Using Crash Frequency Analysis to Identify Error-prone Software Technologies in Multi-System Monitoring

Andreas Schörgenhumer*, Mario Kahlhofer*, Hanspeter Mössenböck† and Paul Grünbacher‡
* Christian Doppler Laboratory MEVSS, Johannes Kepler University Linz, Austria
andreas.schoergenhumer@jku.at
† Institute for System Software, Johannes Kepler University Linz, Austria
‡ Institute for Software Systems Engineering, Johannes Kepler University Linz, Austria

Abstract—**Faults are common in large software systems and must be analyzed to prevent future failures such as system outages. Due to their sheer amount, the observed failures cannot be inspected individually but must be automatically grouped and prioritized. An open challenge is to find similarities in failures across different systems. We propose a novel approach for identifying error-prone software technologies via a cross-system analysis based on monitoring and crash data. Our approach ranks the error-prone software technologies and analyzes the occurred exceptions, thus making it easier for developers to investigate cross-system failures. Finding such failures is highly advantageous as fixing a fault may benefit many affected systems. A preliminary case study on monitoring data of hundreds of different systems demonstrates the feasibility of our approach.**

I. Introduction

It has been shown that large-scale software systems tend to fail more often due to their higher complexity [1], [2]. Such failures can lead to crashes, often with dramatic impact. System outages, downtime and performance degradations can potentially lead to bad reputation or financial loss of the service provider of a system [1]. Researchers have thus been developing approaches for analyzing crashes (e.g., [3]–[5]). Since systems will grow even more in the future, the number and impact of crashes can also be expected to increase, making a manual analysis infeasible. Instead, a promising idea is to find similarities among crashes by bucketing, grouping or classifying crash reports. This can be used to prioritize crashes, e.g., by their frequency, so that developers can first fix the issues with the highest impact. Previous research in this area focused on inspecting crashes, either occurring on a specific platform such as Android [6] or in single software systems [7]–[14]. Research so far often concentrated on a specific product (e.g., Mozilla Firefox) or product family (e.g., Microsoft products such as Windows and Office).

However, analyzing failures in today's large-scale software systems is particularly challenged by the diversity of programs and processes, which are executed on heterogeneous technologies. Moreover, different versions of programs or components are frequently deployed to different environments, thereby creating a vast and diverse landscape of software technologies. In such a context, crash analysis thus needs to go beyond the boundaries of individual systems. Our aim is to find commonalities of crashes in such multi-system landscapes, allowing to identify and fix problems potentially affecting multiple systems. Hence, we are particularly interested in frequently happening crashes which are not specific to only one service provider, i.e., crashes that occur in multiple systems. Since large numbers of crashes may occur, crash prioritization becomes essential.

We use anonymized system and crash data from a monitoring infrastructure of an industry partner. Rather than inspecting the crashes individually, we look for common properties of the crashed *processes*. Such properties are, for example, the frameworks or technologies (e.g., Java, .NET, PHP) used by these processes. We automatically create technology tuples and connect them to the occurred crashes. We consider a technology tuple as a set of framework types and versions. For example, `((Java, 1.8), (Tomcat, 8.0))` is a technology tuple with two elements. We then merge the tuples across all systems and apply a user-definable ranking metric, allowing us to reveal error-prone software technologies. We consider a software technology as error-prone not only if it has faults, but also if it is incorrectly used. This is important as bugs are frequently introduced when integrating a particular software technology in a system. The resulting top ranked tuples are further analyzed and grouped by exceptions, yielding the final targets for manual investigation. An engineer can then browse through these targets to determine the root cause of the crashes. In case a common solution is found, all affected service providers can then be notified to benefit from a fix for such common crashes.

Our paper claims the following contributions:

(i) We describe a novel approach for aggregating and prioritizing crashes that occur in heterogeneous systems by automatically identifying error-prone software technologies via merging, ranking and grouping system monitoring data as well as crash data. The goal is to highlight common crashes that occurred in multiple systems where fixing one such crash may benefit many affected service providers.

(ii) We provide a preliminary case study based on hundreds of different systems with thousands of processes and crashes.

We show that our cross-system crash analysis is capable of identifying common problematic cases for which a solution can potentially be applied to every affected service provider.

The rest of this paper is organized as follows: Section II describes the monitoring and crash data required for our approach. Section III presents our cross-system crash analysis. In Section IV, we apply our approach on a large dataset and highlight identified problematic cases. In Section V, we discuss related work and Section VI concludes this paper.

II. Monitoring and Crash Data

Our research is conducted in collaboration with an industry partner. However, it is independent of a specific monitoring infrastructure. We assume that the following data about processes and crashes will be collected from the monitored systems, allowing our approach to identify error-prone software technologies based on common crashes (cf. Table I, II):

A. Processes

Data need to be continuously collected about important properties and communication connections of all processes executed by the monitored systems. Process properties include information about the used software technologies together with timestamps indicating if the process and the used technologies are active. Connection properties describe all process-to-process communications and also allow to create a communication graph.

TABLE I
Example Properties of a Process and Software Technologies

Process: TomcatServerA			
Start		End	
1491631002730[1]		1499218873887	
Software Technologies			
Type	Edition	Version	Active Until
Java	OpenJDK	1.8.0_121	1495535781954
Tomcat	-[2]	7.0.65.0	1495535781954
Java	OpenJDK	1.8.0_131	1499218873887
Tomcat	-	8.0.44.0	1499218873887

[1] All timestamps are in milliseconds of UTC.
[2] The character "-" indicates a missing entry.

Example properties of a Tomcat server process hosting a web service are listed in Table I. The start and end timestamps of the process indicate the points in time when the process was first launched and when it was terminated. The software technologies encompass the following information: the type of the technology (e.g., Java), an optional edition (e.g., OpenJDK), an optional version (e.g., 1.8.0_121), and a timestamp indicating until when the particular technology was active on the process. As can be seen from the timestamps in the example, an older version of Apache Tomcat was running on Java until both were updated. The new software technologies then were executed until the termination of the process.

B. Crashes

Crash data contain properties of all occurred failures in the monitored systems. They contain the process causing a crash, a timestamp, as well as the exception or error message. For example, a programming error in the Tomcat source code could have led to a server crash, which is described by the properties shown in Table II. The timestamp indicates that the crash occurred while the software technologies `(Java, OpenJDK, 1.8.0_121)` and `(Tomcat, 8.0.44.0)` were active.

TABLE II
Example Crash Properties

Process	Exception	Timestamp
TomcatServerA	java.lang.NullPointerException	1498540000391

Our approach is not limited to exceptions. If other properties are stored for each crash (e.g., the top stack trace frame or the location where the crashed process was executed), we can utilize this information as we will show in Sections III-D and IV.

III. Approach

Our automated approach extracts information about software technologies used at crash points to reveal suspicious technologies likely leading to system failures. It comprises four steps: (i) It first creates a history of system topology snapshots by preprocessing the monitoring data described in the previous section. (ii) It then creates software technology tuples based on an analysis of the occurred crashes. (iii) It ranks the software technology tuples to reveal the most suspicious ones; and (iv) performs a detailed analysis of the crash properties.

A. Creating System Topology Snapshots

The raw data described in the previous section need to be preprocessed and transformed to a representation facilitating the subsequent analyses. In this step, we thus build a snapshot-based topology for each system, i.e., process graphs that also include associated crash events. The nodes in a graph represent processes that were active at the time of the snapshot, while the edges indicate communication links between the processes, i.e., a graph could be disconnected. In order to capture the evolution of the system topologies over time, we create multiple process graphs for each system. In this way, we can represent structural changes that happen when a new process starts or when an existing process terminates.

B. Crash Frequency-based Creation of Software Technology Tuples

Our approach aims to extract information about software technologies at crash points to reveal error-prone technologies. Therefore, we create so-called software technology *tuples* that capture the technologies used during the lifetime of a process. We determine how often a tuple was seen at a crash point and how often it was active at a point where the process did not crash. We also capture the processes and systems that were affected and the crash events that were recorded.

Specifically, our approach currently encompasses two types of tuples:

1-tuples are created by extracting every single software technology of each process. For example, if a process A has two technologies X and Y, we create two 1-tuples, namely (X) and (Y).

2-tuples are created based on the process graphs by considering pairs of neighboring, i.e., communicating, processes. We create all possible technology pairs for each process pair and their active software technologies. Figure 1 shows an example of two neighboring processes A (with software technologies X, Y) and B (with software technology Z). The timeline shows the timestamps of both processes: B started at t_1 and ended at t_4, whereas A started at t_2 and ended at t_5. The timestamps for the software technologies show how long they were active: X was active from the start of $A = t_2$ until t_3 while Y was then active until the end of $A = t_5$. Software technology Z was active the entire time. In this example, we would create two 2-tuples, namely (X, Z) and (Y, Z). In case process B would start after timestamp t_3, we would create only one 2-tuple, namely (Y, Z) because technology X would no longer be active.

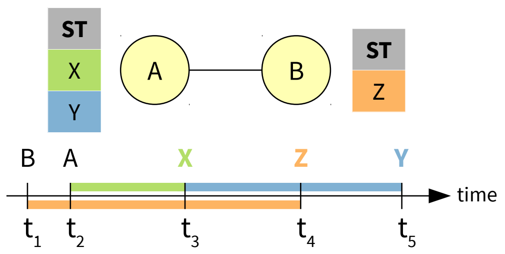


Fig. 1. A 2-tuple example showing the timelines of three software technologies (ST) on two processes.

We analyze 2-tuples as we want to investigate possible software incompatibilities, i.e., crashes might occur more frequently if specific (versions of) technologies communicate with each other. Such misconfigurations have been identified as a significant cause of system failures [15]. Obviously, our approach could also handle n-tuples as we will discuss in Section IV-D.

For every created tuple, we check if it crashed during its lifetime, i.e., the time span from activating to deactivating its software technologies. We consider a tuple as crashed if at least one of the involved processes crashed during its lifetime. Otherwise, the tuple is considered as not crashed. In both cases, we record to which system a tuple belongs, yielding the following tuple result r:

- *r.techs:* The actual software technology tuple, e.g., (X, Y).
- *r.crashed:* 1 if (at least one of) the process(es) running the *techs* crashed, 0 otherwise.
- *r.events:* The set of all crash events of process(es) where *r.techs* was active. The set is empty if no crash occurred.
- *r.system:* The monitored system of the service provider hosting the process(es).

After storing all results in a database, our approach next merges tuples in each system by creating buckets of similar versions, thereby compacting the version information recorded for the software technologies. Specifically, we remove the last version token but keep at least two tokens (e.g., for the version information 1.8 we keep 1.8 while for 7.0.65.0 we shorten it to 7.0.65). This merging steps improves our prioritization results, while still providing sufficiently detailed information. Afterwards, we aggregate the results for equal technology tuples. For $\geq$ 2-tuples, equality means that two tuples are either exactly equal or “reverse-equal”, i.e., (X, Y, Z) is considered equal to (Z, Y, X) since it is in the reverse order, while (X, Z, Y) would be a different tuple. This means that we form groups G of equal software technologies of the results r (with the compacted version field) and create the new merged results m for each such group G as follows:

- $m.techs = r.techs$, arbitrary $r \in G$
- $m.crashed = \sum_{r \in G} r.crashed$
- $m.\neg crashed = \sum_{r \in G} (1 - r.crashed)$
- $m.events = \cup_{r \in G} r.events$
- $m.system = r.system$, arbitrary $r \in G$

After the merging process, *crashed* and *¬crashed* indicate how often a specific tuple crashed (respectively did not crash) in the observed time period of the monitored system.

In the last aggregation step of tuple processing, we merge the results for all the different systems. Again, we form groups G of equal software technologies of the merged results m and create new final results f for each such group G as:

- $f.techs = m.techs$, arbitrary $m \in G$
- $f.crashed = \sum_{m \in G} m.crashed$
- $f.\neg crashed = \sum_{m \in G} m.\neg crashed$
- $f.crashedSystems = \cup_{m \in G \wedge m.crashed > 0} \{m.system\}$
- $f.\neg crashedSystems = \cup_{m \in G \wedge m.\neg crashed > 0} \{m.system\}$

The new entries *crashedSystems* and *¬crashedSystems* indicate in which systems the particular tuple crashed or did not crash respectively.

C. Ranking of Software Technology Tuples

After computing these tuple results, the approach aims to extract the most “interesting” tuples, i.e., the ones revealing software technologies that likely result in failures. We aim to find tuples that frequently led to crashes in many different systems. We defined the following metric to calculate a rank for each final result tuple f:

$$\text{rank}(f) = \frac{|f.cS|}{|f.cS| + |\neg f.cS|} \cdot \frac{|f.cS|}{\max_{f' \in F}(|f'.cS|)} \cdot f.c$$

where $|\cdot|$ is the number of elements of the set, cS stands for *crashedSystems*, c for *crashed* and F is the set of all final results. The first factor is the ratio of crashed systems to all systems in which this tuple occurred. The second factor is a scaling factor that rewards tuples with many crashed systems. If this factor was missing, tuples with few crashed systems and zero not-crashed systems would be ranked high despite the low total number of systems, which is contrary to what we defined as interesting. The third factor simply ranks tuples higher if they crashed more frequently, thus ensuring a focus on reoccurring crashes rather than one-time crashes.

D. Crash Property Analysis

After applying the ranking metric, we sort the tuple results in descending order and analyze the top tuples in more detail to find crash commonalities. Specifically, we first group and sort the data by the reported crash exceptions to yield groups of equal exceptions. We then divide these groups based on the individual systems where the exception occurred. For every tuple, we visualize the final result as a bar plot showing all crashes, first grouped by their exceptions and then by their originating systems.

As discussed above the most interesting results are tuples with many crashes that occurred in multiple systems. Engineers can then manually investigate such cases by inspecting the affected systems, the crashes and the processes affected by the crashes. They may need to use additional data to identify the crash's root cause and to develop a fix for affected service providers.

As mentioned in Section II, we can also apply the grouping algorithm on any other crash property. For instance, if the top stack frame is also stored for each crash, we can obtain plots showing crash groups of equal top stack frames, again divided by the originating systems. If crashes have no exception property we cannot directly apply our grouping in the last processing step. However, we can treat such cases as "empty" exceptions rather than no exception at all. This way, we can still see these crashes in the final plot and, in case of interest, inspect them in more detail.

IV. Evaluation

The goal of our approach is to identify error-prone software technologies across multiple systems by automatically finding common crashes potentially affecting multiple service providers. In our preliminary evaluation we investigate the following research questions to evaluate the usefulness of the approach:

RQ1. Is the *automated analysis* capable of finding error-prone software technologies? – We performed a 1-tuple analysis and also analyzed crash properties across systems based on an industrial dataset.

RQ2. Are the results found by the crash property analysis meaningful? – We conducted a *manual inspection* of selected results to determine whether our approach can help in identifying common cross-system crashes.

RQ3. Does the consideration of *process communication* yield additional information? – We performed a 2-tuple analysis and compared the results to the 1-tuple analysis.

We analyzed systems from our industry partner's environment. These systems are completely independent of each other and their sizes range from small systems with only a few processes to huge and complex systems with hundreds of machines and thousands of processes. The dataset was created to cover a longer period of time with more likely changes to systems and technologies. Our dataset ranges from April 2017 to March 2018. To keep the size of the dataset manageable, we included the first week of each month. The average number of crashes was approximately 18,000 crashes per month, the average number of systems was about 500.

A. RQ1 – Automated Analysis

To check whether our approach can find problematic error-prone cross-system technologies, we first performed a 1-tuple analysis as defined in Section III-B and then analyzed the exceptions of the occurred crashes. Figure 2 shows characteristic examples of top ranked 1-tuples, grouped by the exception crash property. Obviously, these examples show only a small subset. As mentioned before, we specifically looked for exception groups with a high number of crashes occurring in multiple systems. Visually speaking, these groups result in large multicolored bars, with each color representing a system. The x-axis of the plots shows the total number of occurred crashes and the y-axis represents the exception groups in descending order of the crash count. To keep the plots readable, we removed exceptions that only occurred in one system and we only kept a maximum of seven exception groups.

For instance, Figure 2a shows the bar plot for the highest ranked tuple of the October export: `(CLR, 4.0.30319)`, which is part of the .NET Framework 4.7 (CLR = Common Language Runtime). The `TargetInvocationException` was thrown most often but only in two systems, whereas the `SqlException` occurred nearly as often but in four different systems. The `TypeInitializationException` occurred in eight systems. The next two exceptions are also of interest, as they occurred in three and six systems.

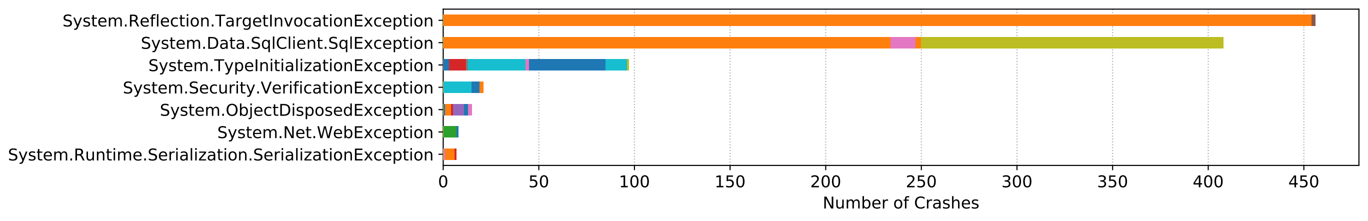
Analogously, one can interpret the results in Figure 2b showing the tuple `(CLR, 2.0.50727)`, which is part of the .Net Framework 3.5. Clearly, the `AppDomainUnloadedException` with six systems and over 100 crashes is worth to be investigated in more detail.

In Figure 2c, we can see the highest ranked tuple of the month March, this time displayed using a logarithmic scale because of the high crash count. After the `TypeInitializationException`, which was thrown in 42 systems, the `SerializationException` causing the second-most crashes occurred in five different systems, followed by the `NullReferenceException` (seven systems). The `MemoryException` and the `ObjectDisposedException` could be worth investigating as well since they were thrown in nine and five different systems.

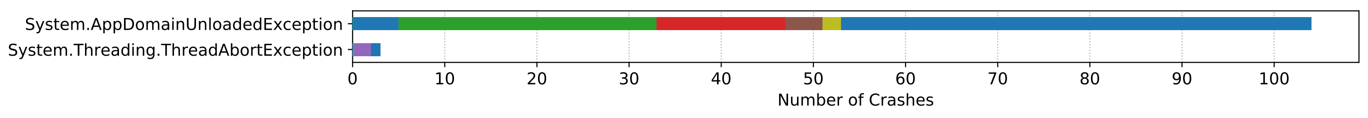
An engineer can use such visualizations to select cases for further inspection to determine any possible common root causes or to discard irrelevant ones.

B. RQ2 – Manual Inspection

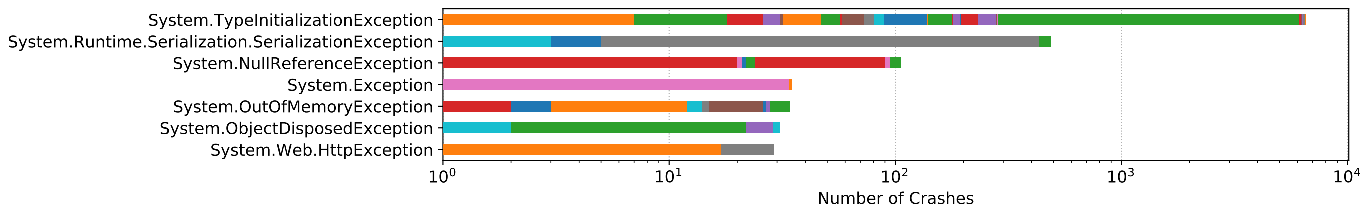
We now demonstrate that the automatically found crashes are indeed meaningful. We illustrate this process by looking at



(a) October 2017: exception groups of the 1st ranked 1-tuple `(CLR, 4.0.30319)`.



(b) October 2017: exception groups of the 13th ranked 1-tuple `(CLR, 2.0.50727)`.



(c) March 2018: exception groups of the 1st ranked 1-tuple `(CLR, 4.0.30319)` in logarithmic scale.

Fig. 2. Examples of top ranked 1-tuples, grouped by the exception crash property. The colors represent different systems.

the example in Figure 2b in more detail to confirm the validity of the results. Specifically, we more closely investigated the systems affected by the `AppDomainUnloadedException` based on the results of our October analysis. First, we searched on the Internet for possible causes and found a handful of related issues. These included bugs in SQLite and NUnit, switching the test runner from MSTest Runner to Visual Studio Test Runner and switching between versions of the .NET framework. We then continued by looking at the crashed processes of the individual systems to find similarities or correlations. As it turned out, most of the crashes occurred in the *ReportingServicesService.exe* within a suspicious 12-hour interval. Further investigations revealed that the crashes were most likely due to Microsoft's *Reporting Services*, which by default recycled application domains every 12 hours [16].

The example shows a problem potentially affecting many service providers, who would benefit from the findings and possible solutions.

C. RQ3 – Process Communication

Our approach either performs a 1-tuple analysis considering the software technologies of individual processes or a 2-tuple analysis also considering pairs of communicating processes. We also explored the benefit of additionally considering communications between processes. Specifically, the 1-tuple approach is expected to yield *single* error-prone software technologies, while the 2-tuple analysis can potentially also discover suspicious technology *pairs*. In such a case, the 2-tuple would be ranked higher than the two individual 1-tuples.

Before we discuss the comparison results in Figure 4, we explain how to read the plot using the small example in Figure 3. The x-axis represents the rank position of the two top ranked 2-tuples while the y-axis represents the difference to the higher ranked corresponding 1-tuple. The rank position is the index of the tuple after sorting the tuples by rank in descending order. In Figure 3, there are two 2-tuples: The 1st ranked 2-tuple has a difference of -3, the second one has a difference of $+1$. Here is how these differences were calculated:

- Suppose the two 2-tuples are (X, Y) and (Y, Z). They have rank positions rp of $rp_{(X,Y)} = 1$ and $rp_{(Y,Z)} = 2$ (directly taken from the x-axis of the plot).
- The corresponding 1-tuples are (X), (Y) and (Z). Suppose they have $rp_{(X)} = 4$, $rp_{(Y)} = 5$ and $rp_{(Z)} = 1$.
- For each 2-tuple, the difference d is now calculated by subtracting the higher ranked 1-tuple rp from the 2-tuple rp. For (X, Y), the higher ranked 1-tuple is (X), which yields $d = rp_{(X,Y)} - rp_{(X)} = 1 - 4 = -3$. For (Y, Z), the higher ranked 1-tuple is (Y), which yields $d = rp_{(Y,Z)} - rp_{(Y)} = 2 - 1 = 1$.

The difference indicates which tuple is considered as more relevant. A positive difference means that the 1-tuple is more relevant, a negative one means that the 2-tuple is more relevant.

Figure 4 shows a comparison for each monitored month. As can be seen in the figure, the 1-tuple results were in general considered as more relevant. With a few exceptions, most top ranked 2-tuples just contained a top ranked 1-tuple (or even two), indicating that analyzing technologies in pairs did not offer additional information, i.e., in most of the cases, a single component could be blamed for the failure. The

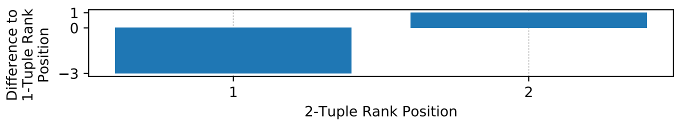


Fig. 3. Example comparison of 2-tuples to 1-tuples.

increasing trend in each subplot just shows that most of the 2-tuples shared a high ranked 1-tuple, e.g., if (X) was at a very low rank position, then (X, A), (X, B), (X, C) would yield monotonically increasing differences. The outliers did not yield interesting insights either. Their rank differs for two reasons: i) The error-prone 2-tuple only occurs within a single system. ii) The fault can only occur in a system with those two tuples (e.g., a web server and its backend), i.e., the 2-tuple outlier indicates a common technology correlation. Neither case shows a general software incompatibility.

D. Further Opportunities

Our preliminary evaluation demonstrated the feasibility of our approach for a common scenario. However, our approach can be adapted and customized in different ways. This includes choosing the definition of the ranking metric, the use of alternative properties for the crash property analysis, as well as the kind of tuple analysis. This raises interesting questions for further research, which we discuss in the following.

Using different ranking metrics: The ranking metric is responsible for extracting the top tuples from the recorded raw data. It thus allows full control of what a user considers as interesting. Equation (1), for example, drops the crash count in our original metric to set the focus even more on different systems. Conversely, the metric could also discard all information on the systems and solely reflect the crash count, as it is done in (2).

$$\text{rank}_a(f) = \frac{|f.cS|}{|f.cS| + |\neg f.cS|} \cdot \frac{|f.cS|}{\max_{f' \in F}(|f'.cS|)} \quad (1)$$

$$\text{rank}_b(f) = \frac{|f.c|}{|f.c| + |\neg f.c|} \quad (2)$$

As different metrics imply different top tuples, this naturally changes the 2-tuple vs 1-tuple scenario, which is shown in Figure 5 for March 2018. Of course, this requires further investigation in an in-depth evaluation.

Considering alternative crash properties: As mentioned in Section II, we can also make use of different crash properties if they are available. For instance, we could create groups based on common *class names* where the crashes occurred. We could also look at *process signals* or the *fault location*. The advantage of utilizing additional crash information is the increased insight we get into the crash data, which helps to filter actually relevant crashes and allows for a faster root cause investigation. An example is shown in Figure 6. The grouping algorithm could also be extended to allow custom equality measures instead of exact matches only. This would prove useful in case of more complicated crash properties such as stack traces, where some sort of stack trace similarity measure (cf. [8], [9], [11], [17], [18]) would then determine equal groups in the plots.

Performing an n-tuple analysis: Based on our 2-tuple to 1-tuple comparison, we concluded that investigating n-tuples with $n \geq 3$ might not be worth the effort as already 2-tuples did not provide significantly more information. Moreover, we argue that failures originating from software

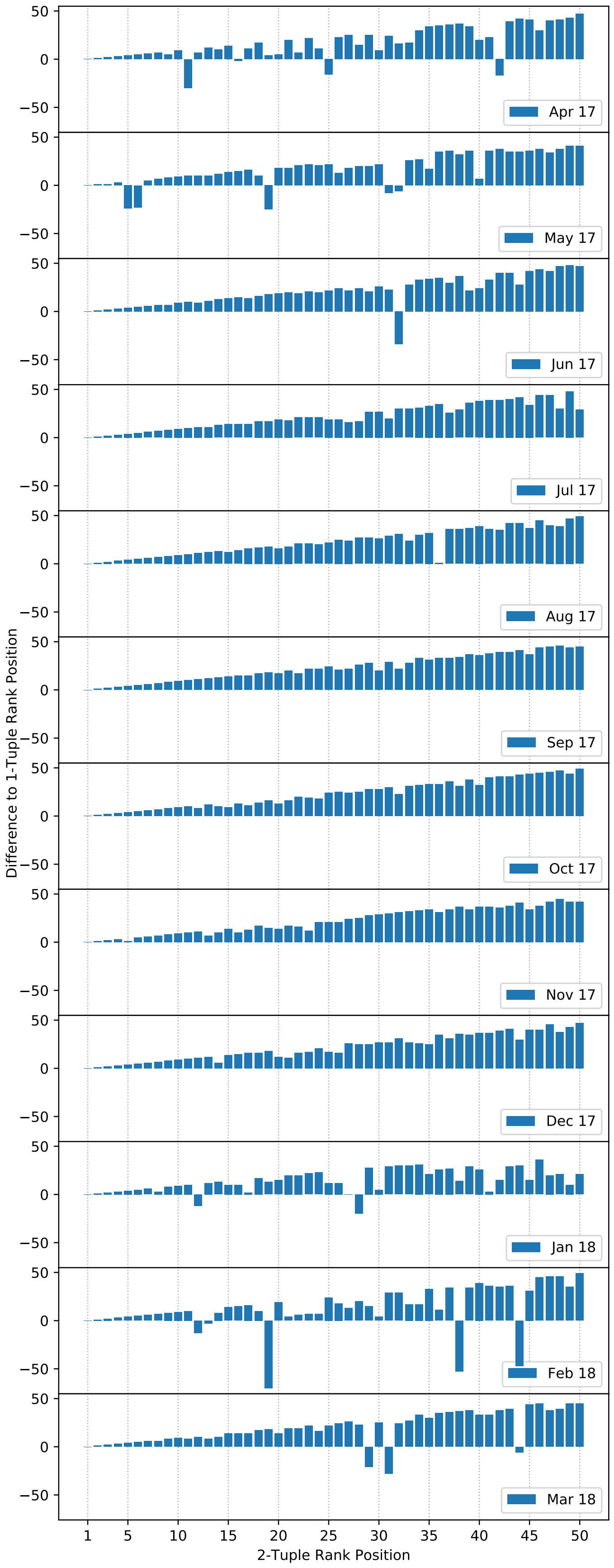


Fig. 4. Comparison of 2-tuples to 1-tuples.

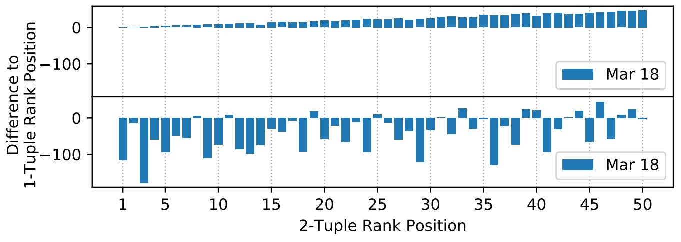


Fig. 5. Comparison of 2-tuples to 1-tuples using (1), upper plot part, and (2), lower plot part.

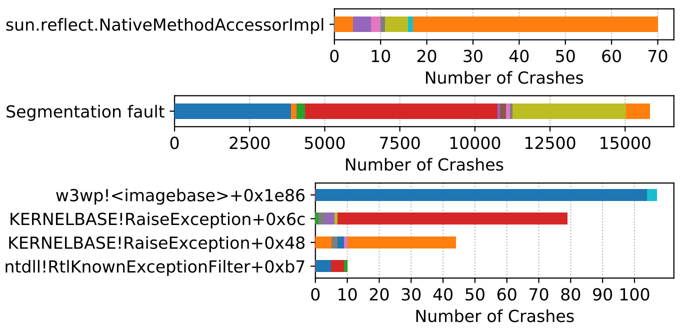


Fig. 6. Example groups for three alternative crash properties: class name (top), process signal (middle) and fault location (bottom).

incompatibilities are typically already covered by 2-tuples, since such incompatibilities are much more likely to occur in direct communication rather than over multiple hops. However, it must be noted that the observation in Figure 4 only holds for the data we analyzed and for the ranking metric we defined in Section III-C. Future data or data from other monitored systems and different metrics might yield other results, so we will consider data, metrics and n-tuples in future work.

V. Related Work

Error analytics is a huge area of research. For instance, many valuable research contributions have been made in the field of software bug management (e.g., optimization, triaging, automatic fixing of bugs) [3]–[5]. The work by Ghafoor and Siddiqui [18] is probably closest to our approach for analyzing and prioritizing cross-system crashes. Specifically, the authors try to find the same bugs for different programs or products (e.g., Mozilla Firefox, Thunderbird, MailNews). For that purpose, they utilize bug reports from the corresponding bug repositories via the Bugzilla API from where they extract stack traces, which they group according to the information available in the bug reports. Since stack trace matching is a promising approach [17], [19] they use a string-based similarity measure to cluster similar bugs. They evaluated their approach on 132 programs. A manual inspection of the resulting clusters revealed that the clustered bug reports are indeed correlated, i.e., fixing a bug in such a cluster also fixes the same problem in different programs.

Classifying and bucketing crashes and bugs or bug reports for prioritization has been researched in recent years, however, existing research mainly focuses on single applications or families of products. Dhaliwal et al. [9] present a classification of Mozilla Firefox crash reports based on stack trace string similarity. Glerum et al. [10] introduce the Windows Error Reporting (WER). The authors collect debugging data from Microsoft products and assign bugs to buckets using different labeling heuristics (program name, version, timestamp, exception code, etc.) as well as classifying-heuristics (stack matching, prioritizing stack frames, etc.). Kim et al. [12] create weighted crash graphs based on stack traces of the crashes bucketed by WER and compute a graph similarity measure to classify crashes. ReBucket [8] is another improvement of WER: The authors additionally analyze the stack trace similarity via their Position Dependent Model, which considers the number of functions, the distance of those functions to the top stack frame, as well as the offset distance between matched functions. A different approach was proposed by Cui et al. [7], who consider program semantics by extracting additional information from crash memory dumps to identify bad functions for crash bucketing. Wang et al. [14] even investigate crash type correlations between groups of equal or similar crash reports via structural (stack traces), temporal (crashes that occurred at similar timestamps) and semantic information (textual similarity of user comments). The concepts and ideas presented in these papers are useful to complement our crash event analysis in the future. Especially stack traces would be a great addition to our approach to improve the automatic grouping algorithm and to support the manual inspection, as stack traces provide great help in finding bugs [20]. Regarding manual inspection, log messages could be useful as already shown in other work [21], [22].

The approach by Murtaza et al. [13] is based on function call traces. They specifically look at recurring faults in different versions of a deployed program and try to detect faults in older versions of a system for which a fix already exists in newer versions. Under certain circumstances, our approach can also identify such cases, e.g., when a specific software technology used in multiple systems contains faults. Recurring crashes are also the main focus of Gao et al. [6], who try to automatically apply known fixes found in Q&A sites (e.g., Stack Overflow) by partial stack trace matching. Incorporating such an approach into our own approach could be interesting, most notably to automatically gather information for the identified problems, which is currently a manual activity.

Predicting whether new crash events should be classified as relevant or not was investigated by Kim et al. [11]. They prioritize crashes by applying machine learning on previous crashes that were labeled as important or unimportant. This could also be a valuable addition to our current system, as it would allow to immediately classify new crashes as essential. There is also recent research on predicting defects in new projects based on defect data from existing projects [23], [24]. Our research focus so far was different, i.e., analyzing occurred crashes for finding fixes as opposed to predicting future defects for prevention. However, it would be interesting to investigate if ideas from cross-*project* defect prediction could also be applied to cross-*system* crash prediction.

As the detection of operational anomalies in continuous monitoring is an essential task in DevOps [25], our approach can also be used to complement and enhance existing tool pipelines.

VI. Conclusions and Future Work

In this paper, we presented an approach for a cross-system crash analysis to identify error-prone software technologies across different service providers. We focused on finding similar or equal crashes that occurred in multiple systems, where fixing one such crash can potentially benefit many affected service providers. For this purpose, we extracted tuples of software technologies of the monitored systems' processes and tracked whether and how often they crashed. Our approach applies a user-defined ranking metric which yields the top error-prone software technologies, and then analyzes these technologies in more detail by grouping the occurred crashes by their exceptions. This results in exception groups per error-prone software technology. These groups ease the selection of cases that can be inspected manually to find the root cause of crashes and, ideally, to provide common fixes or solutions.

Based on the data of hundreds of different systems, we investigated three research questions in a preliminary evaluation covering a time period of 12 months. In this study, we executed a 1-tuple analysis and also performed a manual root cause investigation for a selected crash-property grouped result. We further compared 2-tuples to 1-tuples. Our results indicate the usefulness and the applicability of our new approach and show opportunities for further research.

We expect that utilizing additional data such as stack traces or logs will further improve our approach and provide additional insight into cross-system analysis. The manual investigation could also be supported by automatically gathering relevant data from crashes, processes of the monitored systems or even by suggesting possible hints or solutions retrieved from online searches. Moreover, the results could be used to populate a knowledge base of found fixes and solutions, allowing to quickly provide remedies for recurring crashes. Taking temporal properties into account might also be an interesting path of future work. This could include detecting patterns in crashes, their properties or even entire processes, systems or specific data thereof.

Acknowledgment

This work was supported by the Christian Doppler Forschungsgesellschaft.

References

[1] R. N. Charette, "Why software fails [software failure]," *IEEE Spectrum*, vol. 42, no. 9, pp. 42–49, 2005.

[2] E. E. Ogheneovo, "Software dysfunction: Why do software fail?" *Jrn. of Computer and Communications*, vol. 2, no. 06, pp. 25–35, 2014.

[3] J. Zhang, X. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Science China Information Sciences*, vol. 58, no. 2, pp. 21 101–021 101, 2015.

[4] T. Zhang, H. Jiang, X. Luo, and A. T. Chan, "A literature review of research in bug resolution: Tasks, challenges and future directions," *The Computer Journal*, vol. 59, no. 5, pp. 741–773, 2016.

[5] J. Uddin, R. Ghazali, M. M. Deris, R. Naseem, and H. Shah, "A survey on bug prioritization," *Artificial Intelligence Review*, vol. 47, no. 2, pp. 145–180, Feb. 2017.

[6] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing Q&A sites," in *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 307–318.

[7] W. Cui, M. Peinado, S. K. Cha, Y. Fratantonio, and V. P. Kemerlis, "RETracer: Triaging crashes by reverse execution from partial memory dumps," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. ACM, 2016, pp. 820–831.

[8] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "ReBucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proceedings of the 34th Int'l. Conf. on Software Engineering*, ser. ICSE '12. IEEE Press, 2012, pp. 1084–1093.

[9] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox," in *Proc. of the 27th Int'l. Conf. on Software Maintenance (ICSM)*. IEEE, 2011, pp. 333–342.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt, "Debugging in the (very) large: Ten years of implementation and experience," in *Proc. of the 22nd Symp. on Operating Systems Principles*. ACM, 2009, pp. 103–116.

[11] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 430–447, 2011.

[12] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Proceedings of the 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 486–493.

[13] S. S. Murtaza, N. H. Madhavji, M. Gittens, and A. Hamou-Lhadj, "Identifying recurring faulty functions in field traces of a large industrial software system," *IEEE Transactions on Reliability*, vol. 64, no. 1, pp. 269–283, 2015.

[14] S. Wang, F. Khomh, and Y. Zou, "Improving bug management using correlations in crash reports," *Empirical Software Engineering*, vol. 21, no. 2, pp. 337–367, Apr. 2016.

[15] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *Proc. of the 23rd Symposium on Operating Systems Principles*, ser. SOSP '11. ACM, 2011, pp. 159–172.

[16] Microsoft, "Application domains for report server applications," Mar. 2017. [Online]. Available: https://msdn.microsoft.com/en-us/library/bb934330.aspx

[17] M. Brodie, S. Ma, L. Rachevsky, and J. Champlin, "Automated problem determination using call-stack matching," *Journal of Network and Systems Management*, vol. 13, no. 2, pp. 219–237, 2005.

[18] M. A. Ghafoor and J. H. Siddiqui, "Cross platform bug correlation using stack traces," in *Proceedings of the International Conference on Frontiers of Information Technology (FIT)*. IEEE, 2016, pp. 199–204.

[19] N. Modani, R. Gupta, G. Lohman, T. Syeda-Mahmood, and L. Mignet, "Automatically identifying known software problems," in *Proceedings of the 23rd International Conference on Data Engineering Workshop*. IEEE, 2007, pp. 433–441.

[20] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *Proceedings of the 7th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2010, pp. 118–121.

[21] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: Error diagnosis by connecting clues from run-time logs," in *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. ACM, 2010, pp. 143–154.

[22] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Proceedings of the 16th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. ACM, 2011, pp. 3–14.

[23] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Transactions on Software Engineering*, vol. PP, no. 99, pp. 1–40, 2017.

[24] F. Porto, L. Minku, E. Mendes, and A. Simao, "A systematic study of cross-project defect prediction with meta-learning," *arXiv preprint arXiv:1802.06025*, 2018.

[25] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables DevOps: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016.