method that directly invokes crashAPI as the crashMethod. For a crashAPI \( f_{ca} \), the crashMethod \( f_{cm} \) is its previous element in \( T_{\text{crash}} \), where \( f_{cm} \in M_{\text{app}} \) and \( f_{ca} \in M_{\text{frame}} \). Trace \( T_{\text{crash}} \) is a slice of \( T_{\text{execute}} \). However, it is not determined whether the buggyMethod is in the trace \( T_{\text{crash}} \) or not, which makes the fault localization challenging. Starting from \( T_{\text{crash}} \), the main task is to find a compact ranked candidate set that contains the most possible buggy methods that exist in \( T_{\text{execute}} \).

Fig. 1 introduces a simplified CG, in which each block denotes a method and each node in the block is a method call statement labeled with the callee. Here, \( f_0 \) and \( f_1 \) are both entry methods. We take dummy as a dummy entry point of the program that invokes \( f_0 \) and \( f_1 \). Among methods, solid and dotted lines are used to represent the direct and indirect call edges. Suppose there is a crash-triggering execution that starts from the entry method \( f_0 \) and ends with the signaler method \( f_{11} \). When the app crashed, only four methods \( (f_1, f_3, f_7, f_{11}) \) are stored in the method stack \( st \), while others have finished their execution. As the buggyMethod may not be in \( st \), existing approaches make an expansion of stack trace based on the call edges in CG. Their candidate size entirely depends on the call depth setting. The call depth of the function \( f_i \) with respect to a given crash trace \( T_{\text{crash}} \) is the least number of function call steps from any functions in \( T_{\text{crash}} \) to \( f_1 \) [40]. In Fig. 1, the nodes with the same color have the same call depth. With a given call depth threshold \( d \), methods that have a call depth no larger than \( d \) will be collected. And a larger depth means a higher possibility to find out the buggyMethod. When \( d = 0 \), we only have \( T_{\text{crash}} \). Then we increase the depth until finding the buggy method. Suppose that \( f_0 \) is the buggyMethod, when \( d = 1 \), we can get seven ineffective candidates. And when \( d = 2 \), twice as many candidates will be collected, which includes \( f_0 \). According to that, there are two challenging cases. First, the buggyMethod may be far away from the mainstream, e.g., buggyMethod = \( f_p \). To find it out, a larger depth brings more candidates to be reviewed and increases the difficulties in fault analysis, especially when \( f_n \in M_{\text{frame}} \). Second, the CG may be incomplete due to the existence of callbacks, native methods, or asynchronous calls, and the buggy method can be called by these unlinked methods, e.g., buggyMethod = \( f_p \). That means, tracing along the CG requires heavy effort, but the buggyMethod still could be lost. Thus, based on the basic CG relationship, we also pay attention to the information hidden in the framework-level thrown exceptions.

III. MOTIVATING EXAMPLE

We use the real crash report displayed in Table I as our motivating example. All the code snippets corresponding to this crash are shown in Fig. 2. First, the application-level method getNumberOfPaints() invokes the crashMethod simpleQuery(). This crashMethod then invokes a framework-level crashAPI in line 8, which invokes the signaler method in line 24 and triggers an exception in lines 33-34. For this crash, the really buggy points (lines 11-12) are in clearPreparedStmts(), which is not shown in the stack trace \( T_{\text{crash}} \). The buggy reason is that the instance of PreparedStmt is closed without a clear operation so that it will not be reinitialized but reused directly the next time, which finally leads to a crash. As the entry method run()
Fig. 2: Motivating Example of Framework-specific Exception

in $T_{\text{crash}}$ is asynchronous, its caller is not stored in the stack, i.e., the real callback that invokes the buggyMethod is missed. Thus, the methods that are called by the real entry are also lost. However, even if the real entry method is included in the stack, the buggyMethod is still hard to be retrieved, as it is far away from the crash trace and is hidden in the large candidate set. Therefore, before making actual fault localization, we should first figure out how and why exceptions are thrown in $M_{\text{frame}}$ and get their characteristics.

IV. FRAMEWORK-SPECIFIC FAULT LOCALIZATION

In this paper, we propose an exception-oriented summary specification that points out the fault-including elements from the view of framework users. It can be the parameters (keyVars) that are wrongly input to the framework code, or the framework APIs (keyAPIs), whose invocation can influence the status-checking results of the exception-related key conditions (keyConds). Based on this specification, we perform a one-time static analysis to automatically compute summaries for framework code. For a crash report to be debugged, we match and apply the computed summary to the crash trace, which helps to retrieve the complete call trace between the application code and framework code. In the application-level analysis, according to the type of the fault-inducing elements, we can focus on the fault-inducing variable in the crashAPI invocation statement and make data tracking on it. Also, we can target the fault-inducing APIs that be invoked in the application code and track its callers. Using this approach, the methods that can be traced on the expanded CG but are not data-related will be excluded, while the methods that are data-related but cannot be traced by CG are included. Fig. 3 introduces the overview of our approach CrashTracker, which takes an APK file and a crash report as input and generates ordered candidate methods with inferred fault-inducing reasons. It has two modules. The exception semantics analysis module works on the framework-level code, which takes multiple Android framework files as input and outputs the summaries for exceptions in $M_{\text{frame}}$. The summaries will be passed to the fault localization module, which then uses the received information to locate buggy methods in $M_{\text{app}}$. 

V. EXCEPTION SEMANTICS ANALYSIS ON $M_{\text{frame}}$

On $M_{\text{frame}}$, the Exception-thrown summary (ETS) is designed for each exception-thrown point to present its fault-inducing elements. It can be formally defined as a 5-tuple $\mathcal{E}TS(e) = (id, S_{\text{cond}}, S_{\text{condVar}}, S_{\text{keyVar}}, S_{\text{keyAPI}})$, where

- $id$ is the identifier of exception $e$, which is a four-tuple $\langle \text{sink}, \text{signaler}, \text{type}, \text{msg} \rangle$, in which $\text{sink}$ is a statement that throws the exception $e$; $\text{signaler}$ is the method that contains $\text{sink}$; $\text{type}$ is the type of the exception $e$; and $\text{msg}$ is the description message when $e$ is triggered, which is composed of constant and dynamically-assigned values. To match the context-related messages in all the forms, we represent $\text{msg}$ by regular expressions;
- $S_{\text{cond}}$ is a set of key conditions ($keyCond \in S_{\text{cond}}$) located in $\text{signaler}$, whose results can decide whether $e$ is triggered. If $e$ is triggered only when $keyCond$ is satisfied, $keyCond$ is a basic check. If the $\text{throw}(e)$ statement can not be executed as the satisfaction of $keyCond$ leads to method return, $keyCond$ is a not-return check;
- $S_{\text{condVar}}$ is a set of condition variables ($\text{condVar} \in S_{\text{condVar}}$) whose values are directly checked in $S_{\text{cond}}$;
- $S_{\text{keyVar}}$ denotes a set of key variables, which can influence the value of $\text{condVar}$ and can be modified by framework users. Each $\text{keyVar} \in S_{\text{keyVar}}$ is a triple
\langle mtd, loc, condVar \rangle$, in which $mtd$ is a framework-level public method; $loc$ is a parameter location in method $mtd$; the $loc^{th}$ parameter in $mtd$ can influence the value of $condVar \in S_{cond\forall}$ by inter-procedural parameter passing, i.e., its value can influence the checking results of the key conditions. For simplicity, we will use $keyVar$ to denote the parameter variable in $keyVar.mtd$ with location $loc$.

- $S_{keyAPI}$ denotes a set of key APIs, which can influence the value of $condVar$ and can be invoked by framework users. Each $keyAPI \in S_{keyAPI}$ is a 4-tuple $\langle mtd, field, condVar, dpt \rangle$, in which $mtd$ is a framework-level public method; $field$ is a class-filed variable that is modified by $mtd$ or its callees, whose value is data-related with the condition variable $condVar$ in $S_{cond\forall}$; $dpt$ records the least number of function call steps from $mtd$ to the method that directly modifies $field$, which is used in candidate sorting. We will use $keyAPI$ to denote the method $keyAPI.mtd$ in the following.

Here, we use the running example in Fig. 2 to exemplify the defined elements in ETS. For the exception thrown in lines 33-34, its $keyCond$ is $mRefCount \leq 0$ and $mRefCount$ is a $condVar$. Also, $mRefCount$ is a filed variable of class $SQLiteClosable$ that can be modified out of the $signaler$ method. As public methods $releaseReference()$ and $close()$ both modify its value, $\langle releaseReference() \rangle$, $mRefCount$, $mRefCount$, 1 and $\langle close() \rangle$, $mRefCount$, $mRefCount$, 2 are added into $S_{keyAPI}$. In this example, as no parameter variable is related to the $condVar$, $S_{keyVar}=\emptyset$. But if line 29 is replaced by lines 30-31 (case 2), we will get a parameter-related variable $count$, which can be modified outside and influence the value of $condVar$. As $signaler$ is public, we will first add $\langle acquireReference(), 2, mRefCount \rangle$ into $S_{keyVar}$ and then trace the caller of method $acquireReference()$ to find more $keyVars$.

**Sink Point Extraction.** The ETS construction process is shown in Algorithm 1. The first step is to identify the sink points (line 1). First, all the $throw(e)$ [14] invocation points are sink points. Besides, developers may also customize exception-thrown information and use the logged information to debug. To recognize them, we detect all the $Runnable$ instances and trace their data flows. For the exceptions that are not thrown directly, we take the methods that receive these instances as exception-handling methods and take the invocation statements of them as sink points. $throw(e)$ statement is the most frequently used sink point type with many instances. Besides it, we get another 33 types of sink points, of which 12 store exception trace into logs or files.

**Message Representation.** Starting from the sink points, one challenge is how to extract the description message of the target exception (line 4). For the same exception, the runtime crash message may be different, as the values of some variables are dynamically assigned. To make a precise matching, we transform the exception message into a regular expression [11] pattern, so that it can match multiple runtime crash messages. The method $regexStringAnalysis()$ in line 4 tracks the definition statement of the target exception, which may be a newly created one, e.g., $e = new RuntimeException(\ldots)$. For the former, we perform backward value tracing of the message-related parameter in the exception’s constructor method. For the latter one, we make inter-procedure tracing to get the real instantiatate point and analyze it as the former. During the value tracing, we model a set of String-related APIs to stitch multiple parts together, in which we use $\backslash[\text{\$\$}]\text{E}$ to represent a symbolic value and use $\backslash[\text{\$\$}]\text{str}$ to represent the constant value of $str$. In lines 33-34 of Fig. 2, the target exception message is "\text{"\text{Qattempt to re-open an already-closed object: \text{\$\$}}\text{E}"\text{", which can match the crash message in Table I.}

**Key Condition and Condition Variable.** Each exception is influenced by a set of conditions (keyConds), whose checking results decide whether the exception can be triggered. In line 6, we invoke method $getKeyCondsAndVars(s)$ to trace all the predecessor statements of the sink point in the control flow graph, record the involved condition checks into $S_{cond\forall}$.

Algorithm 1 Exception-thrown Summary Extraction

**Input:** method $signaler$ in $M_{frame}$

**Output:** the exception summary set $s_{ets}$ on method $signaler$

1: for sink in $signaler$.getSinkPoints() do
2: ets = createETS($signaler$, sink)
3: ets.type = exceptionTypeAnalysis()
4: ets.message = regexStringAnalysis()
5: cfg = $signaler$.getCFG()
6: getKeyCondsAndVars(ets, cfg)
7: if ets.keyConds.size() != 0 then
8: for return stmt retStmt predsOf sink in cfg do
9: getKeCondAndVars(ets, retStmt)
10: end for
11: end if
12: worklist = ets.keyCondVars.copy()
13: while worklist.size() > 0 do
14: if worklist.get(0) is parameter or field related then
15: add worklist.getAndPop(0) as outsideVars
16: else
17: defStmt = getDefStmt(worklist.getAndPop(0))
18: worklist.add(defStmt.getRightOp(), getVars())
19: end if
20: end while
21: for varp in ets.parameterOutsideVars do
22: add varp to ets.keyVars if $signaler$ is public
23: track varp in $signaler$’s caller to update ets.keyVars
24: end for
25: for varf in ets.fieldOutsideVars do
26: add public methods that modify varf into ets.keyAPIs
27: update public caller of keyAPIs into ets.keyAPIs
28: end for
29: $s_{ets}$.add(ets)
30: end for
31: return $s_{ets}$

Algorithm 2 getKeyCondsAndVars

**Input:** ETS $ets$, control flow graph $cfg$, statement $s$

**Output:** updated ETS $ets$

1: for condition check stmt condStmt predsOf $s$ in $cfg$ do
2: ets.keyConds.add(condStmt.getCond())
3: ets.keyCondVars.add(condStmt.getCond()).getVars()}
4: end for
and collect all the condition-related variables into $S_{\text{condVar}}$, as displayed in Algorithm 2. After that, we can collect all the basic checks. For example, $s=t$ is a basic check for \{if (s==t) {throw(e)}\}. In another case, developers may take the method-return operation as expected behavior and throw exceptions if the method didn’t jump out in time. The conditions related to these return statements are also key conditions. For example, $s=t$ is a not-return check for \{if (s==t) {return} throw(e)}\}. In lines 7-11, if there is no basic check, we extract the not-return checks by analyzing the basic checks of all the return statements prior to the sink point in the control flow graph. Besides, as the conditions far away from the exception-thrown point may have a weak relationship with the exception, we count the average condition length as a threshold size of $S_{\text{cond}}$.

**Key Variable and Key API.** To find out keyVars and keyAPIs, we first use a worklist algorithm to locate the method parameters and class-field variables that influence the value of condVars by performing backward data tracing along the use-def-chains [15] (lines 12-20). Note that, in line 17, method getDefStmt() returns the Jimple [9] IR level definition statements, which reflect the variables that can influence that target one. As these variables can be modified outside signaler, they are called outside variables outsideVar. In lines 21-24, for each parameter-related outsideVar varp, we record its method, the location in the parameter list, and the influenced condVar. If the signaler is a public method, varp itself is a keyVar (line 22), e.g., we have \{acquireReference(), 2, mRefCount\} for case 2 in Fig. 2. Similarly, we perform backward inter-procedural parameter call-chain analysis (line 23) to trace more key variables passed through other framework APIs, which invoke the signaler. That is, for a formal parameter, we find the actual parameter variable in its caller and judge whether the passed variable is influenced by the caller’s parameter-related outsideVars. If it is, these newly detected outsideVars will be added into $S_{\text{keyVar}}$, e.g., for method f(int count) \{acquireReference(1, count)\}, we further have \{f(), 1, mRefCount\}. In lines 25-28, for each field-related outsideVar varf, its value can be changed by other framework APIs that influence the checking results of keyConds. In line 26, we record the methods that change the value of varf, the field varf, the influenced condVar, and the call depth from method to signaler. For Fig. 2, we first get the keyAPI \{releaseReference\}, mRefCount, mRefCount, 1\}. Then in line 27, we trace callers of the collected keyAPIs and get \{close\}, mRefCount, mRefCount, 2\}. Finally, in line 29, we add ETS for each exception into the ETS set $S_{\text{ets}}$.

### VI. Crash Fault Localization on $M_{\text{app}}$

Based on the exception semantics analysis module, we further match the given crash report to its corresponding ETS and use the extracted keyVars and keyAPIs to guide the application-level buggy method analysis.

**ETS Mapping.** Algorithm 3 displays the process of crash fault localization on $M_{\text{app}}$. In line 1, we match the given crash message with the regular expression format message of each ETS. When the version of the framework is given, the target ETS is unique and can be used directly. If the version is undetermined, multiple ETSs may be matched as the exception with the same type and the description message can exist in many versions. In the latter case, the target ETS should be picked by a proper version-choosing strategy. Here, we first classify the matched ETSs by their characteristics into five ETS-related types, which are listed in Table II. Then we get a list of matched versions for the most classified type. To keep the randomness, the ETS in the middle of the list is picked.

**Candidate Picking.** In lines 2-14, according to the information provided by the target ETS, we use different candidate-picking strategies. Table II also displays the four strategies that can be used for each ETS-related type. For the ETS who has no condition variable, e.g., no condVar and keyCond, we use strategy S1. In this case, the signaler method should not be invoked directly, framework users should override that method and invoke the newly implemented method by the polymorphic mechanism. For this type, the number of candidates is limited by the number of subclasses of the declared class of signaler. For target ETS who has condVar but has no outside variable, we use strategy S2. The exceptions caught from try-catch blocks and the exceptions whose condition variables are related to methods with unknown implementations (e.g., native method) are both in this type. Without extra information about how the fault could be induced, we just make data tracing starting from the invocation statement of crashAPI in the

<table>
<thead>
<tr>
<th>ETS-related Types</th>
<th>Fault Localization Strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1: No CondVar</td>
<td>S1: Override analysis in subclasses</td>
</tr>
<tr>
<td>T2: No OutsideVar</td>
<td>S2: Data tracing from variables in crashAPI</td>
</tr>
<tr>
<td>T3: Only have keyVar</td>
<td>S3: Data tracing of variables keyVars</td>
</tr>
<tr>
<td>T4: Only have keyAPI</td>
<td>S4: Call tracing of methods invoking keyAPIs</td>
</tr>
<tr>
<td>T5: Have keyVar, keyAPI</td>
<td>S5: Call tracing of methods invoking keyAPIs</td>
</tr>
</tbody>
</table>

**Algorithm 3 Crash Fault Localization**

**Input:** Android app app, crash stack trace st and message msg, framework version ver, exception summary set $S_{\text{ets}}$, on $M_{\text{frame}}$

**Output:** fault localization reports reports

1: $ets = \text{getBestMatchETS}(st, msg, S_{\text{ets}}, ver)$
2: $\text{condType} = \text{getConditionTypeOfETS}(ets)$
3: $\text{strategies} = \text{getStrategiesByType}(\text{condType})$
4: for strategy in strategies do
5: if strategy = S1 then
6: locate methods that should override $ets$.signaler
7: else if strategy = S2 then
8: locate methods that are data-related with the crashAPI-invoking statement
9: else if strategy = S3 then
10: locate methods that are data-related with keyVars
11: else if strategy = S4 then
12: locate methods that are callers of keyAPIs
13: end if
14: end for
15: candis = filterAndSortCandidates()
16: getCodeAndNonCodeFaultReasonReports(candis, reports)
17: return reports
**Candidate Reasoning.** To make the localization results explainable, the reports should contain both the summary information of the corresponding exception, and the code-/non-code-level relationship between each buggy candidate and the triggered exception. By previous ETS matching, we can get the corresponding ETS summary of each crash and provide that to users. In line 16, based on the analysis results of the $M_{\text{frame}}$ and $M_{\text{app}}$ code, we report candidates with code-related buggy reasons, which discover the relationship between each candidate and the triggered exception. If one candidate is traced by multiple strategies, only the highest score is reserved, but all the possible reasons are recorded. More detailed fault localization reports and instructions can be found in our online tool documentation [17]. Overall, our approach gives six types of code-related reasons, including 1) *KeyAPI*\_Related, invokes method *keyAPI* with trace $t$; 2) *KeyVar*\_Related\_1, influences the value of *keyVar* by modifying the value of the passed parameter $p$; 3) *KeyVar*\_Related\_2, influences the value of *keyVar* by modifying the value of related field $f$; 4) *Executed*\_1, doesn’t influence the *keyVar* but is in the crash trace; 5) *Executed*\_2, not in the crash stack but has been executed; 6) *Not Override*, forgets to override the *signaler*. For the motivating example in Fig. 2, the simplified report is like this: **Candidate:** clearPreparedStmts(); **Rank:** 1; **Type:** *KeyAPI*\_Related; **Trace:** (clearPreparedStmts() → call *keyAPI* close() → call releaseReference() → modify the outsideVar mReCount → call acquireReference() with the condVar mReCount → trigger exception.).

In some cases, the crash-triggering not only relates to the misuse of APIs but also relates to the non-code reasons, e.g., Kong et al. [27] points out five non-code tags, including *Manifest*, *Asset*, *Hardware*, *OS Version* and *Resource*. To get this extra information, we scan the statements that are data-related with the *condition variables* to judge whether non-code-related keywords in Table III are hit or not. For instance, for tag *Manifest*, the usage of field mActivityInfo and all the permission strings are detected. For tag *OS Version*, we trace whether the filed *targetSdkVersion* is read. If these tags are matched, they will be displayed as non-code buggy reasons.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Keywords in Code Slices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manifest</td>
<td>ActivityInfo mActivityInfo, android.permission</td>
</tr>
<tr>
<td>OS Version</td>
<td>ApplicationInfo mtargetSdkVersion</td>
</tr>
<tr>
<td>Hardware</td>
<td>MediaPlayer, BluetoothAdapter, Camera, etc.</td>
</tr>
<tr>
<td>Asset</td>
<td>android.content.res.AssetManager</td>
</tr>
<tr>
<td>Resource</td>
<td>android.content.res.Resources</td>
</tr>
</tbody>
</table>

VII. Evaluation

We implemented the framework-specific fault localization approach as a prototype CrashTracker [17], which consists of 11,554 lines of Java code. It is extensively based on the static analysis framework Soot [12] and uses Flowdroid [6] to construct call graphs. The evaluation of CrashTracker aims to answer the following research questions.

- **RQ1 (ETS Construction):** How many ETSs can we extract from multiple-version Android frameworks?
• RQ2 (Fault Localization): How beneficial are our strategies in fault localization? Can CrashTracker help to locate buggy methods effectively compared to the existing tool?

• RQ3 (Precision Analysis): What are the key reasons that lead to false positives and false negatives?

A. Experimental Setup

To answer RQ1, we collect multiple version framework files. As the android.jar files in the Android SDK only contain stub methods but not real code implementation, we load the published Android images and pull the complete jar files from the system. Overall, we collect ten versions of framework files, which correspond to Android 2.3 to 12.0 [2], in which Android 3 is excluded as it is unavailable.

To answer RQ2-RQ3, the Android framework-specific crash dataset is required. There are two off-the-shelf benchmarks that relate to Android crash datasets. For example, Fan et al. [21] extract 194 crashes from GitHub [7] to form a crash dataset. And in the recent benchmark ReCBench [26], more than 1,000 crash apps are collected, in which nearly half of the crashes are framework-related. To focus on the framework-specific crashes only, Kong et al. [27] filters these two datasets with the following criteria. First, the stack trace must contain the application-level method. Second, the signaler must locate in \( M_{frame} \). After filtering, it extracts 500 crashes (D500) on ReCBench [26] and 69 crashes (D69) in existing work [21], which are divided into three buckets according to the location of the buggyMethod. Category A: buggyMethod in \( T_{crash} \); Category B: buggyMethod in \( T_{exec} \) but not in \( T_{crash} \); Category C: crash arises from non-code reasons. We reuse and update these 569 crashes as our evaluation dataset by adding oracle information about both the condVars and outsideVars.

For type C, the buggyMethods are labeled with non-code characteristics only in the original dataset, e.g., lack of resources. These labels indicate the external cause of the crash triggering, while in fact, these non-code characteristics also have their corresponding code snippet, e.g., resource loading statements. As our approach can both provide code-level and non-code-level localization, we point out the code-level buggyMethods for crashes in category C and record their original labels as extra non-code characteristics. As CrashTracker is a general technique, it can also be applied to other scenarios. Besides the collected Android APKs, we also take the Android third-party SDKs as the application-level code. We search the SDK library projects on GitHub that have large star numbers, active commit behavior, and normalized issue submission specification. By manually reviewing, we pick two popular projects, facebook-android-sdk [5] and google-map [8]. Then, we filter issues with the keyword `is:issue is:closed "AndroidRuntime" OR "Crash"`. For the 84 + 53 issues matched, we manually check whether the crash is Android framework-specific and whether a fixing commit is given. Overall, 11 framework-specific crash reports with fixing revision (D11) are collected. They form a large dataset with 580 crash reports (D580).

For effectiveness evaluation, as the analysis-based Java fault localization tool CrashLocator [40] is not available, we implement a similar strategy (b1-ExtendCG) in our tool and perform self-comparison. Besides, Anchor [27] is a novel Android framework-specific fault localization tool, which first applies machine learning algorithms to categorize each new crash into a specific category (A/B/C), and then combines the application-level static analysis and similar crash query to achieve the final buggy ranking. We also compare with it the 569 test cases provided by them. All of our analyses are performed on a Linux server that has two Intel® Xeon® E5-2680 v4 CPUs and 256 GB of memory. Our approach has two phases, in which the framework-exception extraction is one-time work. We analyze ten versions of frameworks with 67 min, i.e., around 6.7 minutes for one version. In the app code analysis, we use around 95 minutes for 580 apps with 8 threads, i.e., 10 seconds per app.

B. RQ1: ETS Construction

The Android framework updates rapidly, so the exceptions may also change with time. Fig. 4 shows the number of ETS in different versions of the Android framework, in which each ETS denotes one unique exception. With the evolution of the framework, the number of exceptions, exception types, and exception-thrown methods all increase. For example, in Android 10.0, 10,385 exception-thrown methods throw 7,415 exceptions with 176 types. Among them, 6,917 ETSs have not-empty messages description, of which 29.0% of them contain non-constant values in the generated regular expression.

Fig. 4: ETS in Multiple Android Frameworks

For each exception, we trace its keyConds, condVars and the outsideVars. When tracing all the key conditions related to an exception, the average condition length is 2.78. So we set the threshold in keyCond collection as 3. Among all the conditions, most are basic ones. The not-return conditions account for 0.7%, which influences the precision of 462 framework exception triggering. Table V shows the distribution of the ETSs with different condition types. We give the statistical results on the oldest version (Android 2.3), the latest version (Android 12.0), and the average value for all ten versions. According to the results, most ETSs only have keyVars (42%) or keyAPIs (22%). Around 18% ETSs have both keyVars and keyAPIs and 4% ETSs do not have keyCond. About 15% ETSs could not link with any outsideVar, most of which are from the caught and re-thrown exceptions or the inter-procedural-call-related conditions. On average, we can get 7,625 ETSs from the Android framework code. Among them, there are 1,859 ETSs that contain 10,227 keyVar records, and 2,877 ETSs with 308,852 keyAPIs. For the keyAPIs, 81,872 are declared in the same class of signaler, and 226,980 are located in different classes.
Moreover, we analyze the relationship between the ETS-related types and the crash categories in D580. As shown in Fig. 5, most crashes in category A only have keyVars, which is consistent with the in-stack behavior of crashes in category A. And for category B, whose buggy methods are out-of-stack, there are more crashes that have keyAPIs than in other categories. This reflects the keyVar and keyAPI analysis work well on both the in-stack and out-of-stack crashes.

![Fig. 5: ETS-related Types on Each Category](image)

To check the correctness of the analyzed ETS-related type, we review the exception-triggering code snippets related to the collected 580 crash reports. Two experienced Java developers read the crash trace information and retrieve the corresponding exception in the source code of the Android framework. By manual analysis, they record the key conditions, the condition variables and the outside variables for all the exceptions triggered in D580. By comparison, we find that CrashTracker can correctly identify the ETS-related types (refer to Table II) for 95.5% crashes. There are 26 ones that are misidentified, of which 12 are exceptions in the Android support libraries, which are not collected as framework code; three crash reports provide empty message information, which makes the message matching fail. Three involve inter-procedural tracing. The others are condition related, including one lack of conditions due to the length limit, three getting unrelated not-return conditions, two re-throwing a catch condition with unknown outsideVar, and two wrongly taking synchronized variable or the final constant field as outsideVar, which actually will not work well on both the in-stack and out-of-stack crashes.

### C. RQ2: Effectiveness of Fault localization

To evaluate the effectiveness of CrashTracker, we first made a group of self-comparisons. The results are displayed in Table IV. The second column displays how many buggyMethod can be located by the tool in its candidate list. The following RankSum gives the cumulative sum of the ranking of buggyMethod in the candidate list. If the buggyMethod is not found, we use max(candiSize + 1, n) as its ranking value, in which we suppose users have to look up at least n candidates to find the target method (n = 20 by default). The column CandiAvg gives the number of provided candidates on average. The following eight columns give the precision evaluation results on D580, especially on the crashes with category B. Two metrics Recall and Mean Reciprocal Rank (MRR) [10] are used, in which #R@N counts the number of reports that rank buggyMethod in its first N candidates and MRR denotes the mean of the multiplicative inverse of the rank of the first correct location. It can be calculated by the formula $MRR = \frac{1}{E} \sum_{n=1}^{E} \frac{1}{\text{Rank}_n}$.

The first line gives the default results of CrashTracker, which can find correct buggyMethod for 568/580 crashes. For CrashTracker, its RankSum is 954. CrashTracker can provide a compact list with only 6.35 candidates. For all the 580 crashes, CrashTracker has high precision at R@1, R@5, and R@10. For crashes in category B, it can still find out most of the buggyMethod with a few candidates, e.g., 68% for 5 candidates and 77% for 10 candidates. Besides the code-level localization, we also label the non-code reasons as buggy tags for 45 crashes located in category C. Comparing the given labels, our non-code reason analyzer can correctly infer 27/45 already labeled tags, and we can observe another 40 tags that are not labeled in the original dataset.

Then, we compare CrashTracker with a set of variants. In strategy b1, we implement a CG-expansion-based approach [40] to trace invoked methods along the call edges with a call depth of 5. However, this approach generates too many candidates, which increases by 20.6 candidates for a crash on average. And it only can locate 57% buggyMethod in category B with 10 candidates. The strategy b2 is used to evaluate whether tracking conditions with length three influence the results. In this strategy, all the conditions will be collected. However, it did not bring higher precision but reported a bit more candidates. In b3, we suppose the ETS information is not available, i.e., only strategy $S_2$ is adopted, which decreases the overall precision, especially for cases in category B. Moreover, in Table VI, we present the precision of CrashTracker on these ETS-related types, including the number of cases under test (count), the ratio of reports that rank buggyMethod in

### TABLE IV: Effectiveness of CrashTracker with Multiple Strategies

<table>
<thead>
<tr>
<th>Strategy</th>
<th>CrashTracker</th>
<th>b1-ExtendCG</th>
<th>b2-NoCondition</th>
<th>b3-NoCondType</th>
<th>b4-NoKeyVar</th>
<th>b5-NoKeyAPI</th>
<th>b6-NoCallable</th>
<th>b7-Version2.3</th>
<th>b7-Version8.0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Find</td>
<td>RankSum</td>
<td>CandiAvg</td>
<td>#R@1</td>
<td>#R@5</td>
<td>#R@10</td>
<td>MRR</td>
<td>#R@1</td>
<td>#R@10</td>
</tr>
<tr>
<td>CrashTracker</td>
<td>568</td>
<td>954</td>
<td>6.35</td>
<td>500</td>
<td>562</td>
<td>567</td>
<td>0.91</td>
<td>14</td>
<td>38</td>
</tr>
<tr>
<td>b1-ExtendCG</td>
<td>-8</td>
<td>+564</td>
<td>+20.63</td>
<td>-5</td>
<td>-11</td>
<td>-13</td>
<td>-0.02</td>
<td>-8</td>
<td>-7</td>
</tr>
<tr>
<td>b2-NoCondition</td>
<td>-0</td>
<td>0</td>
<td>-0.42</td>
<td>+0</td>
<td>+0</td>
<td>+0</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b3-NoCondType</td>
<td>-15</td>
<td>+409</td>
<td>-0.21</td>
<td>-4</td>
<td>-12</td>
<td>-15</td>
<td>-0.02</td>
<td>-8</td>
<td>-11</td>
</tr>
<tr>
<td>b4-NoKeyVar</td>
<td>-5</td>
<td>+666</td>
<td>+0.15</td>
<td>-5</td>
<td>-5</td>
<td>-2</td>
<td>0.01</td>
<td>-1</td>
<td>-5</td>
</tr>
<tr>
<td>b5-NoKeyAPI</td>
<td>-11</td>
<td>+182</td>
<td>-1.18</td>
<td>-4</td>
<td>-8</td>
<td>-11</td>
<td>-0.01</td>
<td>-4</td>
<td>-7</td>
</tr>
<tr>
<td>b6-NoCallable</td>
<td>0</td>
<td>-4</td>
<td>-0.01</td>
<td>-2</td>
<td>-0</td>
<td>-0</td>
<td>-0.00</td>
<td>-0</td>
<td>-0</td>
</tr>
<tr>
<td>b7-Version2.3</td>
<td>-7</td>
<td>+219</td>
<td>-2.57</td>
<td>-3</td>
<td>-5</td>
<td>-8</td>
<td>-0.01</td>
<td>-3</td>
<td>-5</td>
</tr>
<tr>
<td>b7-Version8.0</td>
<td>-0</td>
<td>+4</td>
<td>-0.01</td>
<td>-2</td>
<td>0</td>
<td>0</td>
<td>-0.00</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

### TABLE V: Statistic of Each ETS Type

<table>
<thead>
<tr>
<th>Version</th>
<th>#ETS</th>
<th>key-Var</th>
<th>key-API</th>
<th>keyVar &amp;keyAPI</th>
<th>No condVar</th>
<th>No outsideVar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ver 2.3</td>
<td>1,643</td>
<td>668</td>
<td>299</td>
<td>217</td>
<td>37</td>
<td>122</td>
</tr>
<tr>
<td>Ver 12.0</td>
<td>13,717</td>
<td>5,129</td>
<td>2,527</td>
<td>1,022</td>
<td>328</td>
<td>619</td>
</tr>
<tr>
<td>Avg 2-12</td>
<td>7,625</td>
<td>2,946</td>
<td>1,245</td>
<td>705</td>
<td>170</td>
<td>356</td>
</tr>
</tbody>
</table>

### TABLE VI: Statistic of Each ETS Type
the first N candidates \( R@N(\%) = \frac{\#R@N}{\text{count}} \), as well as the MRR. From the results, the crashes only relating to keyVars are easier to be located, as most of them exist in the stack. For type No CondVar, we can quickly locate the method which needs to be overridden. And the keyAPI-related crashes and those with unknown outsideVar are difficult to be located with one chance. Overall, the precision improves much on R@1(%) to R@5(%) which indicates the effectiveness of CrashTracker with a compact candidate set. Strategies b4 and b5 are designed to validate the effectiveness of keyVar and keyAPI identification, respectively. All of them decrease the overall precision and increase the sum of the ranking value. In b6, we do not filter the candidate methods that have too many callers. The results show that the filtering will not influence the precision of fault localization. Finally, b7 does not match target ETS from multiple versions and only considers a fixed version. For example, when using the fixed versions 2.3 and 8.0, parts of crashes cannot match their target exception. According to the results, both the ETS construction and the multiple strategies contribute a lot to the precise fault localization.

<table>
<thead>
<tr>
<th>Source Type</th>
<th>Count</th>
<th>R@1(%)</th>
<th>R@5(%)</th>
<th>R@10(%)</th>
<th>MRR</th>
</tr>
</thead>
<tbody>
<tr>
<td>No CondVar</td>
<td>6</td>
<td>0.85</td>
<td>1.00</td>
<td>1.00</td>
<td>0.92</td>
</tr>
<tr>
<td>No OutsideVar</td>
<td>133</td>
<td>0.77</td>
<td>0.92</td>
<td>0.94</td>
<td>0.83</td>
</tr>
<tr>
<td>Only have keyVar</td>
<td>304</td>
<td>0.97</td>
<td>1.00</td>
<td>1.00</td>
<td>0.98</td>
</tr>
<tr>
<td>Only have keyAPI</td>
<td>58</td>
<td>0.74</td>
<td>0.91</td>
<td>0.96</td>
<td>0.82</td>
</tr>
<tr>
<td>Have keyVar, keyAPI</td>
<td>59</td>
<td>0.68</td>
<td>0.98</td>
<td>0.98</td>
<td>0.80</td>
</tr>
</tbody>
</table>

After the self-comparison, we compare the effectiveness of CrashTracker with the state-of-the-art tool Anchor. Table VII gives the R@N(%) and MRR results of Anchor and CrashTracker on the datasets D500 and D69. Overall, CrashTracker achieves a big improvement in precision on both datasets, i.e., achieves 7.4%, 11.5%, and 12.6% improvement than Anchor on R@1, R@5, and R@10 metrics, and improves MRR from 0.84 to 0.91. One special case is the result of R@1 in category B, which also influences the MRR. For these out-of-traces crashes, CrashTracker fails to find out the buggyMethod in the first place compared to Anchor. But if we review the first five candidates, our tool can actually locate more buggyMethods. The reason is that the crashes in category B are usually triggered due to the invocations of keyAPIs relate to the signaler method. The most common usages are pairwise API operations, e.g., register() and unregister(). It is difficult to know whether the register() is redundant or the unregister() is missed without understanding the developers’ intention. Though it brings parts of FPs when only the top one candidate is checked, we provide the complete call paths from each candidate to the signaler in the bug report to help make quick confirmation among multiple candidates.

Finally, we evaluate whether our explainable reports are understandable to testers. As the most difficult part of fault-localization is to identify the out-of-trace buggy method, we manually inspected all the buggy reports in D580 with category B. Two experienced Java developers read both the reports and the apps’ code. The feedback is that among the 56 cases in category B, 12 failed in fault-localization, and the other 44 were reported with reasons. The manual evaluation shows that the reasons for 41 candidates are well understandable for debugging and 3 contain invalid information. Among the 44 reports, reasons with type KeyAPI_Related, KeyVar_Related_2 and Not_Override are the most common, which match 29, 16, and 9 crashes, respectively. There are 18 candidates having multiple types of reasons, which give many-sided explanations about the crash. More details can be found in our artifact [17].

D. RQ3: Precision Analysis

False positive (FP) candidates denote the non-buggyMethods that are reported in the candidate list. For the 580 crashes, we totally get 3,684 candidates, i.e., 3,104 are FPs. There are two key reasons that can lead to FPs. First, our approach uses static analysis to compute the keyConds, based on which we further get the keyVars and keyAPIs. The imprecision in the static analysis, e.g., the FPs in CG construction, will lead to misidentified information in ETS and finally influence the fault localization results. Besides, even though conditions are collected, we did not combine the constraint-solving techniques to reduce the ineffective results, which will be explored in further work. Second, considering there may be misidentified ETS or ETS whose triggering information is unknown, we add the application-level methods in the crash stack as the default candidates conservatively, which also brings some FPs.

False Negative (FN) denotes the buggyMethods that are not in the candidate list. One reason is also the imprecision in the static analysis. Meanwhile, the exceptions with incomplete ETS information, e.g., caught from try-catch blocks or from native methods, may bring FNs. Also, we collect at most three conditions for an exception and filter the candidates whose callee has too many callers. These settings make a balance between efficiency and effectiveness but may cause unexpected FNs. For the 580 cases in D580, we can find 568 buggyMethods and miss 12 ones. Among them, 8 are related to native signaler methods, one suffers from unknown crash message information, and two are missed for lacking implicit data flow relationship, e.g., when the signaler android.widget.Spinner.setAdapter is invoked, another method must be overridden as its default value will lead to a crash.

VIII. Threats to Validity

The threats to external validity related to the generalizability of the experimental results. Although we reuse the two datasets proposed in recent works [21], [27] and add extra crashes
relate to the third-party-SDKs, the data scale is limited and the crashes are not evenly distributed. That is because the keyVar-related misuses are more common, but the keyAPI-related issues are not many, which may be more difficult to find and fix. As our approach is analysis-based, it does not rely on the size of the dataset and achieves a high precision when locating these far-away buggyMethods. We will continuously explore the scalability of CrashTracker on more crash reports when more datasets are publicly available. Threats to internal validity are about the control over extraneous variables. In our collected datasets, the crash-triggering environment is unknown. To keep the randomness, we use the middle version of the matched frameworks, which may bring bias compared with other random strategies. There are several heuristically designed values in candidate ranking that are set according to our experience. Users can adjust them according to their requirements, which does not influence the effectiveness of the tool. Besides, to construct a practical and compact candidate set, we set threshold values during condition collection and caller filtering, which have a weak effect on the results, as evaluated in Table IV (b2 and b6).

IX. RELATED WORK

Crash Trace Based Fault Localization. The crash stack trace is the key element in the crash report, based on which, Chen et al. [19] perform reverse symbolic execution and generate unit test cases. More works [25], [27], [36], [39], [40], [43] use crash stack information to narrow down or locate the buggyMethod. The key challenge is that the stack only contains partially executed methods and may not include the buggy one. So Gu et al. [25] provide an automatic approach to predict whether a crashing-fault resides in a stack trace or not, which denotes the existence of the out-of-stack buggyMethods. To locate them, CrashLocator [40] tries to recover the complete execution trace by CG extension on Java projects. However, without code separation and summary construction, CrashLocator may suffer from a large candidate set or low precision when handling framework-specific crashes. Compared with it, CrashTracker weakly relies on the CG but relies more on the extraction of keyVars and keyAPIs. To perform stack-trace-based fault-localization on Java programs, both Sinha et al. [35] and Ginelli et al. [24] focus on the semantic of exceptions. However, they do not analyze the real exception-thrown points in the frameworks and require manual modeling of specific exceptions, which limits the scalability. For this work, we first perform an automatic semantic analysis for all kinds of framework-level exceptions without any manual modeling. The extracted summaries can help the application-level analysis be more targeted.

To address the problem of Android framework-specific fault localization, researchers combine the learning-based approach with the stack-based analysis. By learning from similar faults, ExLocator [22] first classifies a crash trace within given exception types, and then generates the root causes by static analysis on target applications. This tool is not publicly available and only focuses on the given 5 exception types. To support more types, Anchor [27] collects a general dataset with 500 crashing reports for model training and testing. For a crash trace, it first predicts whether the buggyMethod exists on the crash stack, then sorts candidates according to the previous classification result. However, it relies on the labeling of a large-scale dataset, which can not completely cover the numerous and quickly evolving Android framework exceptions. Compared to it, our tool does not need any prior knowledge of crash fixing and works well for newly detected exceptions.

Analysis upon Pre-computed Summaries. Considering the large size of the framework code, a set of works focuses on how to make an analysis based on the pre-computed summaries. Some works [18], [32] noticed that the large-scale framework hinders the inner call relations and hinders the program understanding and debugging. Among them, Cao et al. [18] detect the implicit control flow transitions through the Android framework. Besides, Perez et al. [32] generate predicate callback summaries for the Android framework and sequence the callbacks. And recently, Samhi et al. [33] try to find out all the atypical methods around ICC in the Android framework. To achieve this goal, they retrieve the framework code with a lightweight static analysis. Though these works are framework-summary-related, none of them focus on exceptions of the Android framework and their summary could not be used as an effective specification to represent the exception-triggering information.

X. CONCLUSION

The post-release crashes are inevitable for application developers, which require a heavy effort of debugging and fixing. In this paper, we adopt a code-separation-based analysis approach to solve the Android framework-specific fault localization problem. We design a novel specification, ETS, for framework exceptions, and propose an effective crash-localization approach. By applying ETS on real crash stack traces, CrashTracker outperforms the state-of-the-art tool with higher precision and fewer candidates. Moreover, our approach is explainable in that it gives reasons for each recommended candidate method.

ACKNOWLEDGMENTS

We are grateful to Jian Zhang and Yajun Zhu for their comments on earlier drafts of this paper, and to Pingfan Kong for providing apps used in [27]. Also, thanks to the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant No. 62102405 and Grant No. 62132020) and the Guangdong Basic and Applied Basic Research Fund (Grant No. 2021A1515011562).

REFERENCES


