Risk-Based Attack Surface Approximation

Christopher Theisen
North Carolina State University
Department of Computer Science
890 Oval Drive, #8206 Raleigh, North Carolina, United States
+1 919 515 2858
crtheise@ncsu.edu

ABSTRACT

In our increasingly interconnected world, software security is an increasingly important issue for development teams. However, there is too much security work to do for these teams as security needs have out-scaled security resources. To help prioritize security efforts, professionals use the *attack surface* of a system, or the sum of all paths for untrusted data into and out of a system, to identify security relevant code. However, identifying code that lies on the attack surface is a difficult and resource-intensive process. Our research proposes the use of crash dump stack traces as an empirical metric for approximating the attack surface. We hypothesize that code that appears on crash dump stack traces represent activity that has put the system under stress, and is therefore indicative of potential security vulnerabilities. The goal of this research is *to aid software engineers in prioritizing security efforts by approximating the attack surface of a system via crash dump stack trace analysis.* In a trial on Mozilla Firefox, the risk-based attack surface approximation selected 15.8% of files and contained 73.6% of known vulnerabilities. Randomly sampling 10% of crash dump stack traces for inclusion in our analysis resulted in only 2.7% fewer known vulnerabilities included on our attack surface. Through our approach, we look to optimize effort for the security community in finding, fixing and preventing security vulnerabilities.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics – *complexity metrics, process metrics, product metrics*

General Terms

Management, Measurement, Design, Economics, Security.

Keywords

Stack traces, crash dumps, attack surface.

1. PROBLEM AND MOTIVATION

The attack surface of a system can be used to determine which parts of a system could have exploitable security vulnerabilities. Items not on the attack surface of a system are unreachable by outside input, and, therefore, less likely to be exploited. If outside input cannot be passed to code containing a security vulnerability, engineering hours spent working on finding and fixing that vulnerability could be spent elsewhere. If generating the attack surface of a system was a more straightforward process, security professionals could focus their efforts on code containing vulnerabilities that are reachable, and therefore exploitable, by malicious users. Reducing the amount of code to be inspected may help improve the economics of security assessments and allow for more efficient proactive reviews of potentially vulnerable code.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author. Copyright is held by the owner/author(s).

The Open Web Application Security Project (OWASP) defines the attack surface of a system as the paths in and out of a system, the data that travels those paths, and the code that protects the paths and the data [1]. In the research community, Howard et al. introduced the concept of an attack surface, describing entry points to a system that might be vulnerable along three dimensions: targets and enablers, channels and protocols, and access rights [5]. Later, Manadhata and Wing [12] formalized the notion of attack surface, including methods, channels, untrusted data, and a direct and indirect entry and exit point framework that identifies methods through which untrusted data passes.

The software engineering community lacks a practical means of identifying the parts of the system that are on the attack surface. The goal of this research is *to aid software engineers in prioritizing security efforts by approximating the attack surface of a system via crash dump stack trace analysis*. We propose *risk-based attack surface approximation* (RASA), an automated approach to identifying parts of the system that are contained on the attack surface through stack trace analysis. We parse stack traces, adding all code found in these traces onto RASA. Code that appears in stack traces caused by user activity is on the attack surface because it appears in a code path reached by users.

Crash dump stack traces from user-initiated crashes have three desirable attributes for measuring attack surfaces: (a) they represent user activity that puts the system under stress; (b) they include both direct and indirect entry points; and (c) they provide automatically generated control and data flow graphs. We seek to assess the degree to which these attributes of stack traces support the identification of attack surfaces. We call our approach "**Risk-Based** Attack Surface Approximation" because it is an efficient means of identifying the part of the attack surface that is most susceptible to containing vulnerabilities.

We assess our approach by analyzing the percentage of actual reported vulnerabilities in the code and whether they occur in our approximated attack surface. The higher the percentage of vulnerabilities covered on our attack surface approximation and the smaller the subset of total code artifacts that appear on our approximation, the better our approach is performing. In addition, we also explore randomly sampling crash dump stack traces for building our approximation. If a randomly sampled subset of crash dump stack traces results in similar performance to using every available crash, then sampling may be an effective way to reduce the amount of data required by RASA.

2. BACKGROUND AND RELATED WORK

In this section, we provide a brief overview of related work.

2.1 Attack Surface

Howard et al. [5] provided a definition of attack surface using three dimensions: targets and enablers, channels and protocols, and access rights. Not all areas of a system may be directly or indirectly exposed to the outside. Some parts of a complex system, e.g. Windows OS, may be for internal use only and cannot be reached or exploited by an attacker. For example, installation routines are left in the system, but they are never accessed again and are unlikely to have security implications.

Manadhata et al. [11] describe how an attack surface might be approximated by looking at Application Program Interface (API) entry points. However, the Manadhata approach does not cover all exposed code, as the authors mention. Specifically, internal flow of data through a system was not identified. While the external points of a system are a useful place to start, they do not encompass the entirety of exposed code in the system. Internal points within the system could also contain security vulnerabilities that the reviewer should be aware of. Previous efforts to determine the attack surface of a system have used API scanning techniques [12], but these techniques have limitations in terms of how much code they can cover. Further, their approach to measuring attack surfaces required expert judgment of security professionals to determine if code is security relevant.

In our previous RASA study [15], researchers found a correlation between binaries that appear on stack traces from crash dumps and code that contained at least one security vulnerability fix. The correlation could be useful to security professionals when targeting security reviews of codebases. By targeting security efforts to binaries in the ASA instead of the entire codebase, security professionals could save engineering hours. The researchers created the ASA by parsing stack traces from Windows 8 OS, and including any binaries involved in a stack trace in their approximation. They evaluated the effectiveness of their approach by comparing the approximation against the location of historical vulnerabilities in Windows 8 OS. In that study, 48.4% of shipped binaries seen in at least one crash dump stack trace in Windows 8 OS contained 94.8% of the vulnerabilities seen over the same time period [15].

2.2 Using Crash Reports

The use of crash reporting systems, including stack traces from the crashes, is becoming a standard industry practice[1] [16][2]. Bug reports contain information to help engineers replicate and locate software defects. Liblit and Aiken [10] introduced a technique automatically reconstructing complete execution paths using stack traces and execution profiles. Later, Manevich et al. [13] added data flow analysis information on Liblit and Aiken's approach. Other studies use stack traces to localize the exact fault location [7][17][16]. Lately, an increasing number of empirical studies use bug reports and crash reports to cluster bug reports according to their similarity and diversity, e.g. Podgurski et al. [14] were among the first to take this approach. Other studies followed [2][9]. Not all crash reports are precise enough to allow for this clustering. Guo et al. [3] used crash report information to predict which bugs will get fixed. Zimmermann et al. [18] assessed the quality of bug reports to suggest better and more accurate information helping developers to fix the bug.

With respect to vulnerabilities, Huang et al. [6] used crash reports to generate new exploits while Holler et al. [4] used historic crashes reports to mutate corresponding input data to find incomplete fixes. Kim et al. [8] analyzed security bug reports to predict "top crashes"—those few crashes that account for the majority of crash reports—before new software releases. As mentioned previously, we expanded on previous studies by exploring the correlation between code appearing in a stack trace and having historical vulnerabilities [15].

3. APPROACH AND UNIQUENESS

In this section, we describe the implementation of RASA and our approach to randomly sampling stack traces for a case study.

3.1 RASA Implementation

To implement RASA for a target system, we first select a collection of stack traces from crash dumps from the software system we are analyzing. These stack traces are chosen from a set period of time. For each individual stack trace pulled from a crash dump, we isolate the binary, file, or function on each line of each stack trace, and record what code artifact was seen and how many times it has been seen in a stack trace. Each of the code artifacts from stack traces should then be mapped to a code artifact in the system. For example, if the file foo.cpp appears in a stack trace, the matching foo.cpp in system should be identified. A software system may have multiple foo.cpp files, so a method for identifying which foo.cpp was in the crash is required. A list of code artifacts in a software system could come from toolsets provided by the company maintaining the system or pulled directly from source control, in the case of open source projects.

We have created a toolset to parse each individual stack trace in our target dataset in sequence, and extract the individual code artifacts that appear on each line. The tool then outputs the frequency in which each unique code artifact appears in a stack trace from the parsed set. For this particular study, we do not consider the number of times a code artifact appears; only that it appears at least once

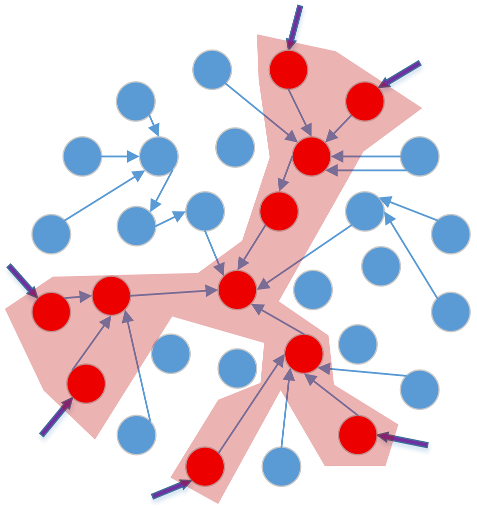

Figure 1: A visual representation of what an attack surface is for a system; the shaded area is the attack surface, where input flows through the system.

[1] http://www.crashlytics.com/blog/its-finally-here-announcing-crashlytics-for android/

To tie stack trace appearances to the codebase, we generate a list of all source code files from the system under inspection and combine that list with the list of appearances in stack traces. A visualization of the code appearing on RASA can be seen in Figure 1. The individual tools developed for some of these steps can be found on our GitHub[2] page. In addition to the list of files on the ASA, we count the number of artifacts that have security vulnerabilities.

After we have the list of code that appears on at least one stack trace and the code that had at least one vulnerability fix, we calculate two RASA evaluation metrics:

1. The percentage of code in the target software system that appears in at least one stack trace (or the Risk-based Attack Surface Approximation), and
2. The percentage of files with security vulnerabilities that appear in at least one stack trace, or vulnerability coverage.

For 1) above, we calculate the percentage of files found on stack traces via the following formula. We define this metric as File Coverage (FC):

$$(1)\ FC = \frac{code\ artifacts\ on\ at\ least\ one\ stack\ trace}{total\ number\ of\ code\ artifacts\ in\ the\ system}$$

For 2) above, we calculate our vulnerability coverage via the following formula. We define this metric as Vulnerability Coverage (VC):

$$(2)\ VC = \frac{code\ artifacts\ with\ vulns.on\ stack\ trace}{total\ number\ of\ code\ artifacts\ with\ vulns.}$$

3.2 Data Requirements

The initial study on RASA was performed on Microsoft Windows 8 [15] and was done with millions of crashes. Not all organizations have as much crash information as these large organizations, so the feasibility of RASA on smaller datasets should be explored. To explore this idea, we take percentages of available stack traces from the target software system, from 90% of the total stack traces available to 10% of the available stack traces. The 100% case is covered by our initial experiment in Section 3.1. We can then explore the difference in code coverage in the resultant RASA, and the difference in covered security vulnerabilities in the resultant RASA. Our hypothesis is as we increase the number of stack traces in our RASA, our code coverage and vulnerability metrics will converge towards our metrics for 100% stack trace use.

For each of these slices, we perform the random sampling analysis as outlined in section 3.2. From those results, we can see how sampling affects the result of ASA. Our hypothesis is as we increase the number of stack traces in our ASA, the effect of random sampling on the end result will decrease.

3.3 Uniqueness

While other researchers have made use of crash dump stack traces as a potential metric for exploring software defects, to our knowledge there is little previous work on using crash dump stack traces as a metric for security vulnerabilities. In addition, previous work has focused on individual stack traces for analysis, while our approach focuses on the aggregation of a large set of stack traces to develop our results.

To our knowledge, RASA is the first approach based on empirical data collection to approximate the attack surface of a system. Other attack surface tools and approaches, such as those described in section 2.1 and tools such as Microsoft's Attack Surface Analyzer, focus on program analysis techniques rather than existing datasets. An approach based on analysis of stack traces may be more practical for the generalization of attack surface analysis, as many organizations already collect crash dump stack traces from their customers. Repurposing this existing dataset could lower the barriers to entry for implementation of RASA in the field.

4. RESULTS AND CONTRIBUTIONS

In the initial RASA study [15], we found a correlation between code artifacts that appear on stack traces generated by the system and where historical vulnerabilities discovered by security professionals have been fixed in code. The attack surface correlation could be useful to security professionals when targeting security reviews. By targeting security efforts at these exposed areas instead of the entire codebase, security professionals can maximize the impact of the engineering hours they have available to them. In the previous study, it was found that 48.4% of binaries in Windows contained 94.8% of historically seen vulnerabilities [15]. Limiting security engineering efforts to half of the codebase while still finding the majority of potential bugs is a tradeoff teams can make.

After applying RASA, 15.8% of files shipped with Firefox are included on the attack surface, and this subset contains 73.5% of the historical vulnerabilities seen over the same period of time. These results suggest that code that appears in stack traces derived from crashes are more likely to have vulnerabilities as well. If the program is crashing, that indicates a data flow path that has put the system under stress and may contain errors that result in security vulnerabilities. From the result, we conclude that the automated attack surface approximation approach may be useful in limiting the scope of code that developers need to review while missing a minimal number of potentially flawed areas.

We have also improved the granularity of attack surface approximation compared to the previous study [1], in addition to the quantitative improvements in coverage and specificity. By performing attack surface approximation at the file level, we provide more actionable results for practitioners. While a single binary file could contain thousands of individual files for developers to review, files are typically a more manageable level of granularity for a developer, depending on the development practices of the organization using attack surface approximation.

The average number of files covered by RASA and the average number of security vulnerabilities covered by RASA at various random sampling points is also found in Figure 2 and Figure 3. As the size of the random sampling increases, we see that the number of files covered by RASA also increases, while the standard deviation of the individual runs shows no discernable trend. For coverage of security vulnerabilities, we also see a slight increase in coverage as the random sampling size increases. In the case of security vulnerability coverage, we see that the standard deviation decreases as the sample size increases. In our full results, only 6 files associated with a security vulnerability fix appeared in only one stack trace from the Firefox dataset. In the dataset, only 15 files associated with security vulnerability fixes appeared in less than 10 crashes. In the full dataset, the difference in total vulnerability

[2] https://github.com/theisencr/stack-trace-parser

coverage from a 10% sample to the complete set of crashes is 11 files.

From these results, we conclude that randomly sampling stack traces for the Firefox dataset is an effective approach for reducing the amount of data required to implement RASA. A 10% random sample has a minimal effect on the final approximation, meaning organizations can store a fraction of their customer crashes and still make use of our approach to improve their security efforts. Our intuition told us that random sampling would cause an equivalent drop in coverage of security vulnerabilities: why is this not the case? Our observation that few files appear less than 10 times in the full Firefox dataset could possibly explain why random sampling had a minimal effect on vulnerable file coverage. In order for a vulnerable file to longer be covered by RASA, it cannot appear in *any* stack trace from a crash in the target system. For example, a 30% sampling of crashes is likely to include at least one occurrence of foo.cpp if it occurs 8 times in the complete dataset.

While this result indicates that RASA can make effective use of sampling for large projects like Firefox, it also has implications for smaller projects that may not have crash dump stack traces on the same scale. For a smaller project that collects 10% of the crashes that Firefox does, RASA may still be a valuable technique for prioritizing security efforts. Overall, these results are promising for the implementation of RASA across a wide variety of software projects.

In this paper and in previous work, RASA was generated based on an on/off approach. If a code artifact appeared in at least one crash dump stack trace, then RASA considers that code entity as part of the attack surface of the system. However, further prioritization within RASA may be possible. The frequency in which code appears in stack traces from crash dumps may be an additional metric to explore for further prioritization of security reviews beyond RASA. The more a code artifact is involved in crashes, the more likely it might be that that code artifact has a related security vulnerability.

In addition to our initial feasibility study and random sampling study, we have also explored the effect of frequency of appearance of code on crash dump stack traces and their likelihood of containing a security vulnerability. We have found a casual relationship between frequency of appearance and our FC and VC metrics. In future studies, we plan to explore this frequency of appearance metric in more detail.

5. FUTURE WORK

RASA currently looks at a specific time period of crashes and vulnerabilities to build its attack surface approximation. In the future, we plan to explore how turning this time period into a sliding window instead of a static period of time. For example, do crashes from a previous time slice predict vulnerabilities that appear in future slices?

RASA is an approximation of the attack surface, and as such not every vulnerability is covered by the approach. Analyzing the types of vulnerabilities not covered by RASA is an important step for determining the "blind spots" of the approach so teams can use other methods to find and fix those vulnerabilities.

RASA currently looks at the code entities themselves as possible locations for security vulnerabilities. The code entities themselves may not be the interesting metric from a security perspective. The *relationships* between code entities may do a better job of pointing out potential vulnerabilities. Many common vulnerability types are the result of bad data handling, including SQL injection attacks and buffer overflow attacks. Examining the relationships between files (or other code entities at various levels of granularity) and determine which relationships appear in crashes most frequently. These bad handoffs may point us towards where vulnerable code lives.

In addition to the statistical results from mining crash dump stack traces, exploring shapes within graph representations of the crash dump stack traces is another area we plan to explore to narrow the scope of code that could contain security vulnerabilities. In particular, do certain shapes of incoming and outgoing nodes result in more frequent sightings of vulnerabilities? We hypothesize that certain shapes, such as many code entities calling into one entity but that entity only calling out to few entities, may exhibit more vulnerabilities than other shapes.

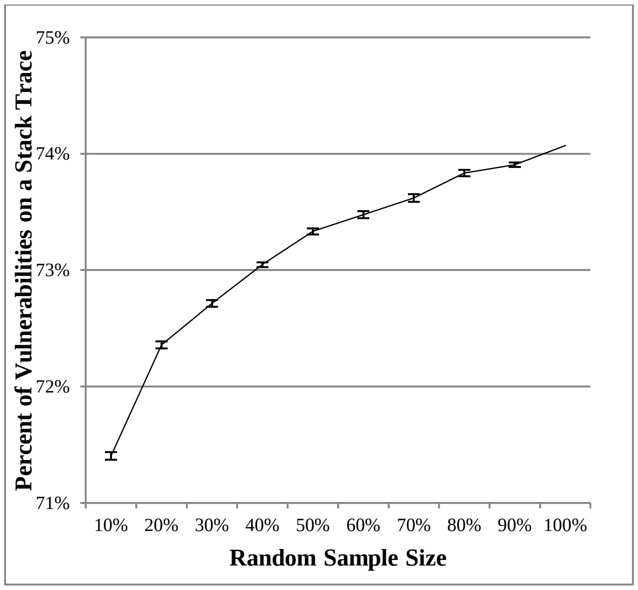

Figure 2: Graph of the percentage of files included on the RASA at random samples, with error bars indicating the deviation between samples.

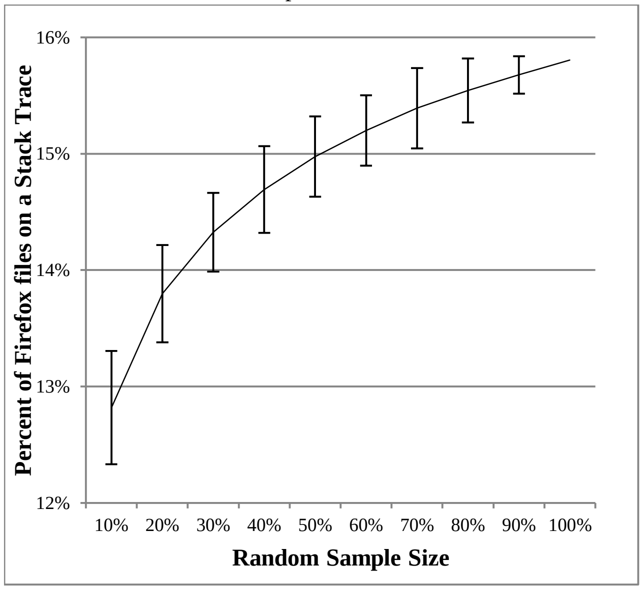

Figure 3: Graph of the percentage of vulnerabilities covered by RASA at random samples, with error bars indicating the deviation between samples.

Where code appears on graph representations of software systems may also be important for prioritization of security efforts. For example, if security bugs are more likely to appear on the "edge" of a software system, or closer to API entry points, then prioritization of those code artifacts may be useful for finding security vulnerabilities faster.

Visualizing the shape and edge effects may also result in meaningful impact for security professionals as well. Building dynamic visualizations similar to the one presented in Figure 1 may help security professionals better understand the systems they are analyzing. Developers already make use of call graphs to understand the relationship between code artifacts in software systems, and extending the graph metaphor to crash dump stack traces could make Figure 1 an effective visualization of the attack surface for developers.

RASA may create several positive impacts on the software engineering community. An automated approach to attack surface generation could allow security teams to make more efficient use of their time, reducing the amount of hours used on these tasks, allowing for more efficient discovery of vulnerabilities, or a combination of both. Because the development of the attack surface of their product would be automated, they would not need to tie up resources developing one themselves. For organizations without a security team or just starting security efforts, RASA gives them the first steps toward targeting what limited security resources they have. In addition, the analysis of how much data is required can help engineering firms make informed decisions on how many resources need to be dedicated to implementation of RASA.

6. REFERENCES

[1] Bird, J. and Manico, J. OWASP Attack Surface Analysis Cheat Sheet. *Open Web Application Security Project*, 2015. https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet.

[2] Dang, Y., Wu, R., Zhang, H., Zhang, D., and Nobel, P. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. *International Conference on Software Engineering (ICSE)*, (2012), 1084–1093.

[3] Guo, P.J., Zimmermann, T., Nagappan, N., and Murphy, B. Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, (2010), 495.

[4] Holler, C., Herzig, K., and Zeller, A. Fuzzing with code fragments. *Proceedings of the 21st USENIX conference on Security symposium (Security'12)*, (2012), 38–38.

[5] Howard, M., Pincus, J., and Wing, J.M. Measuring Relative Attack Surfaces. *Computer Security in the 21st Century*, CMU-TR-03-169 (2005), 109–137.

[6] Huang, S.K., Huang, M.H., Huang, P.Y., Lu, H.L., and Lai, C.W. Software crash analysis for automatic exploit generation on binary programs. *IEEE Transactions on Reliability 63*, 1 (2014), 270–289.

[7] Jin, W. and Orso, A. F3: Fault Localization for Field Failures. *International Symposium on Software Testing and Analysis*, (2013), 213–223.

[8] Kim, D., Wang, X., Kim, S., Zeller, A., Cheung, S.C., and Park, S. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering 37*, 3 (2011), 430–447.

[9] Kim, S., Zimmermann, T., and Nagappan, N. Crash Graphs: An Aggregated View of Multiple Crashes to Improve Crash Triage. *Dependable Systems & Networks (DSN)*, (2011), 486 – 493.

[10] Liblit, B., Aiken, A., Liblit, B., and Aiken, A. Building a Better Backtrace: Techniques for Postmortem Program Analysis. *Technical Report. University of California at Berkeley*, October (2002).

[11] Manadhata, P., Wing, J., Flynn, M., and McQueen, M. Measuring the attack surfaces of two FTP daemons. *2nd ACM workshop on Quality of Protection*, (2006), 3–10.

[12] Manadhata, P. and Wing, J. An attack surface metric. *IEEE Transactions on Software Engineering 37*, 3 (2011), 371–386.

[13] Manevich, R., Adams, S., Das, M., and Yang, Z. PSE: Explaining Program Failures via Postmortem Static Analysis. *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering (SIGSOFT '04/FSE-12)*, (2004), 63–72.

[14] Podgurski, A., Leon, D., Francis, P., et al. Automated support for classifying software failure reports. *25th International Conference on Software Engineering, 2003.*, (2003), 465–475.

[15] Theisen, C., Herzig, K., Morrison, P., Murphy, B., and Williams, L. Approximating Attack Surfaces with Stack Traces. *IEEE/ACM 37th IEEE International Conference on Software Engineering*, (2015).

[16] Wang, S., Khomh, F., and Zou, Y. Improving bug localization using correlations in crash reports. *IEEE International Working Conference on Mining Software Repositories*, (2013), 247–256.

[17] Wu, R., Zhang, H., Cheung, S.-C., and Kim, S. CrashLocator: locating crashing faults based on crash stacks. *International Symposium on Software Testing and Analysis (ISSTA)*, (2014), 204–214.

[18] Zimmermann, T., Premraj, R., Bettenburg, N., Just, S., Schröter, A., and Weiss, C. What makes a good bug report? *IEEE Transactions on Software Engineering 36*, (2010), 618–643.