Risk-Based Attack Surface Approximation: How Much Data is Enough?

Christopher Theisen
Computer Science
NC State University
Raleigh, NC, United States
crtheise@ncsu.edu

Brendan Murphy
Microsoft Research
Cambridge, UK
bmurphy@microsoft.com

Kim Herzig
Microsoft Research
Redmond, WA
kimh@microsoft.com

Laurie Williams
Computer Science
NC State University
Raleigh, NC, United States
lawilli3@ncsu.edu

Abstract— **Proactive security reviews and test efforts are a necessary component of the software development lifecycle. Resource limitations often preclude reviewing the entire code base. Making informed decisions on what code to review can improve a team's ability to find and remove vulnerabilities. Risk-based attack surface approximation (RASA) is a technique that uses crash dump stack traces to predict what code may contain exploitable vulnerabilities. The goal of this research is *to help software development teams prioritize security efforts by the efficient development of a risk-based attack surface approximation.* We explore the use of RASA using Mozilla Firefox and Microsoft Windows stack traces from crash dumps. We create RASA at the file level for Firefox, in which the 15.8% of the files that were part of the approximation contained 73.6% of the vulnerabilities seen for the product. We also explore the effect of random sampling of crashes on the approximation, as it may be impractical for organizations to store and process every crash received. We find that 10-fold random sampling of crashes at a rate of 10% resulted in 3% less vulnerabilities identified than using the entire set of stack traces for Mozilla Firefox. Sampling crashes in Windows 8.1 at a rate of 40% resulted in insignificant differences in vulnerability and file coverage as compared to a rate of 100%.**

Keywords- stack traces, attack surface, prediction models

I. Introduction

The attack surface of a system can be used to determine which parts of a system's codebase could have exploitable security vulnerabilities. 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] 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 should be spent elsewhere. Vulnerability detection and removal techniques, such as security reviews and penetration testing, can therefore be prioritized to code attack surface, rather than being applied indiscriminately. Reducing the amount of code to be inspected may help improve the economics of security assessments and allow for more proactive reviews of potentially vulnerable code.

Risk-based attack surface approximation (RASA) [1] is an approach to identifying code in a software system that is contained on the attack surface through crash dump stack trace analysis. Code that appears in stack traces caused by outside activity is at risk of having security vulnerabilities as well. For RASA, all code found in the stack traces from crash dumps is classified as being on the attack surface of a system. Mining crash dumps to determine what code is crashing may result in a useful metric for determining where security vulnerabilities are in code, because stack traces in crash dumps indicate what code was involved in a failure. Additionally, attackers may use stack traces from crash dumps to determine where flawed input handlers may be in a software system. From the attacker's perspective, a repeatable crash could be exploited as a potential denial of service attack.

The goal of this research is to help software development teams prioritize security efforts by approximating the attack surface of a software system via risk-based attack surface approximation. For some organizations, storing and analyzing every single crash seen internally and by customers may be infeasible, as some products can quickly generate hundreds of millions of crashes if popular. Therefore, we explore the effect of randomly sampling stack traces from crash dumps on the final RASA to determine of it is a viable strategy to make RASA more economical for practitioners.

We explore the following research questions:

- **RQ1:** How effective is risk-based attack surface approximation in predicting the location of security vulnerabilities?
- **RQ2:** How does random sampling of crash dump stack traces affect the variability and effectiveness of the resulting risk-based attack surface approximation in predicting the location of security vulnerabilities?

In this paper, we performed a RASA for Firefox and Microsoft Windows, based on stack traces collected from both products. To assess RASA, we compare the set of known security vulnerabilities from each product's respective bug database against the files identified as part of the approximation. After generating the initial RASA, we then rerun the experiment using random samples of crashes from Firefox and Windows to determine how sampling may change the final result, compared to using the entire set of crashes.

[1] https://www.owasp.org/index.php?title=Attack_Surface_Analysis_Cheat_Sheet&oldid=156006

We include the following as contributions in this paper:

- An evaluation of the effectiveness of risk-based attack surface approximation for an open source application that corroborates with an earlier attack surface approximation study on a proprietary product [1].
- An analysis of the effect of random sampling of crash dump stack traces on the final result of risk-based attack surface approximation
- An exploration of the amount of stack traces from crash dumps required for actionable results for risk-based attack surface approximation.

The rest of the paper is organized as follows: Section 2 discusses background and related work, Section 3 presents our research methodology, Sections 4 and 5 present our case studies for Firefox and Windows, respectively, Sections 6 and 7 discuss the results and why we ended up with them, Section 8 presents limitations and threats to validity, and Section 9 concludes and discusses future work.

II. Background and Related Work

In this section, we provide a brief overview of related work and the previous study done in the area of attack surfaces and defect prediction.

A. *Attack Surface*

As mentioned previously, The Open Web Application Security Project (OWASP) defines the *attack surface* of a system as the paths into and out of a system, the data that travels those paths, and the code that protects the paths and the data. The OWASP attack surface definition also includes "the sum of all paths for data/commands into and out of the application." Howard et al. [17] provided a definition of attack surface using three dimensions: 1) targets and enablers; 2) channels and protocols; and 3) access rights. Not all areas of a system may be directly or indirectly exposed to the outside. Some parts of a complex software system, e.g. Windows OS, may not be reachable or exploitable by an attacker. In Figure 1, we present a graphical representation of what the attack surface of a system is. The nodes with the thick dark arrows pointing at them are the entry points into a system, showing where an outsider can pass input into a system. The remaining shaded nodes and arrows represent the path outside input takes through the system, with data eventually terminating in the center of the system.

Manadhata et al. [33] 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 [42], 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.

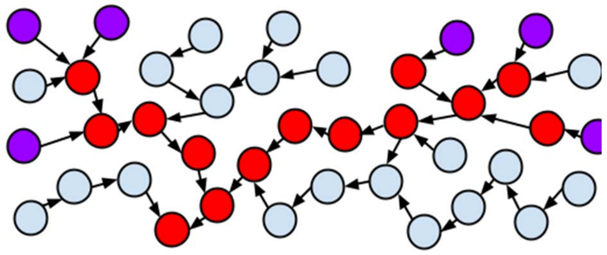
Figure 1. A visual representation of what an attack surface is for a system. The darker nodes represent the attack surface, where input flows through the system. Nodes represent individual binaries, files or functions in a target system.

In a previous RASA study [1], 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 RASA instead of the entire codebase, security professionals could save engineering hours. The researchers created the RASA 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 [1].

However, the industrial study has a few limitations. First, the approximation was only performed at the binary level of the application. A single binary could contain thousands of files, making the metric difficult to act on. Second, the industrial study only looked at an operating system. To address these concerns, we constructed an RASA using stack traces at the file level instead of the binary level on an open source application instead of an operating system and report the results in this paper.

B. *Exploiting Crash Dumps*

The use of crash dumps, including stack traces from the crashes, is becoming used more frequently for identifying defects and vulnerabilities[2] [23][25]. Liblit and Aiken [18] introduced a technique automatically reconstructing complete execution paths using stack traces and execution profiles. Later, Manevich et al. [19] added data flow analysis information on Liblit and Aiken's approach to explain program failures. Other studies use stack traces to localize the exact fault location [21][22][23]. An increasing number of empirical studies use bug reports and crash dumps to cluster bug reports according to their similarity and diversity, e.g.

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

Podgurski et al. [24] were among the first to take this approach. Other studies followed [25][26]. Not all crash dumps are precise enough to allow for clustering. Guo et al. [27] used crash dump information to predict which bugs will get fixed. Bettenburg et al. [28] assessed the quality of bug reports to suggest better and more accurate information for helping developers to fix bugs.

With respect to vulnerabilities, Huang et al. [29] used crash dumps to generate new exploits while Holler et al. [30] used historic crash reports to mutate corresponding input data to find incomplete fixes. Kim et al. [31] analyzed security bug reports to predict "top crashes"—those few crashes that account for the majority of crash dumps—before new software releases.

III. Research Methodology

In this section, we discuss our research methodology to answer our three research questions.

A. Risk-Based Attack Surface Approximation (RASA)

To create the 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. The use of frequency as a potential metric for future RASA studies is discussed in section 9. 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 flowchart detailing the process is shown in Figure 2. In addition to the list of files on the RASA, we count the number of artifacts that have security vulnerabilities. An example of a list of files with counts of appearances on crash dump stack traces is found in Figure 3. 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):

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

B. Random Sampling of Crashes

For some organizations, analyzing every crash available may result in the storage and analysis of hundreds of millions of crashes. Storing and analyzing that much data may be unfeasible for the organizations, so a random sampling approach may be required to limit the amount of crashes stored and analyzed. However, random sampling of crashes may result in variations in the final result of RASA, weakening the approximation and decreasing its usefulness to practitioners. Determining the effect of random sampling on the end result of the RASA is therefore important to understand the use cases of the approach.

To test the effect of random sampling, we take set percentages of stack traces from the overall dataset in our study. The stack traces chosen are by random selection. We then repeat random selection 10 times, resulting in 10 different sets of stack traces representing the same total percentage of stack traces from the original dataset. For example, a practitioner could choose to permanently store 20% of all crashes reported by users. To simulate this case, we randomly sample 20% of the crashes from a target dataset 10 times to determine the resulting variance. We then compare several metrics on the 10 separate samplings:

1) The percentage of files included on stack traces in each sample versus the entire set of files in the system.

2) The total coverage of the sampled RASA of the vulnerabilities included on the attack surface.

3) The change in individual code entities covered on different samplings, or the standard deviation between the 10 samplings.

4) The change in individual security vulnerabilities covered on different samplings, or the standard deviation between the 10 samplings.

From these results, we can then draw conclusions on the effect of sampling on this approach to RASA.

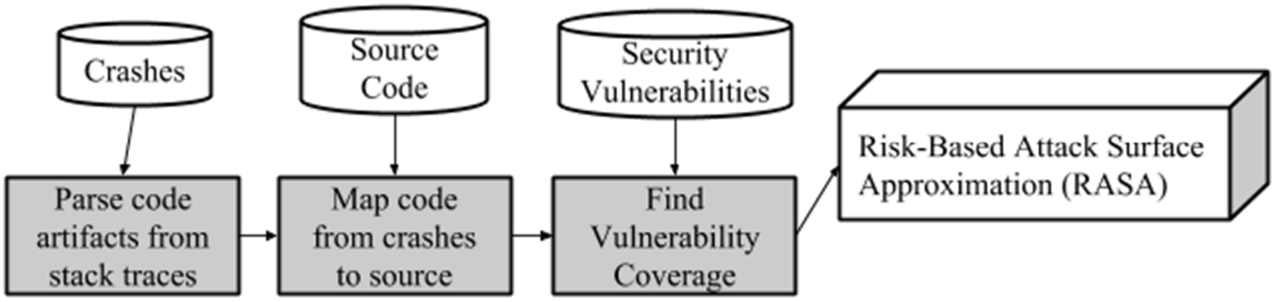
Crashes
Source Code
Security Vulnerabilities
Parse code artifacts from stack traces
Map code from crashes to source
Find Vulnerability Coverage
Risk-Based Attack Surface Approximation (RASA)

Figure 2. A visualization of the workflow for developing a Risk-Based Attack Surface Approximation for a target software system.

C. Data Requirements

The initial study on RASA was performed on Microsoft Windows 8 [1] 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. We expect that an increase in the percentage of stack traces included in our study will result in our code coverage and vulnerability metrics converging towards our result with all stack traces.

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 RASA. As we increase the percentage of stack traces in each step of our study, we expect the error bounds of our metrics will decrease.

IV. Firefox Case Study

In this section, we discuss our first case study on Mozilla Firefox. The Firefox team makes crash data from customers (with identifying information removed) available publicly. They also make security vulnerability fixes public after the affected release has passed out of recommended public use. Firefox is written in several languages and is a large codebase, with approximately 50,000 files in the production codebase in the study period of May 2010 through March 2012.

A. Data Collection

Mozilla only makes security vulnerability details available once the vulnerability has passed out of public use in all versions of Firefox. Because of that policy, vulnerability information is only available from before 2012. Therefore, we could not make use of Mozilla's primary stack trace data website, Mozilla Crash Reports[3], as it only keeps full stack traces from crashes for approximately 6-7 months. Instead, we made use of the historical dumps at https://crash-analysis.mozilla.com/crash_analysis/. We performed our analysis on crashes occurring from May 2010 to March 2012 due to the available security data. An example of the format of code crash strings is found in Figure 4. For the crash-analysis dataset, this format was consistent throughout the entire dataset allowing us to build a string parser to grab the file path and filename from the middle of the string, as delimited by the colons. Crash dumps from the historical dataset do not contain the entirety of the stack trace. Only the topmost filename is included in each trace. While the Firefox stack traces provide less detail than the Windows stack traces, observing only the last file seen on the stack trace may be another approximation technique that eliminates more files from the attack surface. We explore the impact on the completeness of RASA, with the metric for completeness being the number of vulnerabilities seen on the approximation.

Name	Crashes
js/src/jsgc.cpp	79705
layout/generic/nsFrame.cpp	73405
js/src/jsobj.cpp	71040
js/src/xpconnect/src/xpcnative.cpp	51309
xpcom/io/nsLocalFileWin.cpp	41783
layout/generic/nsObjectFrame.cpp	39853
modules/plugin/base/src/ns.cpp	37226
js/src/jstracer.cpp	36076
js/src/jsapi.cpp	35671
js/src/jsinterp.cpp	28912

Figure 3. A subset of the final dataset used for analysis (some names shortened).

```
hg:hg.mozilla.org/releases/mozilla-
1.9.2:view/src/nsViewManager.cpp:
  448d0d2d310c
hg:hg.mozilla.org/releases/mozilla-
1.9.2:xpcom/threads/nsThread.cpp:
  28ef231a65a3
hg:hg.mozilla.org/releases/mozilla-
1.9.1:layout/generic/nsFrame.cpp:
  c307a617e5a5
hg:hg.mozilla.org/releases/Mozilla-
1.9.2:nsprpub/pr/src/md/windows/w95sock
.c:28ef231a65a3
hg:hg.mozilla.org/releases/mozilla-
1.9.1:objfirefox/dist/include/string/ns
Algorithm.h:c307a617e5a5
```

Figure 4. Examples of files seen in the topmost_filename field in the Firefox crash dumps.

[3] https://crash-stats.mozilla.com/home/products/Firefox

To collect our security data, we parsed security reports from Mozilla's security advisory blog from the same May 2010 to March 2012 period. Each security report presented by Mozilla has an associated diff or bug report, indicating what files were changed as part of the security fix. As mentioned previously, part of the reasoning behind selecting this time period was the availability of these security reports. Mozilla does not always release security bug details for newer vulnerabilities for a variety of reasons, such preventing the exploit from becoming more widespread. Mozilla makes vulnerability details available later after they are confident the issue has been resolved for their users.

B. *Random Sampling*

To create our random samples to answer our data sampling and randomization questions, we make use of the random library in Python 3.X to create our random samples. We consider each stack trace for inclusion in our sample by generating a random number from 0 to 1 and comparing it against our desired percentage for inclusion. For example, if we want to include approximately 30% of stack traces in our RASA, then we take all stack traces that have random numbers generated for them that are less than 0.3. It is important to randomly sample each individual stack trace rather than randomly choosing sets of stack traces (by hour, minute, or day) as specific time periods may be weighted towards specific types of crashes. By considering every stack trace for random inclusion in our dataset as opposed to blocks of time, we simulate how a version of RASA running for a practitioner may choose to keep crashes for later analysis.

V. Windows Case Study

In this section, we discuss our second case study on Microsoft Windows 8.1.

A. *Data Collection*

Each line of a stack trace is organized as follows. The binary is shown at the beginning of the string, followed by a "!" delimiter and the function name. In the square brackets, the full path of the file associated with this binary/function relationship is shown. Not all stack traces will include the name of the source file. Some stack traces may even display anonymous placeholders for functions and binaries, depending on the permissions and ability to identify these details during runtime. For example, Windows stack traces contain no details about artifacts outside Windows, e.g. a third-party application causing the crash.

Each stack trace is parsed and separated into individual artifacts, including binary name, function name, and file name. We then map each of these artifacts to code as they are named in Microsoft's internal software engineering tools. File information is not always available. In these cases, we make use of software engineering data indicating relationships between binaries, files, and functions to find the missing data if possible. If these symbol tables contain the function name referenced by the stack trace, we pull the corresponding source file onto the attack surface. In case the function name is not unique, e.g. overloading the function in multiple files, we over approximate the attack surface and pull all possible source files onto the attack surface. If no function name can be found, e.g. function not shipped with Windows, we leave the file marked as unknown. Thus, this approach generates an attack surface that is an approximation of reality.

When code is seen in a stack trace, we place information about that code into a database table containing all code on the attack surface approximation. When this code is added to the database, we enter as much information as possible about the line in the stack trace. In some cases, this is just the binary, as the file and function cannot be mapped. Other cases may have the exact file and/or function. We also collect the list of artifacts appearing directly before and after each artifact in each stack trace. This data can be used in a variety of helpful ways, particularly in visualizing these relationships in graph format as seen in Figure 1.

Sometimes actual entities within the system are unable to be mapped from the stack traces. For example, errors occurring during the process of creating the stack trace could result in unknown code entities on the trace. When a mapping is unable to be made, we label that entity as "unknown," and do not place that entity on the attack surface. The output used by the development and security teams is a classification of whether an entity is on or off the attack surface. This classification can be used for prioritizing defect fixing, validation, and verification efforts.

After parsing out individual traces, we were left with approximately 9 million crashes from Windows 8.1 to run our sampling study.

We map security bug information to specific code artifacts found during our parsing of crash dump stack traces. We collect security bug information at the file level, and map the bug information to files from stack traces. Individual bugs are also defined as pre-release or post-release, depending on when the bug was found during the development process. Pre-release is defined as bugs found in code before its official release to customers, where official does not include customer alpha or beta releases. Post-release is defined as bugs found in code after an official release. We use post-release bugs as our set of vulnerabilities for this study.

B. *Random Sampling*

To create our random samples, the collection of stack traces was placed in a MSSQL database, with a table dedicated to individual lines for individual classes, and a second table with mappings for individual crashes to the version of Windows that crash was found on. Using the C# programming language, we randomly sampled individual crashes at each sampling size (10% of the available crashes, 20% of the available crashes, etc. If a crash was chosen for a particular sample, we included files that were seen in that crash in RASA for that run. For Firefox, this is the last file seen for a particular crash, while in Windows this is all files seen in the crash. We ran 10 samples at each sampling level using 10 different fixed seeds so the analysis could be replicated. We aggregated the 10 runs for each sample size into average percentage of shipped files for Windows 8.1 covered by RASA, average percentage of vulnerabilities covered, and the standard deviation of our random runs for both statistics.

TABLE I. RESULTS OF RISK-BASED ATTACK SURFACE APPROXIMATION ANALYSIS ON MOZILLA FIREFOX.

Sample Size	Avg %files	Avg %vulns	Stdev %files	Stdev %vulns
10%	12.8%	70.9%	0.03%	0.49%
20%	13.8%	71.9%	0.03%	0.42%
30%	14.3%	72.2%	0.03%	0.34%
40%	14.7%	72.6%	0.02%	0.37%
50%	15.0%	72.8%	0.03%	0.35%
60%	15.2%	73.0%	0.03%	0.30%
70%	15.4%	73.1%	0.03%	0.35%
80%	15.5%	73.3%	0.03%	0.27%
90%	15.7%	73.4%	0.02%	0.16%
100%	15.8%	73.6%	X	X

TABLE II. RESULTS OF RISK-BASED ATTACK SURFACE APPROXIMATION ANALYSIS ON WINDOWS 8.1 .

Sample Size	Avg %files	Avg %vulns	Stdev %files	Stdev %vulns
10%	13.8%	32.0%	0.03%	0.1%
20%	16.6%	35.9%	<0.01%	0.1%
30%	18.3%	38.1%	<0.01%	0.08%
40%	19.6%	39.5%	<0.01%	0.1%
50%	20.7%	40.8%	.<0.01%	0.1%
60%	21.4%	41.9%	<0.01%	0.1%
70%	22.1%	42.5%	<0.01%	0.1%
80%	22.7%	43.3%	<0.01%	0.07%
90%	23.2%	44.2%	<0.01%	0.05%
100%	23.6%	44.5%	X	X

VI. RESULTS

In this section, we present our results and discuss what each of the results means for security professionals.

A. *Attack Surface Approximation (RQ1)*

RQ1: How effective is risk-based attack surface approximation in predicting the location of vulnerabilities?

After applying RASA to Mozilla Firefox, 15.8% of files contained 73.6% of the vulnerabilities in our study. When applied to Windows 8.1, 11.6% of files contained 20.2% of vulnerabilities in our study. The initial study on Windows 8 found that 48.4% of binaries contained 94.8% of historical vulnerabilities when filtering by binaries with a minimum of one appearance on a stack trace.

We have 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. We have lost vulnerability coverage in comparison to the original study, indicating a tradeoff as we modify RASA for more practical levels of granularity.

B. *Random Sampling (RQ2)*

RQ2: How does random sampling of crash dump stack traces affect the variability and effectiveness of the resulting risk-based attack surface approximation in predicting the location of security vulnerabilities?

The average number of files covered by RASA and the average number of security vulnerabilities covered by RASA at various random sampling points is found in Table 1 for Mozilla Firefox and Table 2 for Microsoft Windows 8.1. As the size of the random sampling increases, we see that the

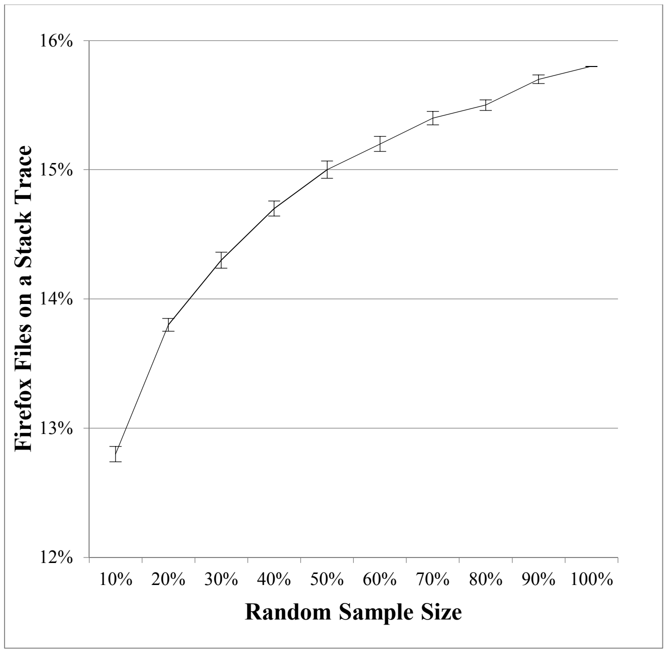

Figure 5. Graph of the percentage of files included on the RASA at random samples for Microsoft Windows.

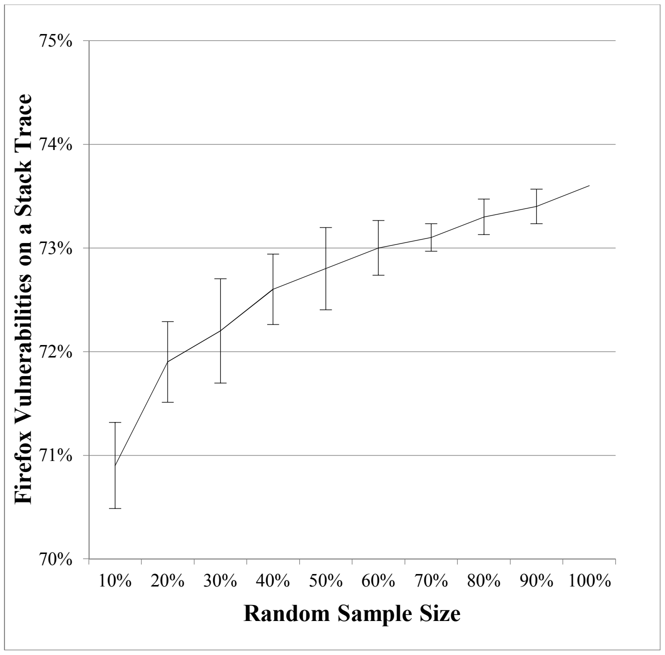

Figure 6. Graph of the percentage of vulnerabilities covered by RASA at random samples for Mozilla Firefox.

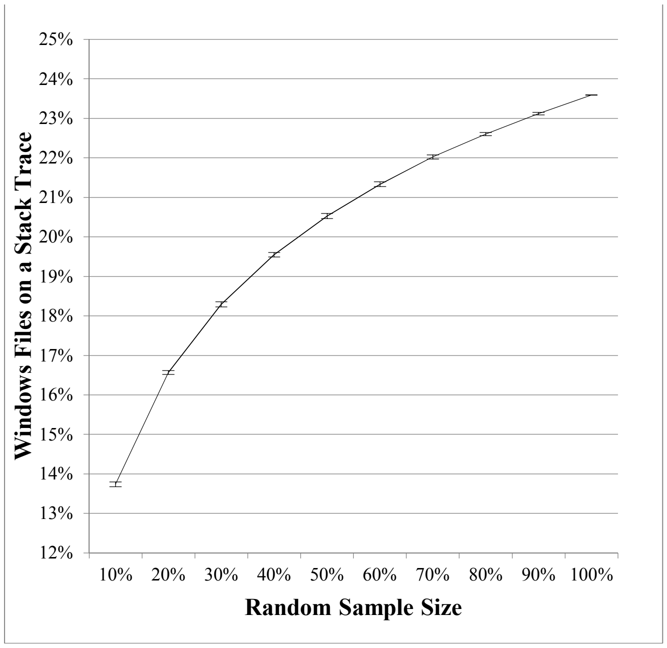

Figure 7. Graph of the percentage of files included on the RASA at random samples for Microsoft Windows.

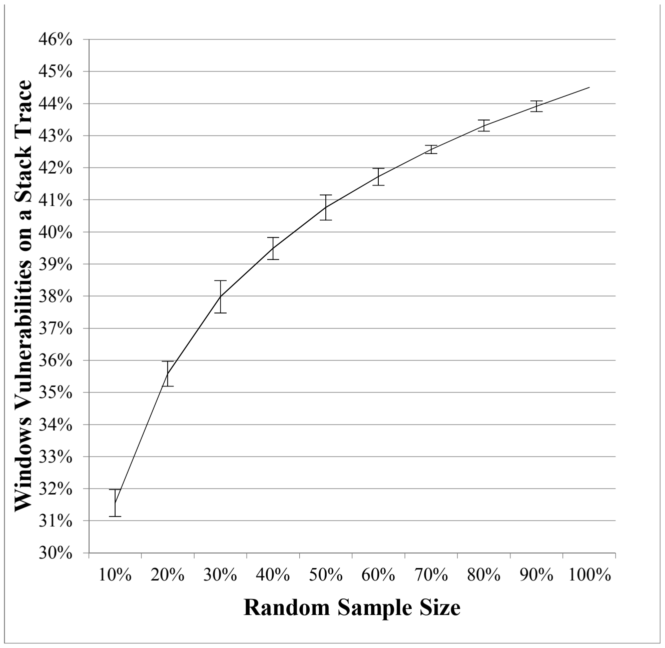

Figure 8. Graph of the percentage of vulnerabilities covered by RASA at random samples for Microsoft Windows.

average 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.

From these results, we conclude that randomly sampling stack traces for the two datasets do not result in appreciable changes in code coverage or vulnerability coverage from run to run. The standard deviation in both cases is small in comparison with the total number to be insignificant in practical terms for determining what code is on the RASA.

A graph of vulnerability coverage for our implementation of RASA for Mozilla Firefox is found in Figure 5. In this graph, we observe that the difference between using 10% of the total crashes of Firefox versus 100% of the total crashes is less than 4% in total vulnerability coverage. Additionally, we see that the trend line levels off quickly, with less than 1% difference in total coverage of vulnerabilities starting at a 40% sampling of crashes. We conclude from this graph that random sampling of crashes has a minimal effect on RASA's ability to cover security vulnerabilities.

A graph of total file coverage (the number of files on RASA versus the total number of files in the system) for Firefox is found in Figure 6. From this graph, we see an increase in the number of files included on the RASA as the sample size increases. From Table 1 and Figures 5 and 6, we can see points of diminishing returns for vulnerability coverage in terms of the tradeoff of additional files on the RASA. From a 70% random sample to a complete set of crashes, we only cover an additional 2 vulnerabilities while adding about 200 files to the RASA, at the cost of the storage and analysis of 30% more data. This result suggests that randomly sampling stack traces could be an effective technique for reducing the cost of running RASA. In a timing test done on the Firefox sample, parsing the complete set of stack traces on a 2013 MacBook Pro using Python scripts took 23 minutes, while the 10% sample took 87 seconds. These results will vary greatly depending on the number of stack traces in a given sample, the clock speed of the system the task is running on, the available memory in the system, and the code for parsing the stack traces.

VII. WHY DOES SAMPLING WORK?

Our data shows that random sampling of crashes from a large dataset does not cause appreciably different results for RASA compared to RASA run over a complete set of available stack traces. Our intuition told us that random sampling would cause an equivalent drop in coverage of security vulnerabilities: why is this not the case?

To explore this idea, we present Table 3, which is a subset of the list of files in Mozilla Firefox. We sorted the list of files by those that had a security vulnerability fix associated with them, and then by the number of times the file was seen in a crash. We then found the point where files had a vulnerability fix, but never appeared in a crash dump stack trace.

From the table, only 6 files associated with a security vulnerability fix appeared in only one stack trace. Returning to Table 1, we see that the difference in total vulnerability coverage from a 10% sample to the complete set of crashes is 11 files.

TABLE III. SNAPSHOT OF FILES WHERE VULNERABILITY COVERAGE STOPS, WHEN SORTY BY THE NUMBER OF TIMES A FILE WAS SEEN IN A CRASH.

File Name	Crashes	Vulnerability?
`js/src/liveconnect/jsj_JavaClass.c`	6	1
`js/src/liveconnect/jsj_JavaArray.c`	6	1
`docshell/base/nsDocShellEnumerator.h`	6	1
`js/src/jsdbgapi.h`	3	1
`modules/libpr0n/decoders/nsIconDecoder.cpp`	2	1
`layout/generic/nsPageContentFrame.h`	1	1
`layout/xul/base/src/tree/src/nsTreeContentView.h`	1	1
`content/base/src/nsNodeIterator.h`	1	1
`modules/oji/src/nsCSecurityContext.h`	1	1
`layout/xul/base/src/tree/src/nsTreeSelection`	1	1
`js/src/jslock.h`	1	1
`security/nss/cmd/strscint/strscint.c`	0	1

This observation could explain why random sampling had a minimal effect on vulnerable file coverage. For a vulnerable file to no longer be covered by RASA, it cannot appear in *any* stack trace from a crash in the target system. When taking random samples of crashes, you remove a set percentage of stack trace lines for potential analysis, but your sample must not catch any occurrence of a file to not include it on the RASA. 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 data 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. Additional studies on smaller projects are needed to confirm our result, as smaller project crashes may not follow a random distribution.

VIII. LIMITATIONS

One of the limitations of our previous work was that the RASA approach was only tested on Microsoft Windows and that the approach may not have been generalizable. In this study, we have demonstrated the value of RASA on Mozilla Firefox, but smaller software systems with fewer stack traces may not work as well with the approach. Both RASA studies have been done on industry leading codebases. Future studies could determine how RASA performs on smaller codebases.

In the absence of an oracle for the complete attack surface, we cannot assess the completeness of our approximation. Our determination of accuracy currently is based only on known vulnerabilities, which may introduce a bias towards code previously seen to be vulnerable. While basing our effectiveness on historical vulnerabilities may be a good assumption, further exploration is needed. RASA outputs, as expected, an approximation, and it cannot identify latent vulnerabilities directly.

IX. Conclusion

In this paper, we have evaluated the effectiveness of RASA for Mozilla Firefox, and confirmed our previous result that crash dump stack traces can help practitioners prioritize code with vulnerabilities. We have analyzed the effect of random sampling of crash dump stack traces on the final result of RASA, and concluded that random sampling is an effective technique for reducing the amount of data required to use RASA. Finally, by moving granularity to file level in this study, we have made the approximation more actionable for developers. Files are a more efficient unit of measure for locating vulnerabilities as compared to binaries.

In the previous study [1], we explored graph representations of stack traces using the order of appearance of code in the trace. For future work, we would like to construct similar graph representations of the trace. A standalone tool or plugin integrated with a modern IDE such as Eclipse is one method for making this representation useful to practitioners.

Mock examples of the types of graph representations we could create are in Figure 1. By showing known failing data paths to the developer, they can focus their triaging efforts on these paths, excluding any paths that crashes were not seen on. By following the visualization, the developer focuses their effort on code that has a higher probability of containing security defects.

In addition to the visualization of the graph representation of the stack traces, graph shape analysis is another methodology we plan to explore to further narrow our scope of code that could contain security vulnerabilities. Do certain shapes of incoming and outgoing nodes result in more frequent sightings of vulnerabilities? We hypothesize that certain shapes, such as many files calling into one file but that file only calling out to few files, may exhibit more vulnerabilities than other areas.

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.

RASA currently looks at the code entities themselves as possible locations for security vulnerabilities. The code entities themselves may not 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. Future work may be prudent to examine 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 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 the generated attack surface approximation 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 the attack surface. The more a code artifact is involved in crashes, the more likely it might be that that code artifact has a related security vulnerability.

X. Acknowledgements

Any opinions expressed in this report are those of the author(s) and do not necessarily reflect the views of North Carolina State University. We also thank the Realsearch research group for providing helpful feedback on this work. We also thank Rahul Pandita and Patrick Morrison for sharing research into security defect locations in the Mozilla Firefox product.

XI. References

[1] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, “Approximating Attack Surfaces with Stack Traces”, in *Companion Proceedings of 37th ICSE,* 2015.

[2] R. Moser, W. Pedrycz and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *Proceedings of the 30th ICSE,* 2008.

[3] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in*. Proceedings of the 27th ICSE*, 2005.

[4] T. Zimmermann and N. Nagappan, "Predicting defects using network analysis on dependency graphs," in *Proceedings of the 30th ICSE*, 2008.

[5] M. Pinzger, N. Nagappan and B. Murphy, "Can developer-module networks predict failures?," in *Proceedings of the 16th ACM SIGSOFT FSE*, 2008.

[6] N. Nagappan, B. Murphy and V. Basili, "The Influence of Organizational Structure on Software Quality: An Empirical Case Study," in *Proceedings of the 30th ICSE*, 2008.

[7] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st ICSE*, 2009.

[8] K. Herzig, S. Just, A. Rau and A. Zeller, "Predicting Defects Using Change Genealogies," in *Proceedings of the 24th IEEE ISSRE*, 2013.

[9] K. Herzig, "Using Pre-Release Test Failures to Build Early Post-Release Defect Prediction Models," in *Proceedings of the 25th IEEE ISSRE*, Neaples, 2014.

[10] T. Hall, S. Beecham, D. Bowes, D. Gray and S. Counsell, "A Systematic Literature Review on Fault Prediction Performance in Software Engineering,", *IEEE Transactions on Software Engineering,* vol. 38, pp. 1276--1304, 2012.

[11] T. Zimmermann, N. Nagappan and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in Proceedings of the 3rd *ICST,* 2010.

[12] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?," *Empirical Software Engineering,* vol. 18, pp. 25--59, 2013.

[13] M. Gegick, L. Williams, J. Osborne and M. Vouk, "Prioritizing software security fortification throughcode-level metrics," in *Proceedings of the 4th ACM workshop on Quality of protection*, 2008.

[14] Y. Shin, A. Meneely, L. Williams and J. Osborne, "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *Software Engineering, IEEE Transactions on,* vol. 37, pp. 772--787, 2011.

[15] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture,* vol. 57, pp. 294--313, 2011.

[16] S. Neuhaus, T. Zimmermann, C. Holler and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[17] M. Howard, J. Pincus and J. M. Wing, "Measuring Relative Attack Surfaces," in *Computer Security in the 21st Century*, Springer US, 2005, pp. 109-137.

[18] B. Liblit and A. Aiken, "Building a Better Backtrace: Techniques for Postmortem Program Analysis," University of California, Berkeley, Berkeley, 2002.

[19] R. Manevich, M. Sridharan, S. Adams, M. Das and Z. Yang, "PSE: Explaining Program Failures via

[20] Postmortem Static Analysis," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004.

[21] W. Jin and A. Orso, "F3: Fault Localization for Field Failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[22] R. Wu, H. Zhang, S.-C. Cheung and S. Kim, "CrashLocator: Locating Crashing Faults Based on Crash Stacks," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[23] S. Wang, F. Khomh and Y. Zou, "Improving bug localization using correlations in crash reports," in *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, 2013.

[24] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*, 2003.

[25] 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 International Conference on Software Engineering*, 2012.

[26] S. Kim, T. Zimmermann and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *Dependable Systems Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, 2011.

[27] P. J. Guo, T. Zimmermann, N. Nagappan and B. Murphy, "Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows," in *Proceedings of the 32th International Conference on Software Engineering*, 2010.

[28] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj and T. Zimmermann, "What makes a good bug report?," in *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, 2008.

[29] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu and C.-W. Lai, "Software Crash Analysis for Automatic Exploit Generation on Binary Programs," *Reliability, IEEE Transactions on,* vol. 63, pp. 270-289, March 2014.

[30] C. Holler, K. Herzig and A. Zeller, "Fuzzing with Code Fragments," in *Proceedings of the 21st USENIX Conference on Security Symposium acmid = 2362831*, 2012.

[31] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung and S. Park, "Which Crashes Should I Fix First?: Predicting Top Crashes at an Early Stage to Prioritize Debugging Efforts," *Software Engineering, IEEE Transactions on,* vol. 37, no. 3, pp. 430-447, 2011.

[32] *Building Security In Maturity Model (BSIMM)*

[33] Manadhata, P., Wing, J., Flynn, M., & McQueen, M. (2006, October). Measuring the attack surfaces of two FTP daemons. In *Proceedings of the 2nd ACM workshop on Quality of protection* (pp. 3-10). ACM.

[34] Younis, A.A., Malaiya, Y.K., Ray, I., "Using Attack Surface Entry Points and Reachability Analysis to Assess the Risk of Software Vulnerability Exploitability" In *Proc. of IEEE 15th International Symposium on High-Assurance Systems Engineering,* p. 1-8, 2014

[35] Shin, Y. and Williams, L., *Can Fault Prediction Models and Metrics be Used for Vulnerability Prediction?*, Empirical Software Engineering, Vol. 18, No. 1, pp. 25-59, 2013.

[36] N. Chawla, V. Nitesh, et al. "SMOTE: synthetic minority over-sampling technique."*Journal of artificial intelligence research* 16.1 (2002): 321-357.

[37] A. Meneely, and L. Williams. "Secure open source collaboration: an empirical study of linus' law." *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009

[38] S. Lessmann, B. Baesens, C. Mues and S. Pietsch, "Benchmarking classification models for software defect prediction: A proposed framework and novel findings." in *IEEE Transactions on Software Engineering*, 34(4), 485-496, 2008.

[39] L. Pelayo and S. Dick, "Applying novel resampling strategies to software defect prediction." *Annual Conference of the North American Fuzzy Information Processing Society - NAFIPS*, 69-12, 2007.

[40] T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald, "Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors.'"" *IEEE Trans. Softw. Eng. 33, 9 (September 2007),* 637-640. 2007

[41] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[42] K. Madahata and J. Wing, "An attack surface metric." *Software Engineering, IEEE Transactions on* 37.3 (2011): 371-386