ABSTRACT

THEISEN, CHRISTOPHER RICHARD. Risk-Based Attack Surface Approximation. (Under the direction of Laurie Williams.)

**Motivation:** Security testing and reviewing efforts are a necessity for software projects, but are time-consuming and expensive to apply. Organizations look to utilize security professionals as efficiently as possible. Identifying vulnerable code supports decision-making during all phases of software development. Potentially vulnerable code identified early in the development of software can facilitate “building security in” before release. Identifying potentially vulnerable code after release can prevent potential vulnerabilities from becoming major breaches. However, identification of potentially vulnerable code is difficult, as vulnerabilities are rare.

**Research Problem:** Prioritizing security reviews is difficult, as we lack effective metrics for determining what code should be considered security relevant.

**Objective:** 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.

**Approach:** We propose Risk-Based Attack Surface Approximation (RASA), an approach to determine the attack surface of software systems by using crash dump stack traces and the code that appears on them as a metric for prioritizing security reviews. We call our approach “risk-based” as it identifies the “riskiest” code in the system. Crashes are empirical evidence that the code involved in the crash was executed in an unexpected way, in the same way that vulnerabilities are often instances of code executed in an unexpected way. In the case of a crash, some outside input was able to cause the state in the software system that caused the crash conditions. The same types of input and states could be correlated with input and states that result in vulnerabilities.

**Results:** We have performed studies on the effectiveness of RASA on Microsoft Windows and Mozilla Firefox, including the effect of randomly sampling crash dump stack traces from the targeted software system. For Windows, 48.4% of binaries appeared on the approximated attack surface measured by RASA, while 94.6% of historical vulnerabilities occurred on the approximated attack surface. For Firefox, 15.8% of files appeared the approximated attack surface, while 73.6% of historical vulnerabilities occurred on the approximated attack surface. We developed a set of attack surface related metrics based on the concepts of a software system changing over time, code complexity, and entry and exit points. We found that over 50% of the approximated attack surface for Windows changes from version to version, that fan-out complexity metrics based on crash data are a stronger predictor of vulnerable code than just the approximated attack surface, and that vulnerabilities are more likely to occur directly on entry and exit points. We compared RASA against the state of the art in vulnerability prediction on Mozilla Firefox, with RASA and a broad sampling of software metrics having the best predictive performance ($F1 = 0.09$). A VPM combining RASA, software metrics,
and text mining approaches for vulnerability prediction having stronger predictive performance than any individual model (F1 = 0.28). We also determined specific classifications and severities of vulnerabilities that RASA covers. RASA covers the most severe vulnerabilities seen for Mozilla Firefox, and covers 14 of 16 classifications of vulnerabilities at a rate higher than 85%.
Risk-Based Attack Surface Approximation

by
Christopher Richard Theisen

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2018

APPROVED BY:

______________________________  ________________________________
Timothy Menzies                William Enck

______________________________  ________________________________
David Dickey                    Laurie Williams
Chair of Advisory Committee
DEDICATION

Dedicated to my wife Kelly, whose unconditional love and support made this possible.
ACKNOWLEDGEMENTS

It is difficult for me to name everyone who has had an impact on this dissertation. Deepest apologies to anyone I forgot.

To Laurie Williams, for taking me under your wing and helping me grow as a researcher and as a person. Thank you for your patience, expertise, kindness, and understanding. Thank you for the pushes when I needed them and the compassion when I needed the opposite.

To Brendan Murphy, who I think of as a co-advisor in all but name. Much of the research in this document would not have been possible without your support, and I am grateful to you for the advice you have given personally and professionally along the way.

To Kim Herzig, who was the first person who dealt with me as a brand new researcher. Thank you for your patience and guidance over the years.

To my committee members Tim Menzies, Will Enck, and Jon Stallings for their time, support, and advice. Each of you have provided key insights and advice throughout this work.

To all the NCSU faculty and staff who provided feedback and help along the way. I especially acknowledge Katie Stolee, Emerson Murphy-Hill, Chris Parnin, Munindar Singh, David Wright, Carol Allen, Kathy Luca, and Andrew Sleeth, along with my committee members.

To my Realsearch colleagues past and present: thanks for the feedback, collaborations, advice, and laughs over the last five years: Sarah Heckman, Andy Meneely, John Slankas, Pat Francis, Eric Helms, JeeHyun Hwang, Da Young Lee, Jason King, Rahul Pandita, Akond Rahman, Maria Riaz, Akond Rahman, Sarah Elder, and Rezvan Hezaveh, among others. I am lucky to have met you all, and each of you have left a small stamp on this document (and me).

To all my colleagues and friends from graduate school and beyond: thanks for screaming into the abyss with me. I cannot possibly name everyone, but especially acknowledge the following alongside Realsearch colleagues: Prairie Rose and Dan Goodwin, Sean Mealin, Michelle Hall, Kevin Lubick, Titus Barik, Denae Ford, Justin Smith, Brittany Johnson, Jim Witschey, Sheldon Abrams, Nuthan Munaiha, Dennis and Nina Murphy, Rob and Melinda Shutler, Carrie Landes, Meredith and Thomas Lopez, Jennifer and Darren Lipper, Jon Rust, Ben Merkel, Corry Paxson, and Corey Downing. Thanks for collaborations, kind words, timely advice, and needed escapes.

To Cynthia Theisen, who made me realize that a PhD was something I could actually pursue. You laid the groundwork that led me here.

To my family: Richard Theisen, Lisa Staats, Kayla Theisen, and Caitlin Theisen. Thank you for your patience, understanding, love, and support while I went missing for five years doing a PhD.

And most importantly to my wife, Kelly Theisen. It is impossible to put into words the impact you had on this document and on me. You are my biggest supporter, my most enthusiastic cheerleader, and my best friend. Each day is made brighter by your love and support.
# TABLE OF CONTENTS

LIST OF TABLES ................................................................. viii

LIST OF FIGURES .............................................................. ix

Chapter 1  INTRODUCTION ................................................... 1

Chapter 2  ATTACK SURFACE DEFINITIONS ............................... 6
  2.1 Introduction .................................................................... 6
  2.2 Systematic Literature Review ......................................... 7
    2.2.1 Definition Citations .............................................. 7
    2.2.2 Definition Granularity ........................................... 8
    2.2.3 Clusters of Definitions ......................................... 9
    2.2.4 Results .......................................................... 10
  2.3 Discussion .................................................................... 11

Chapter 3  RELATED WORK .................................................. 12
  3.1 Crash Dump Stack Traces .............................................. 12
  3.2 Vulnerability Prediction and Prioritization ....................... 13
  3.3 Classifying Defects .................................................... 14
  3.4 Classifying Vulnerabilities ........................................... 15
  3.5 Attack Surface .......................................................... 15
    3.5.1 Attack Surface Research ..................................... 15
    3.5.2 Attack Surface Tools .......................................... 16

Chapter 4  RISK-BASED ATTACK SURFACE APPROXIMATION .... 18
  4.1 Motivation .............................................................. 19
  4.2 Why Approximation? .................................................. 20
  4.3 Crash Dump Stack Traces .............................................. 21
  4.4 Risk Based ............................................................. 22
  4.5 Process ................................................................. 23
    4.5.1 Collect Crash Data ............................................. 23
    4.5.2 Parse Stack Trace ............................................... 24
    4.5.3 Map Stack Trace Data To Source Code .................... 26
    4.5.4 Collect Vulnerability Data ..................................... 27

Chapter 5  VULNERABILITY COVERAGE .................................. 29
  5.1 Introduction ............................................................ 29
  5.2 Methodology ........................................................... 30
    5.2.1 Risk-Based Attack Surface Approximation ............... 30
    5.2.2 Random Sampling .............................................. 31
  5.3 Windows Case Study .................................................. 32
    5.3.1 Data Sources ................................................... 32
    5.3.2 Data Collection ................................................ 33
7.4.2 Coverage of Vulnerability Severity ................................................................. 67
7.5 Discussion ........................................................................................................... 68
7.6 Limitations ......................................................................................................... 69

Chapter 8  VULNERABILITY PREDICTION MODEL COMPARISON ......................... 71
8.1 Motivation .......................................................................................................... 71
8.2 Vulnerability Prediction Model - Windows 8 ......................................................... 72
  8.2.1 Crash Dump Stack Traces .............................................................................. 73
  8.2.2 Code Metrics ............................................................................................... 74
  8.2.3 Pre and Post-Release Vulnerabilities ............................................................ 75
  8.2.4 Prediction Models ......................................................................................... 75
  8.2.5 Data Sources ................................................................................................ 76
  8.2.6 Results ......................................................................................................... 77
8.3 Vulnerability Prediction Models Comparison - Firefox ........................................ 77
  8.3.1 Software Metrics - Churn and Complexity ................................................... 77
  8.3.2 Text Mining .................................................................................................. 78
  8.3.3 Methodology ................................................................................................ 78
  8.3.4 Mozilla Firefox Case Study .......................................................................... 80
  8.3.5 Results ......................................................................................................... 87
  8.3.6 VPM Combinations ..................................................................................... 88
  8.3.7 VPM Resource Usage .................................................................................. 90
8.4 Discussion .......................................................................................................... 90
  8.4.1 VPM Comparisons ....................................................................................... 90
  8.4.2 VPM Combinations ..................................................................................... 92
  8.4.3 Feature Generation and Practical Usage ....................................................... 92
8.5 Limitations .......................................................................................................... 94

Chapter 9  ATTACK SURFACE MEASUREMENT COMPARISON ................................. 96
9.1 Motivation .......................................................................................................... 96
9.2 Munaiah Attack Surface ...................................................................................... 97
9.3 Younis Attack Surface ......................................................................................... 98
9.4 Methodology ...................................................................................................... 98
9.5 Case Study - Wireshark ...................................................................................... 99
9.6 Results and Discussion ....................................................................................... 100
9.7 Limitations ......................................................................................................... 101

Chapter 10  CONTRIBUTIONS AND FUTURE WORK .............................................. 102
10.1 Contributions .................................................................................................... 102
10.2 Summary .......................................................................................................... 103
10.3 Future Work ...................................................................................................... 104
  10.3.1 Crash Data Quality ..................................................................................... 104
  10.3.2 Dynamic Software Metrics ......................................................................... 104
  10.3.3 Oracles for Vulnerability Studies ................................................................. 105
  10.3.4 Framing Results ........................................................................................ 106
# LIST OF TABLES

Table 4.1 An example of a database or file format for summarizing crash dump stack trace data.  

Table 5.1 Examples of files seen in the topmost_filename field in the Firefox crash dumps.  
Table 5.2 Descriptive statistics for attack surface at binary level, broken down by type of stack trace category that identified the binary.  
Table 5.3 Results of randomly sampling crash dump stack traces while implementing Risk-Based Attack Surface Approximation on Microsoft Windows 8.1.  
Table 5.4 Results of randomly sampling crash dump stack traces while implementing Risk-Based Attack Surface Approximation on Mozilla Firefox.  
Table 5.5 Snapshot of files with associated vulnerabilities where the number of times the file was found in a crash was six or less, sorted by total number of crashes.  
Table 6.1 Number of crash dump stack traces parsed in 2014 and 2015, by OS version.  
Table 6.2 Percentage of files appearing on crash dump stack traces and vulnerabilities appearing on crash dump stack traces for operating system/year pairings.  
Table 6.4 Spearman rank-order correlation coefficient for Fan-In and Fan-Out values compared to the log2ratio of the SF and VF metrics for Windows 8, 8.1, and 10.  
Table 6.5 Percentage of files appearing on the boundary of Windows, as measured by crash dump stack traces, and the percentage of files with vulnerabilities in the set.  
Table 7.1 The list of keywords used to collect potential unlabeled vulnerabilities from Mozilla's Bugzilla database on Mozilla Firefox.  
Table 8.1 List of features included in our best-effort replication of the VPM described by Zimmermann et al. [Zim10]  
Table 8.2 List of features included in our best-effort replication of the VPM described by Shin et al. [Shi11]. Descriptions for complexity metrics from SciTools [Sci17].  
Table 8.3 Median Precision, Recall, and F1 score for each Vulnerability Prediction Model for Mozilla Firefox, using a Random Forest Classifier.  
Table 8.4 Median Precision, Recall, and F1 score for each Vulnerability Prediction Model for Mozilla Firefox, using a Gaussian Naive Bayes Classifier.  
Table 8.5 The relative importance of the top 10 features used in the trees featured in the Random Forest Classifier in the “All” model. Quotes indicate a token from the text mining approach.  
Table 8.6 Resource usage statistic for each Vulnerability Prediction Model on Mozilla Firefox.
LIST OF FIGURES

Figure 4.1 A graph representation of the attack surface of a software system. Blue nodes represent code that does not appear on the attack surface, while red nodes indicate code on the attack surface. .......................................................... 19

Figure 4.2 A Venn diagram representing the relationship between a vulnerability, the consequence of the vulnerability, and the threat posed by the vulnerability. Each of these factors add up to the overall risk of the vulnerability being exploited. .......................................................... 23

Figure 4.3 Example of an anonymous stack trace. From the stack trace, we can identify binary, function, and sometimes source file information. Each of these stack traces can be transformed into a graphical representation. Each stack trace contributes one particular path (see middle) to the overall attack surface graph (see right hand side). .......................................................... 24

Figure 5.1 Graph of the percentage of files included on the RASA at random samples for Mozilla Firefox. .......................................................... 40
Figure 5.2 Graph of the percentage of vulnerabilities covered by RASA at random samples for Mozilla Firefox. .......................................................... 41
Figure 5.3 Graph of the percentage of files included on the RASA at random samples for Microsoft Windows. .......................................................... 41
Figure 5.4 Graph of the percentage of vulnerabilities covered by RASA at random samples for Microsoft Windows. .......................................................... 42

Figure 6.1 The overall design of the database used to collect crash dump stack trace data and relate it to code artifacts in Windows 8, 8.1, and 10. Boxes represent individual tables. The full database schema design is available online. ........... 48
Figure 6.2 A trendline of the distribution of files in Windows 8, based on the number of incoming and outgoing calls from a file. .......................................................... 57
Figure 6.3 A trendline of the distribution of files in Windows 8.1, based on the number of incoming and outgoing calls from a file. .......................................................... 58
Figure 6.4 A trendline of the distribution of files in Windows 10, based on the number of incoming and outgoing calls from a file. .......................................................... 59

Figure 7.1 Distribution of classifications of vulnerabilities occurring on crashing code versus classifications occurring on non-crashing code. The percentage of vulnerabilities covered for a specific classification is next to the label. Vulnerability classifications with no entries in our dataset are omitted. ............... 66
Figure 7.2 Distribution of severity of vulnerabilities occurring on crashing code versus severities occurring on non-crashing code. The percentage of vulnerabilities covered for a specific severity is next to the label. ............... 68

Figure 8.1 Precision values for each Vulnerability Prediction Model, along with the combinations of each model. .......................................................... 91
Figure 8.2  Recall values for each Vulnerability Prediction Model, along with the combinations of each model. ................................. 93
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 [BM15]. 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 resources spent working on finding and fixing that vulnerability could be used elsewhere. 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 [How05]. Later, Manadhata and Wing [MW11] 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.

An approach for prioritizing security effort using the concept of attack surface would help security professionals 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. However, the software security community still lacks a practical means of identifying the parts of the system that are on the attack surface. An automated approach for identifying the attack surface of a software system would be useful for practitioners so their security
testing and review efforts could be targeted effectively and efficiently.

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.

To fill the need for a practical understanding of the attack surface of a software system, 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 the RASA. Code that appears in stack traces caused by user activity is on the attack surface by definition, because it appears in a code path reached by outside input.

Crash dump stack traces from user-initiated crashes represent user activity that puts the system under stress. In our research, we provide an assessment of the degree to which crash dump stack traces support the identification of attack surfaces. We call our approach “Risk-Based Attack Surface Approximation” because we hypothesize that RASA is an efficient means of identifying the part of the attack surface that has a high risk of containing vulnerabilities. RASA is a useful approach since it is scalable, as crash dump stack traces are relatively easy to parse. The results from RASA can be widely used in industry since many organization collect crash dump stack traces already, such as Microsoft and Mozilla. Finally, unlike other approaches, RASA does not require complex language-specific code analysis tools since it is a report of the dependencies between binaries, files or methods. Hence, RASA can be applied to systems that are combinations of different programming languages.

Our research hypothesis is that binaries and source code files that appear in crash dump stack traces from a software system are more likely to have exploitable vulnerabilities than binaries and source code files that do not appear in crash dump stack traces.

To test the above hypothesis, we propose the following research questions:

Approximation (RQ1): Are vulnerabilities more likely to occur on in code appearing on crash dump stack traces than code not appearing on crash dump stack traces?

We first study whether historical vulnerabilities correlate with the attack surface generated by the RASA approach for Windows 8. If historical vulnerabilities are found to correlate with the attack surface, RASA could be used by software security teams to prioritize their efforts. We also explore the effect of random sampling of crash dump stack trace data on the RASA approach on three versions of Windows, along with Mozilla Firefox. Exploring the data requirements of RASA is necessary to see if the approach is usable by projects of varying sizes. Random sampling of crash dump stack traces without replacement (not choosing the same crash dump stack trace multiple times in the same sample) may be an effective simulation of situations where less data is available. For Windows 8 at the binary and file level and for Mozilla Firefox at the file level, RASA indicates that vulnerabilities are more frequently found in crashing code rather than non-crashing code.

The results and feedback suggested further prioritization and refinement of the approach. In
particular, more specific metrics will be needed to move from a proof-of-concept to an approach that can be used by practitioners on live software projects. With the goal of developing additional metrics for practitioners to use, we asked the following questions regarding metric generation using crash dump stack traces:

**Change (RQ2):** Does the approximated attack surface need to be re-computed with each new release or periodically during development?

Knowing if the attack surface of software can change over time, as measured by RASA, is useful to understand whether the calculated attack surface of a system is static or needs to be recomputed periodically to provide meaningful results. We found that recomputing RASA as frequently as weekly during active development and at minimum once per version change will allow practitioners to keep up with attack surface changes as the software changes.

**Complexity (RQ3):** How are security vulnerabilities correlated with code complexity metrics such as fan-in and fan-out, generated from crash dump stack traces?

McCabe et al [McC76] first described the concept of code complexity, specifically fan-in and fan-out measures. Shin et al. [Shi11] found a positive correlation between the presence of vulnerabilities and code complexity metrics, specifically fan-out. We explore whether fan-in and fan-out measures calculated from code graphs created from crash dump stack traces also follow their result. We found that fan-in and fan-out are both correlated with the likelihood of a security vulnerability fix being present in a source code file.

**Boundary (RQ4):** Is reviewing data flows in a software system necessary to find a majority of security vulnerabilities, or can entry and exit points be prioritized for security review?

A previous study by Manadhata et al. [Man06] measured the attack surface of an FTP daemon by an API scanning approach which found the entry and exit points to the system. Our definition of attack surface includes all of the paths data takes throughout a system, but the concept of an attack surface boundary may be useful for prioritization efforts. We analyzed whether code on the boundary of systems is more likely to have vulnerabilities than the rest of the codebase. Code included on RASA for Windows 8, 8.1, and 10 was twice as likely to have vulnerabilities as non crashing code, while code that was included on RASA and was also on the boundary of the software system was four times likely to have vulnerabilities compared to non crashing code.

Next, we explore vulnerabilities themselves. Accounting for the classifications and severities of vulnerabilities is important in vulnerability prediction studies [MM16]. Some vulnerabilities are more severe than others, and some vulnerability types may not be covered by the RASA approach.

**Vulnerability Classifications (RQ5):** What classifications and severities of vulnerabilities appear on an attack surface approximated by crash dump stack traces in comparison to vulnerabilities that are not on crash dump stack traces?

For Mozilla Firefox, 14 classifications of vulnerabilities were present, with RASA covering 11 of them at a rate higher than 90%. The distribution of the most severe vulnerabilities in Firefox was
proportional to the coverage of all vulnerabilities in Firefox.

The RASA approach could be useful as an input into Vulnerability Prediction Models (VPMs) that have been proposed. RASA could help these tools overcome the inherent scarcity problem of vulnerabilities for statistical learners by removing a large percentage of the codebase.

**VPM Comparison (RQ6):** Does risk-based attack surface approximation improve the precision and recall values of vulnerability prediction models when predicting the location of vulnerabilities in software system?

Our study comparing VPMs included RASA, along with two software metric based approaches by Shin et al. [Shi11] and Zimmermann et al. [Zim10], and a bag of words text mining approach by Scandariato et al. [Sca14]. RASA has the best reported recall at .86 in our comparison of individual VPMs. In addition, combining RASA with the text mining approach resulted in a significant improvement in precision, albeit at the cost of recall.

We compare RASA approach to other attack surface measurement research: the Younis et al. [You14] scanning approach and Munaiah’s proximity approach [MM16]. We compare these approaches in terms of the size of the resultant code to be analyzed, the percentage of vulnerabilities covered, and time to set up and run the approach. Collecting information about the predictive power of each approach along with the resource requirements of each means practitioners can make an informed decision on what approach is best for them.

**Attack Surface Comparison (RQ7):** How does risk-based attack surface approximation compare with the state of the art of attack surface approximation in terms of computation time, size of the resultant attack surface, and presentation of feedback?

We found

We have completed the following studies to address the seven research questions proposed above. Each study description is followed by a list of research question that study addresses:

- Evaluation of whether risk-based attack surface approximation is feasible for prioritizing code artifacts that may contain vulnerabilities. (RQ1) [The15c; The15a]

- Evaluation of random sampling of crash dump stack traces and whether the resultant iteration of RASA still allows for vulnerability prioritization. (RQ1) [The17]

- An exploration of “advanced metrics” for risk-based attack surface approximation derived from crash dump stack traces, such as the change in attack surface over time, the relationship between code complexity and vulnerabilities, and the relationship between the boundary of software systems and vulnerabilities. (RQ2, RQ3, RQ4)

- Profiling the types of vulnerabilities that appear on and off RASA, and whether vulnerabilities of certain types are more or less likely to be covered by RASA. (RQ5) [The18c]
• A comparison of VPMs before and after RASA data is applied to limit the amount of code being used as part of the model (RQ6) [The18b]

• A comparison of the RASA approach with the state of the art in attack surface research (RQ7)

We make the following contributions to the research and industrial communities:

• Risk-Based Attack Surface Approximation, a process using crash dump stack traces to prioritize security testing and review efforts.

• An analysis of current definitions of attack surface and a proposal for a unified definition for the research community.

• An open source toolset to help organizations and research teams implement Risk-Based Attack Surface Approximation on their projects.

The rest of this document is organized as follows: Chapter 2 provides a short discussion of the term “attack surface” and how it is used in research literature. Chapter 3 provides background information on concept of attack surface, and work related to the studies proposed and performed in this document. Chapter 4 discusses the motivation and implementation of the RASA approach. Chapter 5 presents the initial study to determine the effectiveness of RASA, along with an exploration of the effect of random sampling on the results of RASA. Chapter 6 describes the implementation and evaluation of the additional RASA-Based metrics of change, complexity, and boundary. Chapter 7 discusses how RASA covers vulnerabilities of specific classifications and severities. Section 8 compares RASA against the state of the art in vulnerability prediction. Chapter 9 compares RASA against the state of the art in attack surface measurement and approximation. Finally, chapter 10 provides a set of conclusions about the results of the RASA studies.
CHAPTER 2

ATTACK SURFACE DEFINITIONS

2.1 Introduction

Michael Howard provided the earliest definition of the term attack surface when he defined it as “the union of code, interfaces, services, protocols, and practices available to all users, with a strong focus on what is accessible to unauthenticated users.” [How04] Software security researchers and professionals have used Howard’s concept of the attack surface to discuss the overall security posture of a system, or the effectiveness of a given security measure [How05]. For example, reducing the attack surface is one way researchers provide evidence that the system is more secure. Practitioners can also use attack surface measurements to prioritize their fortification efforts.

However, a variety of definitions exist for the phrase attack surface, which drives how researchers conduct their measurements. Recent research using the attack surface concept varies, with some prioritizing the localization of vulnerabilities, others identifying vulnerable code, and others determining how risky specific vulnerabilities are. Younis et al. [You14] analyzed the relationship between the attack surface of Apache HTTP Server and the density of vulnerabilities in the system. Munaiah et al. [MM16] used call graphs to determine the proximity of security vulnerabilities to the attack surface of the software system and found that vulnerabilities were found near the surface of the target system. Our own work [The15c; The15b; The15a; The17] uses crash dump stack traces to approximate the attack surface of a target system. Having a seminal definition (or a set of definitions) to reference as the “official” attack surface definition(s) would help clarify discussions about and
measurements of attack surfaces.

Understanding how the term attack surface is used in the research literature and establishing a baseline definition will help facilitate discussion about RASA in comparison to other approaches for security testing and review. If RASA and another approach both claim to measure or approximate the attack surface of a system, but are using different conceptual models for the term, then comparing the approaches will be difficult. Additionally, inconsistent definition between tools and approaches could confuse practitioners. They may attach to one tool's concept of attack surface and not understand why another attack surface tool doesn't model attack surfaces in the way they expect.

2.2 Systematic Literature Review

To explore the use of the phrase attack surface in the research literature, we performed a systematic literature review [The18a] of the use of the phrase, including research papers, magazine articles, and technical reports. After identifying a set of 1,433 potential papers for inclusion in our study, we selected 644 that used the phrase attack surface in some way in the text. For these papers, we determined whether they provided their own definition, cited another definition, or expected the reader to understand their use of the phrase intuitively. We identified the source of each paper, such as a specific database or aggregator. Additionally, we analyzed the level of granularity of each paper's use of attack surface. For example, some papers discussed the attack surface of an entire network of systems, while others discussed how specific functions affect the attack surface of one system.

2.2.1 Definition Citations

The full text of the relevant papers identified in the previous stage are read and categorized into categories that are based on papers' usage context of the phrase attack surface. Since the goal of our study was to understand the various definitions of the phrase attack surface used in the community, we categorized the relevant papers into one of four categories. Each paper fits into exactly one category [The18a]:

- **Define** - The paper defines, formally or informally, the phrase attack surface in the text. Papers in prior attack surface literature or papers that include a glossary with a definition of the phrase attack surface are placed into the Define category.

- **Supported Use** - The paper uses the phrase attack surface without explicitly defining it. However, the usage of the phrase is supported by citing a paper that defines the phrase.

- **Unsupported Use** - The paper uses the phrase attack surface with neither an explicit definition for it nor a citation to a paper that defines the phrase.
• Not Relevant - The paper is not relevant to our SLR. Papers that are categorized into the Not Relevant category are false positives from the previous stage.

The order of the categories presented above is the order in which the categorizations were made. For instance, if a paper is extending an existing (cited) definition of the phrase attack surface, the paper will be categorized as Define and not Supported Use.

When reading the full text of relevant paper for categorized, we also captured the association (through citation) between papers categorized as Supported Use and those categorized as Define. We also captured self-associations between papers categorized as Define in cases where a paper was extending or modifying an existing (cited) definition.

In our corpus of 644 papers that used the phrase attack surface, 71% used the term without citing a definition; that is, they assumed the meaning of the term attack surface was well understood. However, the term attack surface carried a variety of different meanings, with 48 definitions cited in the remaining 29% of papers that provided a cited definition. Authors not providing support for their use of the phrase indicates a lack of understanding of the different possible definitions, or an assumption that the audience of their paper have the same definition in mind. From the same data, we see there are 48 papers categorized as Define, or providing a definition of attack surface. These 48 papers provide 48 definitions of attack surface in the literature with varying degrees of overlap and authority. Many of these definitions are not themselves cited.

By contrast, 86 papers (or 13%) in our corpus provided Supported Use, or a citation or a footnote of their use of the phrase attack surface. These papers skewed towards security focused works. We observed that the more closely related the research was to the topic of attack surfaces, the more likely it was that the researcher supported their definition of attack surface in some way.

2.2.2 Definition Granularity

In addition to categorizing the papers based on the usage of the phrase attack surface, we categorized papers in the Define and Supported Use categories based on the granularity at which the phrase was used. We used the following categories for the papers based on the granularity of phrase usage [The18a]:

• Function - The paper uses the phrase attack surface with methods, functions or individual lines of code as the lowest unit of reasoning. For example, a paper that discusses the attack surface implications of allowing certain set of functions to be accessible through the Application Program Interface (API).

• File - The paper uses the phrase attack surface as applied to source code files. For example, a paper that discusses the attack surface implications of having source code in certain files vulnerable to particular type of attack.
• **Binary** - The paper uses the phrase attack surface with source code packages such as binaries, packages, modules or components. For example, a paper that presents different approaches to reduce the attack surface of a binary.

• **System** - The paper uses the phrase attack surface when reasoning about entire systems. For example, a paper that presents the security implications of enabling certain features in an operating system such as Windows and Linux.

• **Computer Network** - The paper uses the phrase attack surface as applied to entire networks. For example, a paper evaluating the notion of isolating certain set of sensitive hosts to a sub-network, inaccessible, in general, to hosts outside of the network.

• **Theoretical** - The paper uses the phrase attack surface in a theoretical capacity. Papers that are categorized into the Theoretical level of granularity typically attempt to quantify the attack surface of an entity based on theoretical notions.

We also used Theoretical to categorize papers when we could not infer the granularity from the full text.

In our corpus, 352 of the 644 (or 55%) papers discussed attack surfaces at the Theoretical level, while 149 of the papers discussed the phrase at a Software System level. Only 41 papers discussed attack surfaces at a Binary, File, or Function level, possibly indicating that software development work represented a minority of the papers in our corpus. The high percentage of theoretical uses of the phrase indicates that attack surfaces are still mostly considered a theoretical exercise, with limited number of researchers or practitioners in our corpus applying the concept to a real software system or computer network.

### 2.2.3 Clusters of Definitions

The final stage in our SLR was to analyze, and cluster, various definitions of the phrase attack surface from the relevant papers categorized as Define. For simplicity, we only considered those papers that has at least one citation at the time of our study. Considering only papers that had at least one citation resulted in 19 papers to place into clusters.

As with other manual steps, two authors independently analyzed each definition and assigned it to an appropriate cluster. The clustering of definitions indicates the conceptual similarity among one or more definitions. The authors generated their own clusters as a part of their individual clustering processes. After generating their own clusters and categorizing the definitions into those clusters, the authors involved in the clustering phase met to discuss their respective clustering schemes. The discussion concluded with both authors agreeing on the cluster membership and an unified name for each cluster. The final set of clusters were reviewed by a third author for validation.
2.2.4 Results

Upon analysis, we found six unique clusters of definitions, or definitions that carried the same or similar meaning to one another. The clusters are as follows:

- **Methods**: The attack surface is the methods of implementation, data channels, and data present in the system, with no specific attack features mentioned.

  The Methods cluster is a programming-centric definition that focuses on program flow through a software system. Such an attack surface could be measured by programs like GNU cflow or other static analysis tools for measuring connections through software systems.

- **Adversaries**: The attack surface is the union of all possible ways an attacker could cause damage to a system.

  The Adversaries cluster focuses on attacker behavior only, as only points in the system with active attacks would be considered part of the attack surface. As an example to differentiate the Methods cluster from the Adversaries cluster, there could be entry points from the Methods cluster that have no equivalent in the Adversaries cluster due to a lack of feasible attacks.

- **Flows**: The attack surface is defined as data flow and control flow only, specifically considering external input and permissions when interacting with the system.

  From the above definition, data flow is the order in which data is manipulated in a software system, and control flow is the sequence in which individual commands are executed. The Flows cluster is based on external input. The Flows cluster would be a subset of the Methods cluster, in that some paths available in the Methods cluster may be unavailable to a subset of external actors. The Flows cluster attack surface could change as user authorization level changes.

- **Features**: The attack surface is a enumeration of all available attack avenues to a target system.

  The Features cluster is a higher level set of definitions that focus on the features available in a system at the level of functionality. The Features cluster differs specifically from the previous definitions in that it encapsulates functionality in a system rather than the paths data takes through a system. The Features cluster requires a specific enumeration of parts of the system that are possible attack features, such as a list of open ports, a list of services running by default, et cetera. The Features cluster is distinct from the methods cluster, as specifics are left ambiguous in the methods cluster. As an example, a Windows machine with 50 running services would have a higher attack surface than a Windows machine with 30 running services.
• **Barriers:** The attack surface is the method of preventing attacks, rather than the paths attacks can occur on, by malicious parties.

The *Barriers* cluster focuses on preventative efforts in security on a system, such as firewalls or security policies. Barriers is independent from the other clusters as it focuses on defensive behavior, rather than possible points of attack.

• **Reachable Vulnerabilities:** The attack surface is the vulnerabilities that are exposed to end users via paths or flows, rather than the paths or flows themselves.

Finally, the *Reachable Vulnerabilities* cluster is focused on the exposure of vulnerabilities that attackers can exploit in a software system. Using the Reachable Vulnerabilities cluster, a system with no vulnerabilities does not have an attack surface.

We recommended that practitioners and researchers cite a definition from the cluster that matches their use case for the term. We did not arrive at a singular definition of attack surface. Future work in the area of attack surfaces may focus on providing different terminology for some of these clusters of definitions, so that the term attack surface is not as overloaded as it is today.

### 2.3 Discussion

Taking our own recommendation, we use the *Flows* cluster as the basis for our usage of the term attack surface for the rest of this document. RASA provides a set of code artifacts with empirical evidence of previous crashes, meaning that the code on RASA is in use and accessible to users. RASA is independent of specific types of attacks, prevention methods, or the vulnerabilities themselves, meaning it cannot use the definitions from the Adversaries, Barriers, or Reachable Vulnerabilities clusters. The Methods cluster may be applicable, but RASA specifically considers end user behavior through the use of crash dump stack traces. Attacker behavior may also be considered applicable to RASA as attackers sometimes crash systems they attack, but the relationship is not a direct consequence of RASA.
3 RELATED WORK

In this chapter, we describe work related to the studies presented in this document.

3.1 Crash Dump Stack Traces

Analyzing crash dump stack traces to build profiles of software or identify defective code is an active area of research. Castelluccio et al. [Cas17] developed an approach for grouping crash dumps together for easier analysis. Grouping similar crashes together results in less manual analysis for practitioners interested in crash data. The Castelluccio approach is deployed in Mozilla’s Crash Reporting System.

Bianci et al. [Bia17] use crash data to generate tests designed to reproduce the failures the data represents. Their approach attempts to recreate the conditions of the crash based on the crash data itself so the practitioner attempting to fix the issue can see it happen themselves.

Recent work in the area of crashes has focused on using crash dump data to reproduce the behavior that led to the crash. Cui et al. [Cui16] built Retracer, an approach for reverse engineering the execution path from crash data. Understanding the execution path and the state of the system during the execution path can help practitioners fix issues. Chen et al. [CK15] developed an approach called Star, another approach for reverse engineering crashes using symbolic execution.

Credal, developed by Xu et al. [Xu16] focuses on localizing memory corruption errors from memory dumps. Performing root cause analysis on memory corruption errors has positive conse-
quences for fixing both defects and security vulnerabilities. The volume of recent work in the area of crash dump reproduction speaks to the value of crash data when attempting to triage defects and vulnerabilities.

### 3.2 Vulnerability Prediction and Prioritization

Vulnerability Prediction Models (VPMs) are a subset of Defect Prediction Models (DPMs) e.g. [OY14; Tan15; Kam16] specifically on vulnerabilities in software. VPMs have traditionally performed worse than DPMs in literature, vulnerabilities are orders of magnitude more rare than regular defects [Woo14].

Neuhaus et al. [Neu07] found that vulnerabilities were unlikely to occur again in the same component, but components with the same or similar function calls and important libraries were more likely to have vulnerabilities as well. The Neuhaus result runs somewhat counter to research in defect clustering, which suggests that defects are likely to occur together [Bet08; Men11].

Gegick et al. [Geg10] research the use of text mining in the security field, particularly locating security bug reports. Meneely et al. [MWa; MWb] explored how Linus’s law affected the generation of security vulnerabilities. They explored developer activity metrics, or measures of how many different teams and developers modified a specific piece of code. Meneely et al. made several findings regarding the effect of developer activity metrics: 1) code modified by independent developer groups were more likely to have a vulnerability; and 2) files modified by nine or more developers were 16 times more likely to have a vulnerability than files changed by fewer than nine developers.

The VPM described by Zimmermann et al. [Zim10] is based on static code metrics, along with pre- and post-release vulnerabilities from Microsoft Windows. Metrics were collected from a Microsoft system called CODEMINE [Cze13]. The CODEMINE system provides 29 software metrics, measuring the code churn in the software system, the complexity of the software system, and the organizational features of the team that built the software system. These metrics are treated as features in a Random Forest [Bre01] classifier. They reported a median recall of 0.2 when using all available metrics, with a median precision of approximately 0.45. Rerunning the classifier while only using features from one of the subsets of metrics (code churn, code complexity, and organizational features) resulted in higher and lower precision and recall numbers from run to run, but the reported figures varied widely (from 0 to 0.4 for recall, and from 0 to 1 for precision). Zimmermann et al. recommended that users of the VPM use all available metrics.

Shin et al. [SW08; Shi11] focus on churn and complexity when predicting which parts of a codebase are likely to have vulnerabilities. Shin uses machine learning techniques, specifically linear discriminant analysis, Bayesian network modeling, and Random Forest, to build the prediction models based on the features in Table 8.2. Their approach focuses on reducing the amount of code that needs to be inspected for security concerns. They achieved a reduction of 71% for Mozilla Firefox and 28% for the RedHat Enterprise Linux kernel. Using Discriminant Analysis, Shin et al.
found that a combined model using complexity, code churn, and developer activity metrics had a recall of 0.71 and precision of 0.05. Using a Bayesian Network approach, the same metrics had a recall of 0.72 and precision of 0.04.

Scandariato and Walden et al. [Hov16; Sca14; Wal14] developed an approach utilizing text mining to predict the location of vulnerabilities in code. Text mining for vulnerability prediction involves splitting source code into individual strings, or tokens, and determining if specific token are correlated with vulnerable code. For example, instances of NULL could be indicative of potential null pointer exceptions. The authors have made two replication sites available which informed our construction of a C/C++ version of their approach [Scaa; Scab]. While the Scandariato and Walden approaches featured the best precision and recall scores of any documented VPM to date, questions remain about the oracle used to evaluate their results. Scandariato and Walden used results from HP Fortify\(^1\), a security static analysis tool, to generate their Oracle. Previous work has shown that static analysis tools, particularly for vulnerabilities, has a high false positive rate [Geg08; Geg10], calling the oracle into question.

### 3.3 Classifying Defects

The classification of defects and vulnerabilities has a long history in the research community. Understanding the types of defects and vulnerabilities that are being predicted (or not predicted) by specific prediction approaches informs the use case for each approach. Additionally, understanding the benefits and drawbacks of individual approaches to defect and vulnerability prediction in terms of specific classifications can provide a way to further develop them. Chillarege et al. developed a technique called Orthogonal Defect Classification (ODC) to develop classifications of defects [Chi92].

Podgurski et al. [Pod03] developed an automated approach for classifying defects using supervised and unsupervised pattern classification and multivariate visualization. They group failures with similar profiles to reduce the total number of unique reports from a software system.

Defect research in software engineering already considers the classifications of defects that occur in different types of software systems. Sullivan et al. compared defects that occur in databases versus operating systems [SC92].

Li et al. profiled defect characteristics in open source projects [Li06]. They analyzed over 29,000 bugs in open source software systems from two large open source software projects. They found that the memory-related bugs has decreased as better tooling for these types of bugs have become available, that some simple memory-related bugs are still present, possibly due to carelessness, and that semantic bugs are the primary root cause of the bugs in their dataset. Additionally, they found that security bug rates are increasing, and the majority of security bugs have severe impact ratings.

\(^1\)http://www.ndm.net/sast/hp-fortify
3.4 Classifying Vulnerabilities

In vulnerability classification literature, researchers have mostly focused on whether code can be classified as vulnerable or not [The15c; The17; MM16; Shi11; Zim10; Sca14].

Austin et al. [AW11] performed a comparison of vulnerability detection techniques and found that a single approach was not sufficient for vulnerability coverage. Austin's work informs the need for a vulnerability dataset that not only considers a binary classification of vulnerabilities, but also a classification process for vulnerabilities.

The Common Weakness Enumeration (CWE) \(^2\) maintained by MITRE provides guidance on types of vulnerabilities practitioners may experience. They provide several summaries of the most dangerous and prevalent vulnerabilities, including Top 25 lists of the most dangerous types of vulnerabilities.

The Common Vulnerability Scoring System (CVSS) \(^3\) is a published standard used by many organizations to quantify the impact of vulnerabilities on specific organizations and the community. CVSS is on its third iteration, and the site features a calculator for practitioners to use to generate scores, along with a user guide and examples of the classification and rating process.

The National Vulnerability Database (NVD) maintained by the National Institute of Standards and Technology (NIST) \(^4\) makes use of CWE classifications and CVSS scores as part of their database of publicly disclosed vulnerabilities. CWE classifications are a part of each publicly reported vulnerability in the NVD. Each vulnerability in the NVD is also assigned a CVSS score as part of the disclosure process. NVD also reports which particular version of a software product is affected by the issue, how the issue was discovered, and links to the fix for the vulnerability if available.

Incorporation of the previous three standards and databases into vulnerability research could help practitioners better understand the strengths and weakness of various approaches to vulnerability prioritization and prediction.

3.5 Attack Surface

In this section, we describe both research on attack surface concepts and tools using the attack surface concept.

3.5.1 Attack Surface Research

Manadhata et al. [MW11] provided one of the earliest measures of attack surfaces in software systems, focusing on entry and exit points. Geer uses the attack surface concept to determine relative risk

\(^2\)https://cwe.mitre.org/
\(^3\)https://www.first.org/cvss/
\(^4\)https://nvd.nist.gov/
measures to the software system [GJ11]. The more exposed a part of a software system is, and the
greater the value of the target asset an attacker may be after, the greater the risk factor for the
software.

Zhang et al. [Zha15] used package dependencies to determine the attack surface of package
based systems.

Research efforts into the configuration definition of attack surface include Heumann et al.'s
work on the attack surface indicator (ASI) metric [Heu10]. ASI is an aggregation of several metrics
of web applications that affect the attack surface, such as URL parameters, file upload fields, search
fields, and number of domains. ASI provides a picture of the deployed application's attack surface
rather than the software system itself. For example, if a single web application was deployed on
multiple different servers, various configuration permutations for an application could result in two
completely different ASI values for the software system.

In related domains such as networks and power grids, researchers look for ways to limit the
exposure of critical infrastructure using the attack surface concept [FG15].

Work in the area of attack surfaces sometimes takes the form of minimizing the attack surface to
limit the areas that need to be tested or reviewed for security issues. Geneiatakis et al. [Gen15] focused
on minimizing the attack surface of databases. They minimize the attack surface by randomizing
SQL databases and providing APIs for legitimate users to access data while making attacker access
to the database prohibitively difficult.

Zhang et al. [Zha18] developed an approach for reducing the attack surface of an operating
system kernel. Their approach, called KASR, reduces the attack surface of a system via two steps.
KASR is part of a hypervisor on the system running a particular software package. First, KASR
removes execution permissions from unused, or dead code. Second, KASR places active code in
separate “segments” and actives each segment as required by the execution of the software package.
KASR found particular success running on the Linux kernel, where it prevented six known kernel
rootkits with minor impacts to system performance compared to a standard hypervisor.

While many approaches exist for approximating or reducing the attack surface of software
systems, the generation of a complete attack surface for large software systems remains an open
question.

3.5.2 Attack Surface Tools

Tools like Microsoft’s Attack Surface Analyzer 5 provide practitioners an understanding of the attack
surface of their specific installation of the Windows operating system. Attack Surface Analyzer
informs users what ports are open, what services are running on their machine, the state of permis-
sions on their machine, among other attack surface measurements. Unlike many approaches in

the research literature, Attack Surface Analyzer focuses on the configuration of specific instances of Windows rather than a code-level understanding of the attack surface. Attack Surface Analyzer may find a million different attack surfaces for a million different instances of Windows, depending on the state of the ports, services, and permissions on the systems.

Anvil Ventures described an early prototype of a Python-based approach for measuring changes in the attack surface of Linux systems\(^6\). The command line tool is run once to establish a baseline attack surface of the system, and then can be subsequently run to determine deviations from the baseline that was established previously. The tool, *Daugmon*, can currently detect changes in open ports and services on the system.

While not a tool, OWASP provides a set of guidelines for analyzing the attack surface of a software system\(^7\). The cheat sheet provides additional feedback on measuring attack surfaces in a web application environment, such as web forms and APIs. OWASP asks the user to consider three questions when adding and removing features from a system:

- What has changed?
- What are you doing different?
- What holes could you have opened?

The guidelines provided could provide a framework for an attack surface tool specifically geared towards web applications or APIs.

---

\(^6\)https://anvilventures.com/blog/linux-attack-surface-analysis.html
\(^7\)https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet
Risk-based attack surface approximation (RASA) is an automated approach to identifying parts of the system that are on the attack surface of a software system. RASA identifies an approximated attack surface by analyzing crash dump stack trace data from the target system. We hypothesize that 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 entry points to the software system; and (c) they provide automatically generated control and data flow graphs. We call our approach an approximation because code entities will only be added to the attack surface when a crash has occurred. We include the term risk-based because RASA is an efficient means of finding the part of the software system that is most susceptible to containing vulnerabilities, as it identifies code that has already been shown to have data flow and control issues via the fact that it appears in a crash. We assess our approach by analyzing the percentage of actual reported vulnerabilities in the code and whether they occur in our approximated attack surface.

In this chapter, we motivate the need for RASA, describe why an approximation of the attack surface is desirable, how an approximation of the attack surface is accomplished, why we refer to RASA as a risk-based approach, and outline the steps required to implement RASA.
4.1 Motivation

Vulnerabilities in software are an unfortunate, but inevitable, reality. As a result, software development organizations must take a proactive approach to security. The Trustworthy Computing Security Development Lifecycle (SDL) introduced at Microsoft in 2004 [Lip04] is one such instance of a proactive approach to software security. One of the elements in the design phase of SDL is the measurement of the attack surface of software. At the time, the phrase attack surface was new, having been introduced only a year earlier by Michael Howard [How03].

Howard described the attack surface as a measure of “attackability” of a software system, along three dimensions: targets and enablers, channels and protocols, and access rights. Geer [GJ11] explores the concept of limiting attack vectors using the example of two PDF readers: Adobe Reader and Foxit Reader. Both Adobe Reader and Foxit Reader reduce their attack surface by making document reader functionality available via plugins, which can be disabled by users if they are not currently using that feature.

Some examples of resources that comprise a system's attack surface include the following: open ports, services available on the inside of the firewall, code that processes incoming data, and user interface forms and fields [Man06]. An attacker can use these attack vectors to attack a software system. Practically, some of these attack vectors must be left open for software to function for users. Knowing the attack surface of a piece of software supports decision-making during all phases of software development.
4.2 Why Approximation?

To date, approaches for empirically measuring the attack surface of software systems have relied on manual effort or on alternative definitions of the attack surface. As valuable as the concept of attack surface is, we lack a practical means of identifying source code in the software system that is on the attack surface and, therefore, accessible to end users. Manadhata [MW11] proposed metrics to measure attack surface size and empirically measured entry and exit points by building a call graph, but left identifying the code artifacts between entry and exit points for future work. Other approaches to defining the attack surface have been done at a configuration level without considering code [BM15; How04], and other approaches by Manadhata et al. using API calls have required significant manual work to accomplish [Man06].

RASA, as an empirical metric-driven approach via crash dump stack trace data, is well suited for approximating the attack surface in environments where generation of a full call graph is infeasible. Using an approximation of the attack surface, practitioners can still benefit from a better understanding of code that appears on the attack surface, even if that understanding is incomplete. Building an approximation of the attack surface of a software system provides practitioners with information about where security effort should be focused in a software system. For experienced security teams, an approximation of the attack surface can inform their understanding of their system’s entry and exit points. For example, security testing could not focus on areas that the approximation indicates is not on the attack surface, while also suggesting areas of the codebase that tests should be written for. For inexperienced teams, an initial approximation of the attack surface of their software system could inform where the first set of security tests or review efforts should take place.

The attack surface of a software system can be considered a subset of a call graph of the system, as highlighted in Figure 4.1. As additional binaries, files, and functions are added to a software system, the complexity of creating a complete call graph of the system grows exponentially. Tools for call graph generation like GNU Cflow have difficulty generating call graphs for systems once they reach file sizes in the thousands or tens of thousands. Mozilla Firefox has approximately 50,000 total source code files as of May 2018, and cannot run GNU Cflow without excluding large portions of the software system, fundamentally breaking the call graph. Without a call graph for the target system, building a complete picture of the attack surface is not feasible.

Therefore, RASA’s crash dump stack trace based approach has several advantages for scalability and usability. RASA can be computed in a few seconds after the collection of crash dump stack trace data. RASA is programming language agnostic, and has no issues moving between different languages. GNU Cflow, on the other hand, is limited to C and C++ code. If we want to consider the attack surface of very large systems (e.g. Mozilla Firefox or Microsoft Windows), then we must offer methods that can be practically applied to large software systems during the development

1http://www.gnu.org/software/cflow
process. Existing approaches for attack surface measurement, such as the approaches proposed by Younis and Manadhata [MW11; You14], do not provide code-level insights past entry and exit points of the software system measured. If we want to use knowledge of the attack surface to guide the prioritization of code-level vulnerability location and removal, we must be able to identify code in all parts of the software system, not just entry and exit points.

Because we are approximating the attack surface using empirical metrics, we are not building a full understanding of the attack surface of the software system, or a complete attack surface. We consider RASA to be a subset of the complete attack surface, because the knowledge that the code on RASA has crashed at least once is empirical evidence that the code is part of the complete attack surface. If code was not part of the attack surface of the system, it would not be possible for it to crash after outside input to the system.

Identifying RASA for a software system is, as described earlier, resource efficient compared to identifying the complete attack surface of a software system. Parsing crash data is a string manipulation problem, and computers can parse crash dump stack trace data more efficiently than it can build a call graph of the entire software system. Building an understanding of the call graph of a software system from crash data requires significantly less resources, as further described in Chapter 9.

4.3 Crash Dump Stack Traces

To build our approximation of the attack surface of a software system, we use crash dump stack trace data to build a partial call graph of the target system. A crash occurs when software stops functioning properly and exits. The software may exhibit a variety of different anomalous behaviors, including hanging, freezing, or causing fatal system errors. Crashes can occur for a variety of reasons, including incorrect addressing to the program counter of a machine, a buffer overflow error, an unhandled exception. Some crashes could be exploited by malicious end users to execute arbitrary code, launch privilege escalation attacks, or exfiltrate data from the system.

Data contained in crash dumps may provide useful information to security practitioners looking to prevent attacks. For example, attackers sometimes use crashes to find possible weaknesses in the target system. If an attacker can consistently cause a program to crash, they can find important forensic information about the system properties at the time of the crash, such as what code was running and what the state of the registers was at the time of the crash. If the crash is readily repeatable, the crash could be considered a Denial of Service (DoS) attack in itself.

To identify artifacts that may be reached and thus exploited from users, we use stack traces from crash data, or crash dump stack traces, to identify code artifacts involved with the crash. Stack traces identify code that was loaded in memory at the time of the crash. Typically used for debugging, traces define the logical path between an external input and the system crash. Stack traces contained
in crash dumps identify code artifacts that were executed prior to the crash taking place. The order of lines, or frames of code artifact in the crash represent the sequence of code artifact calls. The initial entry point is typically the top line item on a stack trace, and should in most cases be the first artifact touched by external input. Consecutive lines in a stack trace identify code artifacts that called each other. If a function in foo.cpp calls a function in bar.cpp, we say that there is a direct neighbor relationship from foo.cpp to bar.cpp. The direct neighbor relationship is indicated in a crash dump stack trace by foo.cpp occurring directly before bar.cpp.

Through the parsing of the stack traces generated from a crash, we can build a listing of the artifacts that external input can access. Our hypothesis is that the subset of code appearing in crash dump stack traces is more likely to have exploitable security vulnerabilities than code that does not appear in crash dump stack traces. Malicious users can only exploit vulnerabilities that they can access. If a bad check on incoming data could result in a buffer overflow attack in a specific function, but no outside user can ever pass input to that bad check, then that vulnerability is not exposed to end users unless code or configuration changes cause it to be exposed.

4.4 Risk Based

Identifying the “riskiest” code in a software system is beneficial from a prioritization perspective, because testing and review efforts can be focused on those parts of the system. The concept of risk is common outside of software engineering as well, and is a good avenue to guide discussion regarding how security vulnerabilities effect a product overall. Linacre et al. [Lin05] produced the Venn diagram in Figure 4.2 illustrating the relationship between a vulnerability, the consequences of the vulnerability, and the threat posed by the actors on the vulnerability. This model of vulnerability can be applied not only to security vulnerabilities, but vulnerable parts of any system.

If a high percentage of discovered vulnerabilities are on RASA and the computation resources requirement of RASA is much lower than that of the complete attack surface, we can consider RASA a 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 exploitable vulnerabilities, despite being a subset of the complete attack surface of the system. Code that has historical crashes can also be considered riskier than code that has not crashed, because we have empirical evidence that the code artifacts on the stack traces in those crashes have errors, otherwise the crash would not have occurred. We also have empirical evidence that the code appearing on crashes is accessible to outside input, otherwise it would not have appeared on a crashing frame. This combination of density of vulnerabilities, probability of errors, and accessibility creates an environment where vulnerabilities are not only more likely to be, but are likely to have real impact on the operation of the software system.
Figure 4.2 A Venn diagram representing the relationship between a vulnerability, the consequence of the vulnerability, and the threat posed by the vulnerability. Each of these factors add up to the overall risk of the vulnerability being exploited.

4.5 Process

In the section, we provide a set of steps, indicated by the subsections, for implementing RASA for a target software system. Each subsection contains implementation details for that step.

4.5.1 Collect Crash Data

The first step of our process is to collect crash data for the target system. Crash data collected for the software system needs to contain several key pieces of information for RASA to be implemented:

- **System**: The name of the software package that crashed.

- **Version**: The specific version of software that was running at the time of the crash.

- **Date**: The date the crash occurred. This is to verify that the crash took place during the time period your particular study is taking place, for mapping to source code and vulnerability data. Crashes that occur in a different time period may refer to code that no longer exists.

- **Stack Trace (Crashing Thread)**: A stack trace in the crash dump, specifically of the thread that crashed during execution.

System, version, and date information are used to verify that the crash collected is relevant for the system you are trying to analyze. The system and version information should match the system
Figure 4.3 Example of an anonymous stack trace. From the stack trace, we can identify binary, function, and sometimes source file information. Each of these stack traces can be transformed into a graphical representation. Each stack trace contributes one particular path (see middle) to the overall attack surface graph (see right hand side).

and version you are trying to generate RASA for, and the date should be checked as appropriate, according to the source code and vulnerability data the implementer has access to.

Once the system, version, and date information has been verified and the crash has been declared relevant, we move on to parsing the stack trace for relevant information.

4.5.2 Parse Stack Trace

Stack traces vary greatly in terms of format and presentation, but there are several consistent properties between stack traces from different systems. In particular, a stack trace has code artifacts in a sequential order, which is the order the code artifacts were called in leading up to the crash. These individual lines are referred to as frames. The order the frames appear in can be sequential or reversed, depending on the system designer’s preference. An example can be seen in Figure 4.3.

Each frame of the crashing thread from the crash dump stack trace is separated into individual artifacts. Stack trace frames can contain a variety of information, including names of code artifacts, offsets into particular functions or binaries, and the state of registers in the system at the time the frame was executed. Including binary name, function name, and file name. The order of the frames
Table 4.1 An example of a database or file format for summarizing crash dump stack trace data.

<table>
<thead>
<tr>
<th>Binary Name</th>
<th>File Name</th>
<th>Function Name</th>
<th>Frequency</th>
<th>Fan-In</th>
<th>Fan-Out</th>
<th>Boundary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foo</td>
<td>temp.cpp</td>
<td>MyFunction</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>Foo</td>
<td>file.cpp</td>
<td>YourFunction</td>
<td>13</td>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>Foo</td>
<td>test.cpp</td>
<td>TheFunction</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Foo</td>
<td>test.cpp</td>
<td>AFunction</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Bar</td>
<td>file.cpp</td>
<td>YourFunction</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Bar</td>
<td>main.cpp</td>
<td>Execute</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bar</td>
<td>opening.cpp</td>
<td>XMLParser</td>
<td>16</td>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Main</td>
<td>closing.cpp</td>
<td>GetInfo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Main</td>
<td>closing.cpp</td>
<td>SetInfo</td>
<td>43</td>
<td>1</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>Main</td>
<td>closing.cpp</td>
<td>PostInfo</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

in a stack trace is preserved when storing the stack trace data to a .csv file or database, and for each frame, we record the following information as a unique identifier of a particular code artifact:

- **Binary Name**: The name of the binary, if applicable, that was seen on a frame in the crash dump stack trace.

- **File Name**: The name of the source code file, if applicable, that was seen on a frame in the crash dump stack trace.

- **Function Name**: The name of the function or method, if applicable, that was seen on a frame in the crash dump stack trace.

We use the combination of binary name, file name, and function name as a key in our dataset. The same binary can be contained in multiple combinations, with different source code files and/or functions associated with the binary. Similarly, the same file can be contained in multiple combinations, with different functions associated with the file. An example of these relationships can be seen in Table 4.1.

For each unique combination, we collect a variety of metrics indicating properties of that combination in our set of crash dump stack traces. The metrics collected are the following:

- **Frequency**: The number of unique crash dump stack traces a combination (set of binary, file, function names) has been seen in the set of crashes.

- **Fan-In**: The number of unique calls to a combination from other code that have been seen in the set of crashes. Indicated by the number of unique combinations that appear directly before the combination in the set of crash dump stack traces.
• **Fan-Out:** The number of unique calls to other parts of the codebase from the current combination that have been seen in the set of crashes. Indicated by the number of unique combinations that appear directly after the combination in the set of crash dump stack traces.

• **Boundary:** The number of times a combination appears as the first or last frame in a crash dump stack trace. A non-zero value indicates that the combination is an entry or exit point for the system.

These metrics are used in the studies described in the following chapters.

### 4.5.3 Map Stack Trace Data To Source Code

The third step in our process is mapping the collected binary, file, and function name information to code artifacts in the software system’s source control. In some cases, corruption in the system or formatting issues with the stack trace results in unclear string matching between the data collected in the stack trace and the source control repository. There are two primary strategies we use in the following studies to make these mappings:

1. Build a complete list of source code artifacts, and verify each combination of binary, file, and function name information exists in source control, or

2. Check each unique combination of binary, file, and function names against source control to verify their existence.

In the case that an automatic string mapping match is unable to be made between stack trace and source control data, manual effort specific to the software system in question is likely required to resolve differences between the two datasets. The mapping between both datasets is important, as it validates that the stack trace data is correct and that vulnerability data collected in later steps can be mapped to stack trace data. The specific strategy used for mapping stack trace data to source control data is up to the implementer, and the best strategy may change depending on the size of the stack trace data and the amount of data in source control. For example, it may be infeasible to build a complete list of source code artifacts in from source control for a system like Microsoft Windows, but feasible for a system like Mozilla Firefox.

#### 4.5.3.1 Handling Edge Cases

String matching between two potentially inconsistent datasets can result in frequent false negatives. For example, source code file names may not be part of the data reported in a crash dump stack trace for a target system, but binary and function names may be reported. In that case, we would use of source control data indicating relationships between binaries, files, and functions to find the
missing file name, if possible. In the case that a binary, file, and/or function name is unable to be mapped or verified, that field is marked as “unknown” in the dataset. We keep “partial” mappings, such as the case where binary and function name information is known and file name information is not. That design decision results in possible missed files on the approximated attack surface for the system.

### 4.5.4 Collect Vulnerability Data

For evaluating the effectiveness of RASA generated for a software system, we need to map specific binaries, files, and functions to specific vulnerabilities. Collection of vulnerability data can be from a variety of sources:

- A public database such as the National Vulnerability Database (NVD), or
- Postings of publicly disclosed vulnerabilities on a organization blog, or
- Disclosures of vulnerabilities on a listserv, or
- An organization’s bug tracker, such as a Bugzilla database.

We map vulnerabilities to specific binaries, files, and functions by the code artifacts that was changed as part of fixing the vulnerability. At least one of the binary name, file name, or function name is required to be included in the vulnerability report in order for the vulnerability to be included in the dataset collected for RASA. Name information is sometimes disclosed directly in the vulnerability report. Name information can also come from a diff attached to the vulnerability report; a diff is a description of changes in a source code file. If at least one of the binary, file, or function names is not present in the bug report, than the vulnerability is discarded.

Assuming at least one code artifact name is present somewhere in the vulnerability report, the following information about the vulnerability is collected:

- **Binary Name:** The name of the binary that the vulnerability fix occurred in.
- **File Name:** The name of the source code file that the vulnerability fix occurred in.
- **Function Name:** The name of the function or method that the vulnerability fix occurred in.
- **Classification:** The classification or type of vulnerability, if available. Classifications could include buffer overflow attacks, cross site scripting attacks, et cetera.
- **Severity:** The severity of the vulnerability, if available. Severity could be determined by the software team itself in their bug repository, or could be a public measurement via CVSS or CWE.
We map binary name, file name, and/or function name in our vulnerability dataset to the binary name, file name, and function name in our crash dump stack trace dataset. In the case that a vulnerability is a partial match with respect to “unknown” classifications (ex. binary and function name are a match, file name information from the stack trace is unknown), we consider it a correct match. Using the vulnerability dataset as an oracle, we can evaluate the effectiveness of RASA for a set of vulnerabilities, including the percentage of vulnerabilities covered, the classifications of vulnerabilities covered, and the severities of vulnerabilities covered. The four metrics from crash dump stack traces can also be used as inputs into VPMs to predict specific keys that are likely to contain vulnerable code.

The output of RASA is a classification of whether a code artifact is on or off the approximated attack surface. The RASA classification can be used for prioritizing defect fixing and validation and verification efforts. The on or off classification using the RASA approach is modified as necessary to explore extensions of RASA in the following chapters.
We address the following research question with this chapter:

**Approximation (RQ1):** Are vulnerabilities more likely to occur in code appearing on crash dump stack traces than code not appearing on crash dump stack traces?

### 5.1 Introduction

RASA is an automated approach to identifying parts of the system that are contained on the attack surface through crash dump stack trace analysis. As part of the RASA process, we parse crash dump stack traces from a target system, adding all code found in these traces onto RASA. Code that appears in stack traces caused by user activity is on the attack surface by definition, because it appears in a code path reached by outside input.

In this chapter, we present two studies on the effectiveness of the RASA approach. First, we build an attack surface approximation for the Windows operating system based on stack traces. To assess our approximation, we compare the set of known security vulnerabilities from Windows 8 against this attack surface.

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. The initial study on RASA was performed on Microsoft Windows 8 [The15c] and was done with millions of crashes. Not all organizations have as much crash information as organizations the size of Microsoft, so the feasibility of RASA on smaller datasets should be explored.

To explore the idea of running RASA on smaller datasets, 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. 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.

5.2 Methodology

In this section, we describe the methodology used in our studies.

5.2.1 Risk-Based Attack Surface Approximation

We first select a collection of stack traces from crash dumps from the software system we are analyzing. For each individual stack trace pulled from a crash dump, we record the binary, file, or function (hereafter referred to as a code artifact) found on each line of each stack trace, and increment a counter for each code artifact indicating the number of times that the code artifact has appeared in a crash. 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 we identify the binary the specific instance of foo.cpp is included in. We accomplish source code file lookup by looking at the binary and function found on the same line of the crash dump stack trace as the foo.cpp. Based on the associated binary and function, we can identify where in the directory structure of the software the file is located. A list of associations between binaries, files, and functions in a software system would come from toolsets provided by the company maintaining the system, or pulled directly from source control in the case of open source projects.

We parse each individual stack trace in our dataset in sequence, and extract the individual code artifacts that appear on each line. We then output the frequency in which each unique code artifact appears in a stack trace from the parsed set. 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. In addition to the list of files created by RASA, we generate a list of code that has had at least one historical vulnerability fix associated with it. The list of historical
vulnerability fixes comes from the organizations bug tracking repository. We check the binaries, files, and functions included in diffs (representations of code changes) attached to bug reports that are flagged as security vulnerabilities. All code artifacts included in these diffs are considered artifacts with historical vulnerability fixes. 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. First, we calculate **Code Coverage (CC)**, or the percentage of code in the target software system that appears in at least one stack trace. We calculate the CC metric via the following formula [The17]:

\[
CC = \frac{\# \text{artifacts in crashes}}{\# \text{artifacts in the system}}
\]  

(5.1)

Next, we calculate **Vulnerability Coverage (VC)**, or the percentage of files with security vulnerabilities that appear in at least one stack trace. We calculate the VC metric via the following formula [The17]:

\[
VC = \frac{\# \text{artifacts with at least one vuln. in crashes}}{\# \text{artifacts with at least one vuln.}}
\]  

(5.2)

For each type of code artifact, different values of CC and VC will be calculated. Binaries and source code files, for example, would be calculated separately from one another.

### 5.2.2 Random Sampling

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 may choose to permanently store 20% of all crashes reported by users. To simulate the partial storage 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 RASA's predictive power.

5.3 Windows Case Study

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

5.3.1 Data Sources

The stack traces used in our Windows study are from the Windows 8.1 operating system. Stack traces for the Windows study come from three complementary sources: fuzzing crashes, user mode crashes, and kernel mode crashes. We keep track of the source of each crash as we parse each stack trace. We now describe each crash source.

Microsoft's security testing teams generate fuzzing crashes. Fuzzing is a testing strategy revolving around sending random or deliberately malformed/malicious data to the input points on a system. The goal of fuzzing is to simulate attacks and to get the system to behave in an unexpected manner. Typically, any response from the system that differs from a standard error message is flagged and investigated. Possible results from fuzz testing include crashes, memory leaks, and security bugs such as data loss or improper access. Fuzzing crashes are a useful data source as any input generated and fed into the system under test could also originate from a user and thus from a potential hacker. Fuzzing aims to discover security vulnerabilities, and any irregular fuzzing results are relevant for determining what code areas were involved in handling the malformed input.

Windows users generate user mode crashes. User mode crashes are field/customer crashes that are not due to hardware failures. Crashes in applications running without administrator rights triggers collection of crash information that may be sent to Microsoft. The system responsible for collecting the data, Microsoft Error Reporting, is executed under the same user privileges as the user that ran the crashed application. However, running in user mode limits the stack trace generator's ability to access and resolve which resources were involved in the crash. Thus, user mode crashes may only identify code areas that are accessible by the user in that mode.

Kernel crashes occur on field and customer machines, but unlike user crashes, they occurred within the Windows kernel. In general, kernel crashes indicate more severe failures and usually include full resolutions of code artifacts, as long as they are part of the Windows system. The kernel is running under administrative privileges allowing the stack trace generator to gather more details about the system's granular details.

To measure the effectiveness of our approach, we required a set of vulnerabilities. We use the set of vulnerabilities seen in Windows 8 both pre- and post-release of the product. We then check to see if these vulnerabilities appear on our attack surface approximation to determine whether they
would be caught or missed by our approach.

5.3.2 Data Collection

Each line of a stack trace for Windows 8.1 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 the 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, RASA 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 the names of code artifacts on the attack surface approximation. When code artifact names are added to the database, we enter as much information as possible about the line in the stack trace. In some cases, the data is just the binary name, as the file name and function name 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. Mapping the relationship between code artifacts can be used in a variety of helpful ways, particularly in visualizing these relationships in graph format.

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. The 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.

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 our study.

5.3.3 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, we use the last file seen for a particular crash, while for Windows we use 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.

5.4 Firefox Case Study

In this section, we discuss our 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.

5.4.1 Data Sources

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 crash dump stack trace database, Mozilla Crash Reports ¹, as it only keeps full stack traces from crashes for approximately 6-7 months. Instead, we made use of the historical crash dumps from a (now defunct) storage site.

¹https://crash-stats.mozilla.com/home/products/Firefox
We performed our analysis on crashes occurring from May 2010 to March 2012 due to available security data. We collected approximately 10 million crashes for our study. Vulnerability data for our study was pulled from the Mozilla Security Advisories blog [Mozb]. We use historical vulnerability data posted on the log from the same time period as the crash data: May 2010 to March 2012. We collected 147 vulnerabilities for our study.

### 5.4.2 Data Collection

For Firefox's crash analysis dataset, crashes had a consistent format, allowing us to build a string parser to grab the file path and filename from the middle of the string, as delimited by colons. An example of the format of the files in Firefox crashes is found in Figure 5.1. 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.

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 the May 2010 to March 2012 time period was the availability of these security reports. Mozilla makes vulnerability details available after they are confident the issue has been resolved for their users.

### 5.4.3 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.

5.5 Results

In this section, we describe the results of our generation of RASA for Windows and Firefox at the binary and file level, along with an exploration of the effect of random sampling on the resulting approximation.

5.5.1 Risk-Based Attack Surface Approximation - Binaries

Table 5.2 contains a summary of our Windows 8 attack surface approximation from our data sources. We split our data sources into three separate columns, indicating user mode stack traces, kernel mode stack traces, and fuzz testing stack traces. The first row indicates the total number of binaries seen in stack traces from that data source. The second row is the percentage of all shipped binaries that is included in row 1. For our purposes, shipped binaries are any binary that is included in a commercial release of the Windows product. For our example, the 32 binaries seen in fuzzing stack traces account for 0.9% of all shipped binaries. There can be overlap between multiple sources of stack traces. The %vulnerabilities row contains the percentage of all known vulnerabilities seen in that specific subset of binaries. Similar to the shipped binaries line, there can also be overlap on the vulnerabilities row, which shows why the percentages add up to over 100%. The table shows that 48.4% of binaries contain 94.6% of the known vulnerabilities within Windows 8 when considering kernel and user mode crashes. Fuzzing crashes were left out of our combined column because it offered no improvement to our approximation. User and kernel mode covered all known vulnerabilities in the fuzzing dataset.

5.5.2 Risk-Based Attack Surface Approximation - Files

Next, we move to file level granularity for Windows 8.1 and Mozilla Firefox. Tables 5.3 and 5.4 contains the results of running RASA against these two systems. After applying RASA to Windows 8.1, 11.6% of files contained 20.2% of vulnerabilities in our study. The binaries study on Windows 8 found that 48.4% of binaries contained 94.8% of historical vulnerabilities when filtering by binaries
Table 5.2 Descriptive statistics for attack surface at binary level, broken down by type of stack trace category that identified the binary.

<table>
<thead>
<tr>
<th></th>
<th>User Mode</th>
<th>Kernel Mode</th>
<th>Fuzzing</th>
<th>Kernel and User Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Binaries</td>
<td>40.2%</td>
<td>7.1%</td>
<td>0.9%</td>
<td>48.4%</td>
</tr>
<tr>
<td>%Vulnerabilities</td>
<td>66.7%</td>
<td>40.6%</td>
<td>14.9%</td>
<td>94.6%</td>
</tr>
</tbody>
</table>

with a minimum of one appearance on a stack trace. When applied to Mozilla Firefox, 15.8% of files contained 73.6% of the vulnerabilities in our study [The17].

Moving to the file level improved the granularity of attack surface approximation compared to the previous study [The15c], 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.

5.5.3 Random Sampling

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 5.3 [The17] for Windows 8.1. As the size of the random sampling increases, we see that the 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.

A graph of total file coverage (the number of files on RASA versus the total number of files in the system) for Windows is found in Figure 5.3. From the graph, we see an increase in the number of files included on the RASA as the sample size increases. A graph of vulnerability coverage for our implementation of RASA for Windows is found in Figure 5.4. We see a roughly linear relationship between the increase of the CC and VC metrics.

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 5.4 [The17] for Mozilla Firefox. As the size of the random sampling increases, we see that the average number of files covered by RASA also increases, while the standard deviation of the individual runs shows no discernable trend.
Table 5.3 Results of randomly sampling crash dump stack traces while implementing Risk-Based Attack Surface Approximation on Microsoft Windows 8.1.

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>Avg %files</th>
<th>Avg %vulns</th>
<th>Stdev %files</th>
<th>Stdev %vulns</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>13.8%</td>
<td>32.0%</td>
<td>0.03%</td>
<td>0.1%</td>
</tr>
<tr>
<td>20%</td>
<td>16.6%</td>
<td>35.9%</td>
<td>&lt;0.01%</td>
<td>0.1%</td>
</tr>
<tr>
<td>30%</td>
<td>18.3%</td>
<td>38.1%</td>
<td>&lt;0.01%</td>
<td>0.08%</td>
</tr>
<tr>
<td>40%</td>
<td>19.6%</td>
<td>39.5%</td>
<td>&lt;0.01%</td>
<td>0.1%</td>
</tr>
<tr>
<td>50%</td>
<td>20.7%</td>
<td>40.8%</td>
<td>&lt;0.01%</td>
<td>0.1%</td>
</tr>
<tr>
<td>60%</td>
<td>21.4%</td>
<td>41.9%</td>
<td>&lt;0.01%</td>
<td>0.1%</td>
</tr>
<tr>
<td>70%</td>
<td>22.1%</td>
<td>42.5%</td>
<td>&lt;0.01%</td>
<td>0.1%</td>
</tr>
<tr>
<td>80%</td>
<td>22.7%</td>
<td>43.3%</td>
<td>&lt;0.01%</td>
<td>0.07%</td>
</tr>
<tr>
<td>90%</td>
<td>23.2%</td>
<td>44.2%</td>
<td>&lt;0.01%</td>
<td>0.05%</td>
</tr>
<tr>
<td>100%</td>
<td>23.6%</td>
<td>44.5%</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

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.

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 5.1. From the graph, we see an increase in the number of files included on the RASA as the sample size increases. A graph of vulnerability coverage for our implementation of RASA for Mozilla Firefox is found in Figure 5.2. In the 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 the graph that random sampling of crashes has a minimal effect on RASA's ability to cover security vulnerabilities.

From Figures 5.1 and 5.3, 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. 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 at a sampling rate of 70% or higher. 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.

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
Table 5.4 Results of randomly sampling crash dump stack traces while implementing Risk-Based Attack Surface Approximation on Mozilla Firefox.

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

of the system the task is running on, the available memory in the system, and the code for parsing the stack traces.

5.6 Discussion

We have evaluated the effectiveness of RASA for Microsoft Windows and Mozilla Firefox, with indications that code artifacts appearing on crash dump stack traces is more likely to have security vulnerabilities than code artifacts that do not appear on crash dump stack traces. 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 our 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.

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. In subsequent chapters, we examine the relationship between files (or other code entities at various levels of granularity) and determine which relationships appear in crashes most frequently. These handoffs may point us towards where vulnerable code lives.

We found that our fuzzing set of stack traces overlapped with the user and kernel datasets, finding only a few additional vulnerabilities. While there remained some overlap between user and
kernel mode stack traces, together these sources are able to account for the majority of known vulnerabilities. Our results indicate that the attack surface approximation may be computed using only user and kernel crashes. For practitioners, the fact that fuzzing crashes are not required means that crash data they have already collected that has symbol data can be used to build a historical model of their software's attack surface.

One of the difficulties with parsing stack traces is the fact that the problem reduces to string parsing to pull out code entities. While string parsing in itself is not a difficult task, stack traces from different parts of a system may be in different formats, and some stack traces contain different information. Our random sampling results suggest that RASA could be used on much smaller software systems, if the crash dump stack traces are of similar quality.

5.6.1 Practitioner Reactions

Within Microsoft, security teams agreed that the idea that attack surface approximation based upon stack traces could be useful in day-to-day operations. Security professionals were quoted:

“Attack surface approximation would be useful to help target our review efforts.”

One of the benefits of RASA is the high degree of automation possible for RASA. The only software-specific element is the parsing of the individual stack traces. RASA is a portable approach and could be applied to any other software project in development that collects crash dump stack trace data from end users, though the accuracy and reliability of these results on different systems may change.

Figure 5.1 Graph of the percentage of files included on the RASA at random samples for Mozilla Firefox.

![Graph of the percentage of files included on the RASA at random samples for Mozilla Firefox.](image)
Figure 5.2 Graph of the percentage of vulnerabilities covered by RASA at random samples for Mozilla Firefox.

Figure 5.3 Graph of the percentage of files included on the RASA at random samples for Microsoft Windows.
From these results, RASA provides an approximation of the attack surface of a system with some accuracy.

5.6.2 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?

In our results, only 6 out of 271 files in Mozilla Firefox associated with a security vulnerability fix appeared in only one stack trace. The difference in total vulnerability coverage from a 10% sample to the complete set of crashes is 11 files. A list of all files in our study associated with a security vulnerability that crashed 6 or less times is in Table 5.5.

Our 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.
Table 5.5 Snapshot of files with associated vulnerabilities where the number of times the file was found in a crash was six or less, sorted by total number of crashes.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Crashes</th>
<th>Vulnerability?</th>
</tr>
</thead>
<tbody>
<tr>
<td>js/src/liveconnect/jsj_JavaClass.c</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>js/src/liveconnect/jsj_JavaArray.c</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>docshell/base/nsDocShellEnumerator.h</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>js/src/jsdbgapi.h</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>modules/libpr0n/decoders/nsIconDecoder.cpp</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>layout/generic/nsPageContentView.h</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>layout/xul/base/src/tree/src/nsTreeContentView.h</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>content/base/src/nsNodeIterator.h</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>modules/oji/src/nsCSecurityContext.h</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>layout/xul/base/src/tree/src/nsTreeSelection</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>js/src/jslock.h</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>security/nss/cmd/strscint/strscint.c</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

While our 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.

5.7 Limitations

We have explored two software systems as part of our study. RASA may not generalize to other systems without similar studies in different domains. In the absence of an oracle for the complete attack surface, we cannot assess the completeness of our approximation. Our determination of accuracy is based on known vulnerabilities, which may introduce a bias towards code previously seen to be vulnerable. While our oracle of historical vulnerabilities may work under the assumption the bias is minimal, further exploration is needed. The set of artifacts set as part of the attack surface is an approximation, and we do not claim to capture all possible vulnerable code artifacts.

Each level of code abstraction (binaries, files, and functions) has different strengths and shortcomings. For example, while binary level allows us to make more accurate claims about where vulnerabilities might be, a single binary could contain hundreds of files that need to be looked at. The large size of binaries limits the practicality of the binary-based approach. File level analysis provides more actionable results, in the Mozilla Firefox study, did not result in a significant loss of coverage of vulnerabilities. However, for Windows, the loss of coverage of vulnerabilities was more significant. Understanding the tradeoffs of each level of granularity is important for practitioners,
as different practitioners may have different requirements for effective use.

Finally, due to code or configuration changes, code that is not on the attack surface may be moved on to the attack surface. However, prioritization of code on the attack surface, using our method or other attack surface identification methods, can be used to reduce security risk.
We address the following research questions with this chapter:

**Change (RQ2):** Does the approximated attack surface need to be re-computed with each new release or periodically during development?

**Complexity (RQ3):** How are security vulnerabilities correlated with code complexity metrics such as fan-in and fan-out, generated from crash dump stack traces?

**Boundary (RQ4):** Is reviewing data flows in a software system necessary to find a majority of security vulnerabilities, or can entry and exit points be prioritized for security review?

### 6.1 Introduction

RASA can be used for the prioritization of security efforts by practitioners. In this chapter, we generate three different metrics to further prioritize efforts for the code on RASA. We developed change metrics to examine if the approximated attack surface is dynamic over time, necessitating periodic re-computation. We utilize the complexity metrics fan-in and fan-out, originally defined by Henry et al. [HK81] to analyze the relationship between complexity and vulnerability density. Finally, previous research on the use of attack surface in a real system focused on identifying only the entry and exit points of software systems [Man06]. Crash dump stack traces can be used to identify the entire attack surface path through a software system, but the concept of a boundary of the system could be useful for prioritization efforts within the attack surface.
6.2 RASA Metrics

In this section, we describe three metrics derived from RASA: change, complexity, and boundary metrics.

6.2.1 Change in Attack Surface Over Time

Knowing whether to periodically re-compute attack surface of a system, and how often you should re-compute, will help prevent security professionals from operating on outdated assumptions.

To that end, the change metric represents how much the set of artifacts found in crash dump stack traces has changed between two approximations. Changes in RASA indicated by changes in crash dump stack traces can indicate changes to code flow in the system. Changes to code flow may expose legacy code that was previously not on the attack surface. Code that is newly appearing on the attack surface will typically have not been reviewed as stringently by the security team, if it was reviewed by the security team at all.

The change metrics track the amount of code covered by RASA in different versions of a software system. We use $V$ to indicate the first version in our comparison, and $V'$ to indicate the next version in our comparison. Our three change metrics are defined as follows: Dropped Code (DC) is defined as the code artifacts that only appear in the earlier RASA (older version, previous time period), and is expressed in Equation 6.1 as the relative compliment:

$$DC = \frac{V}{V'}$$  \hspace{1cm} (6.1)

New Code (NC) is defined as the code artifacts that only appear in the later RASA (newer version, next time period), and is expressed in Equation 6.2 as the relative compliment:

$$DC = \frac{V'}{V}$$  \hspace{1cm} (6.2)

Persistent Code (PC) is defined as the code artifacts that appear in both RASAs in the comparison, and is expressed in Equation 6.3 as the intersection:

$$DC = V \cap V'$$  \hspace{1cm} (6.3)

These metrics are generated using data from the appropriate time and version of the software system. For example, if you wanted to compare the attack surface of a system in Windows 8 versus Windows 8.1, you would collect crash dump stack traces from those two versions, create RASA for both, and compare the results using the above metrics.
6.2.2 Complexity Metrics

Stack traces preserve the order in which that code was accessed leading up to the crash. The order in which code artifacts appear in the stack trace can indicate the complexity of individual code artifacts that appear in failure cases of the software. We used the concept of complexity metrics, particularly fan-in and fan-out defined by Henry et al. [HK81]. By aggregating all of the crash dump stack traces from our dataset, we can determine fan-in and fan-out for code artifacts based on crash dump stack traces. As an example, if binaries X, Y, and Z appear directly after binary A in stack traces, binary A has a fan-out value of 3. Additionally, these measures of complexity could be used to provide an estimation of how interconnected vulnerable code is. Code artifacts with higher fan-in and fan-out values as measured by crash dump stack traces may need to be prioritized for vulnerability fixes, as the greater span of possible flows through that code artifacts could indicate vulnerabilities with more severe consequences for the software system.

For each code artifact in a crash dump stack trace, we record the fan-in, the fan-out, and indicate where one or more vulnerabilities were found on the artifact. We define two metrics:

**Crash Fan-In (CFI)** is defined as the number of unique code artifacts that appear directly before the artifact in the crash dump stack traces.

**Crash Fan-Out (CFO)** is defined as the number of unique code artifacts that appear directly after the artifact in the crash dump stack traces.

Additionally, we aggregate the CFI and CFO measurements into the following two metrics:

**Vulnerability Percentage** \( VPI(X) \) is defined as the percentage of the total number of vulnerabilities that appear for a specific value of CFI. For example, a VFI(5) value of 20% would indicate that 20% of vulnerabilities occurred on artifacts with a value of 5 for CFI.

**Vulnerability Percentage** \( VPO(X) \) is defined as the percentage of the total number of vulnerabilities that appear for a specific value of CFO. For example, a VFO(3) value of 15% would indicate that 15% of vulnerabilities occurred on artifacts with a value of 3 for CFO.

These metrics can be used as inputs to generate vulnerability correlations or visualizations to help security professionals prioritize efforts based on complexity.

6.2.3 Boundary Metrics

Boundary metrics determine which artifacts from the target system appear first after outside artifacts, and which outside artifacts are seen directly before code artifacts in the target system. The boundary could be useful for determining which code entities to focus hardening and testing efforts on, while the outside list may indicate which third parties are causing the most issues for end users.

Therefore, we measure the amount of vulnerabilities that appear on the boundary of a target system by analyzing crash dump stack traces. We identify the boundary from the stack traces by flagging the first target binary on the stack trace versus the external binary that calls the target
system. By aggregating the crash dump stack traces from our dataset, we can determine where the boundary of the system is. To identify the boundary, we first tag all code artifacts from the target system as "System Related". Next, we parse the stack trace sequentially, looking for the first "System Related" code artifact. Once the first system related artifact is identified, that artifact is flagged as on the boundary of the system. Finally, we calculate our boundary metrics: Boundary Code (BC) and Boundary Vulnerabilities (BV). BC is calculated by Equation 6.4:

\[
BC = \frac{\text{artifacts on the boundary}}{\text{artifacts}}
\]  

(6.4)

BV is calculated by Equation 6.5

\[
BV = \frac{\text{artifacts on the boundary with at least one vuln.}}{\text{artifacts with at least one vuln.}}
\]  

(6.5)

From these metrics, software developers can make data informed decisions on whether to focus maintenance efforts on the boundary of their system, where their code interacts directly with outside developers, or on other sections of the codebase.
Table 6.1 - Number of crash dump stack traces parsed in 2014 and 2015, by OS version.

<table>
<thead>
<tr>
<th></th>
<th>Windows 8</th>
<th>Windows 8.1</th>
<th>Windows 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>5 million</td>
<td>3.5 million</td>
<td>1 million</td>
</tr>
<tr>
<td>2015</td>
<td>4 million</td>
<td>5.5 million</td>
<td>5.5 million</td>
</tr>
</tbody>
</table>

6.3 Data Sources

To improve on previous work [The15c], larger datasets were required for sufficient samples to work from. The dataset consists of approximately 24.5 million stack traces from 2014 and 2015 from Windows 8, 8.1, and 10, illustrated by Table 6.1. These crashes represent approximately 500 million records in our database, with a single line in a crash dump represented by a single record. On average, each crash dump has 20 lines or records.

Stack traces in Windows typically contain binary/function information, and file information is filled in based on a mapping process. In most cases, a single binary/function pair will map to a single file. In the case of multiple mappings, we take the first match.

6.4 Data Collection Methodology

Each stack trace from a crash dump is parsed and separated into individual artifacts, including binary name, function name, and file name. We then map each of these artifacts to code in the system under study.

Formalizing a database schema for the storage of stack traces from the target software system is the next step for an experimental setup, as proper indexes on these tables can then be created for faster queries and faster experimentation. The full database diagram for our collection of crash dump stack traces can be seen in Figure 6.1. The starting point for the database design is the “StackTraces” table, which contains all of the stack traces that are collected for these experiments. One individual record in

The “StackTraces” table represents a single line of a single stack trace, with individual stack traces having a unique “stack_threat_id.” As an example, an individual stack trace with 22 lines in it would have 22 records in “StackTraces”, with a unique “stack_id” for each line and a single “stack_threat_id” for the 22 records. Crashes can contain multiple stack traces from different threads that are running at the time of the crash. Each stack trace record is mapped to a specific crash, organized in the “Crashes” table. Having separate tables for crashes and stack traces allows users to look for potential correlations with stack trace types that frequently appear together in the same crash. Each
“Crashes” record is mapped to a specific version of Windows, itemized in the “Products” table. For the “StackTraces” to “Crashes” mapping and the “Crashes” to “Products” mapping, a matching “Crashes” record or “Products” record must be assigned for a new “StackTraces” or “Crashes” record to be created. We add each crash to the database first, then add the associated stack traces to the database.

In addition to mapping stack traces and crashes to specific products, we also map individual code artifacts at the binary, file, and function level to specific artifacts in versions found in the “Products” table. Each level of granularity of code artifacts has its own associated table: “Binaries”, “Files” and “Functions”, respectively.

Each of these artifacts is given a unique identifier and is mapped to a specific version of Windows in the “Products” table. Similar to the “Crashes” mapping for the “StackTraces” table, new stack traces placed into “StackTraces” must have a valid entry in the “Binaries”, “Files”, and “Functions” tables. The mapping between binaries, files, and functions is performed by using string parsing to determine what the correct name is for each line in the stack trace.

Mapping binary names to binaries is straightforward because binaries must have unique names. However, mapping file and function names from stack trace entries to files and functions in the target system is more difficult due to the duplication of names of files and functions. To circumvent issues with mapping file and function names to their place in the system, we map code names in the following order: first binaries, then files, then functions. We can use the higher-level association to help our mapping technique in subsequent steps. The extra mapping step is assisted by the “BinaryFileMap” and “FunctionMap” tables. In the “BinaryFileMap” table, we associate specific “rp_id” (an internal convention to uniquely identify files in the Windows products) with specific “binary_id” entries. In the case of a file name that is not unique, we use the “BinaryFileMap” information to determine which unique file is being referenced. Similarly, for functions, we associate specific “function_id” entries to the “binary_id” and “rp_id” that the function is contained in. By performing function mapping last, we can use the previously parsed binary and file data for a specific stack trace record to determine which specific function is being referenced by the stack trace. In the case where we are unable to perform the mapping, each table has an “Unknown” record that these failed mapping are placed in, which satisfies the database schema. Each level of granularity has a frequency table, specified as “BinaryFrequency”, “FileFrequency”, and “FunctionFrequency.” Each table has a unique entry for each code artifact for each product. For each record, we then track three values used in our later metric generation step:

- Edge_count tracks how many unique incoming and outgoing edges a specific instance of a code artifact has in a list. For example, if “foo.cpp” has 5 unique ways to call into a file, then “edge_count” would have a value of 5 for the incoming entry in the list;

- Crash_count tracks how many times a code artifact appears in a unique crash. For our “foo.cpp”
example, appearing in 8 different crashes would mean the “crash_count” value would be set to 8. Note that the crash_count value represents unique occurrences on crashes; if “foo.cpp” appears 6 times in a single crash, it would only contribute one additional occurrence of “foo.cpp” to the “crash_count” entry; and

• stackLine_count is a count of the total number of times a code artifact appears in our datasets, with multiple occurrences in the same crash adding multiple entries. In our previous example of 6 occurrences of “foo.cpp” in a single crash, that would result in adding 6 to the “stackLine_count” entry for “foo.cpp.”

Finally, we map security vulnerability information to specific code artifacts found during our parsing of crash dump stack traces. We collected security vulnerability information at the file level, and map the vulnerability information to the “Files” table directly, along with the product the vulnerability was found in via the “Products” table. Individual vulnerabilities are also defined as pre-release or post-release, depending on when the vulnerability was found during the development process. Pre-release is defined as vulnerabilities found in code before its official release to customers, where official does not include customer alpha or beta releases. Post-release is defined as vulnerabilities found in code after an official release. We use post-release vulnerabilities as our vulnerabilities for the rest of our study. Using the “BinaryFileMap” and “FunctionMap” tables described previously, we can map individual security vulnerabilities to binaries and functions in addition to the direct mapping we have to files.

6.5 Change Metric Generation

We calculate the DC, NC, and PC metrics for three different comparisons: Windows 8 to 8.1, Windows 8.1 to 10, and Windows 10 (2014) to 10 (2015). The first two comparisons represent how the attack surface of Windows, as calculated by RASA, changes across major version updates. The Windows 10 (2014) to 10 (2015) comparison represents how the attack surface of Windows, as calculated by RASA, changes while a product is under development. We report both the number of files seen for each metric and the percentage of the total files seen across both parts of the comparison that fall into each of the three metrics. We cannot report what specific binaries, files, and functions were removed from RASA in one version and added to RASA in another version in our study. However, the data would be available to Windows security team members to make informed decisions on what code to add to regular review and what code to remove from regular review.
6.6 Complexity Metric Generation

We use the “ShapeEdge” tables from our database to generate our complexity metrics. Using the “edge” entries, we can determine the amount of unique incoming and outgoing calls to a specific code artifact at the binary, file, and function levels of granularity. For each code artifact, we store CFI and CFO into a temporary table, and then determine if there are any post-release security vulnerabilities associated with the code artifact in question. If there is at least one security vulnerability, we label that artifact with a 1.

After the result table has been generated, it can be used to generate supporting visualizations or charts for developers and managers. Examples could include pivot tables indicating the density of vulnerabilities in combinations of CFI and CFO, or the ability look at specific binaries, files and functions and determine whether the number of incoming and outgoing calls have changed.

For our study, we generate the log2ratio of CFI and CFO versus VFI and VFO, which indicates whether a specific permutation of CFI and CFO is more or less likely to contain a vulnerability than the average file in the system. We use the Spearman rank-order correlation coefficient to validate whether there is a correlation between more complex files (higher CFI and/or CFO) and vulnerability density, as measured by the log2ratio.

6.7 Boundary Metric Generation

We determine boundary metrics for Windows from the Windows-specific entities that appear first after external entities, and which external entities are seen directly before Windows entities. The boundary was determined by flagging each individual binary seen in our dataset as “Windows Related” or “Non Windows Related.” For each stack trace, we read each individual line, noting when the flag switched from “Non Windows Related” to “Windows Related” during a crash. The changeover was marked as the “boundary” for each stack trace. After the boundary is set, we can then determine the percentage of vulnerabilities that appear on the boundary.

6.8 Results

In this section, we describe the results of generating each type of metric.

6.8.1 RASA at source code file level

Table 6.2 contains a summary of the results from running RASA on Windows 8, Windows 8.1, and Windows 10 on a set of crash dump stack traces from 2014 and 2015. We excluded Windows 10 in 2014 due to the small number of crashes available because Windows 10 was still in development.
Table 6.2 - Percentage of files appearing on crash dump stack traces and vulnerabilities appearing on crash dump stack traces for operating system/year pairings.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Year</th>
<th>Code Coverage (CC)</th>
<th>Vulnerability Coverage (VC)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 8 (binaries)</td>
<td>2014</td>
<td>48.4%</td>
<td>94.8%</td>
<td>2.0</td>
</tr>
<tr>
<td>Windows 8 (Files)</td>
<td>2014</td>
<td>25.2%</td>
<td>51.3%</td>
<td>2.0</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>25.9%</td>
<td>50.2%</td>
<td>1.9</td>
</tr>
<tr>
<td>Windows 8.1 (Files)</td>
<td>2014</td>
<td>17.8%</td>
<td>37.5%</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>20.8%</td>
<td>40.7%</td>
<td>2.0</td>
</tr>
<tr>
<td>Windows 10 (Files)</td>
<td>2015</td>
<td>13.6%</td>
<td>29.2%</td>
<td>2.1</td>
</tr>
</tbody>
</table>

The table contains the CC and VC metrics, which are the percentage of code in the target software system that appears in at least one stack trace and the percentage of files with security vulnerabilities that appear in at least one stack trace, respectively.

The results in Table 6.2 represent a loss in VC (ranging from 29.2% to 51.3% of vulnerabilities) compared to the 94.8% reported by Theisen et al. [The15c]. However, our VC is at the file level instead of the binary level, which is a more actionable metric for developers, as binaries could contain hundreds of files unrelated to the vulnerability. Files are a smaller unit of measure for code, and easier for professionals to prioritize. All studies demonstrate a similar 2x ratio of code covered versus vulnerabilities covered. A prioritization scheme that results in twice as many vulnerabilities being found over random selection is an economic win for security teams, as they will find the same number of vulnerabilities in half the time. Based on our result, RASA is able to predict the location of 30-50% of vulnerabilities in target systems at the file level, indicating a trade-off between granularity and vulnerability coverage.

6.8.2 Change Metrics

Table 6.3 contains the Dropped Code (DC), New Code (NC), and Persistent Code (PC) metrics for our three comparisons. For the Windows 8 to 8.1 and Windows 8.1 to 10 comparisons, we see a significant difference in the files that are measured for each version, with 39% of the code measured changing between 8 to 8.1 and 58.7% of the code measured changing between 8.1 to 10. While the VC metric for each version of Windows is relatively similar (see Table 6.3), the significant change in code measured indicates that the RASA of each version of Windows has significantly changed.

6.8.3 Complexity Metrics

To evaluate whether increases in fan-in and fan-out values are correlated with more security vulnerabilities, we use the Spearman rank-order correlation coefficient. The Spearman correlations

<table>
<thead>
<tr>
<th>Version</th>
<th>Metric</th>
<th>Total Files</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 8 to 8.1</td>
<td>DC</td>
<td>7951</td>
<td>23.1%</td>
</tr>
<tr>
<td></td>
<td>NC</td>
<td>5490</td>
<td>15.9%</td>
</tr>
<tr>
<td></td>
<td>PC</td>
<td>21025</td>
<td>61.0%</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>13645</td>
<td>43.7%</td>
</tr>
<tr>
<td>Windows 8.1 to 10</td>
<td>NC</td>
<td>4677</td>
<td>15.0%</td>
</tr>
<tr>
<td></td>
<td>PC</td>
<td>12870</td>
<td>41.3%</td>
</tr>
<tr>
<td></td>
<td>DC</td>
<td>232</td>
<td>1.3%</td>
</tr>
<tr>
<td>Windows 10 - 2014 to 2015</td>
<td>NC</td>
<td>15497</td>
<td>89.0%</td>
</tr>
<tr>
<td></td>
<td>PC</td>
<td>1674</td>
<td>9.6%</td>
</tr>
</tbody>
</table>

Table 6.4 Spearman rank-order correlation coefficient for Fan-In and Fan-Out values compared to the log2ratio of the SF and VF metrics for Windows 8, 8.1, and 10.

<table>
<thead>
<tr>
<th>Windows Version</th>
<th>Year</th>
<th>Coefficient</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 8</td>
<td>Fan-In</td>
<td>0.63</td>
<td>p &lt;0.01</td>
</tr>
<tr>
<td></td>
<td>Fan-Out</td>
<td>0.75</td>
<td>p &lt;0.01</td>
</tr>
<tr>
<td>Windows 8.1</td>
<td>Fan-In</td>
<td>0.82</td>
<td>p &lt;0.01</td>
</tr>
<tr>
<td></td>
<td>Fan-Out</td>
<td>0.90</td>
<td>p &lt;0.01</td>
</tr>
<tr>
<td>Windows 10</td>
<td>Fan-In</td>
<td>0.65</td>
<td>p &lt;0.01</td>
</tr>
<tr>
<td></td>
<td>Fan-Out</td>
<td>0.83</td>
<td>p &lt;0.01</td>
</tr>
</tbody>
</table>

for Windows 8, 8.1, and 10 for fan-in and fan-out values are presented in Table 6.4. Each version of Windows expresses a positive correlation between increases in complexity and vulnerability density, ranging from 0.6 to 0.9 for both fan-in and fan-out. The p-values for each correlation coefficient are p < 0.01, indicating that the correlation is very unlikely to happen by chance.

Figures 6.2, 6.3, and 6.4 are graph representations of the Spearman correlation. For each graph, we see that vulnerability density, as measured by the log2 ratio, increases as both fan-in and fan-out values increase. Additionally, fan-in and fan-out values appear to increase together, with the trendlines for both fan-in and fan-out appearing close together with similar slopes. Based on our findings, we speculate that fan-in and fan-out measures from crash dump stack traces have similar impact on the likelihood that a file has a security vulnerability. Security professionals can use our result to prioritize security efforts for code that has the highest fan-in and/or fan-out values in their codebase, and work backwards from that point.
Table 6.5 Percentage of files appearing on the boundary of Windows, as measured by crash dump stack traces, and the percentage of files with vulnerabilities in the set.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Year</th>
<th>Boundary Code (BC)</th>
<th>Boundary Vulns. (BV)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 8</td>
<td>2014</td>
<td>4.5%</td>
<td>17.2%</td>
<td>3.8</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>4.6%</td>
<td>18.6%</td>
<td>4.0</td>
</tr>
<tr>
<td>Windows 8.1</td>
<td>2014</td>
<td>4.6%</td>
<td>16.5%</td>
<td>3.6</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>6.9%</td>
<td>23.7%</td>
<td>3.4</td>
</tr>
<tr>
<td>Windows 10</td>
<td>2014</td>
<td>3.4%</td>
<td>10.5%</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>3.9%</td>
<td>25.1%</td>
<td>6.4</td>
</tr>
</tbody>
</table>

6.8.4 Boundary Metrics

Table 6.5 contains the results from calculating our Boundary Code (BC) and Boundary Vulnerabilities (BV) metrics. We see that the ratios for BV versus BC are higher than the result for the entire codebase in Table 6.2, with vulnerabilities appearing on the boundary at a rate from 3.1 to 6.4 times higher than the rest of the codebase. However, the vulnerabilities covered by code on the boundary only represent about a quarter of the vulnerabilities in the system. Therefore, security efforts can prioritize the boundary as a first step, but follow-on work on the rest of the data flows in the system is necessary to find the rest of the vulnerabilities in the system.

6.9 Discussion

We have replicated the RASA approach for Windows 8, 8.1, and 10 at the file level, improving the granularity, and therefore applicability, of the approach. Further improvements in mapping specific files on crash dump stack traces to files in the system are needed to improve the VC metric. In our study, the attack surface of Windows, as measured by our approach, significantly changes from version to version and year to year. Therefore, security teams should be periodically re-computing their understanding of the attack surface of their systems, as new versions are released and new features are added to systems under development.

The Spearman rank-order correlation coefficient shows that that there is a correlation between the complexity a code artifact, as measured by the CFI and CFO metrics, and the likelihood that the code artifact has a security vulnerability. Our result is significant to $p < 0.01$. Our result is therefore consistent with prior work by Shin [Shi11] and Zimmermann [Zim10] indicating that code complexity is correlated with more security vulnerabilities.

Finally, vulnerabilities are denser on the boundary of the system compared to RASA and the average artifact in the system in our study. Security practitioners can use RASA to prioritize accord-
ingly: first the boundary of their systems, then artifacts on RASA, then the rest of the code in the
system. Our result supports work by Manadhata et al. [MW11] on identifying entry and exit points,
but suggests that entry and exit points are inefficient for complete vulnerability coverage.

Based on our results, we see that change, complexity, and boundary metrics derived from
-crash dump stack traces by RASA have positive correlations with security vulnerabilities found in
codebases. Practitioners could use these metrics to guide their efforts in finding and fixing potential
security vulnerabilities, and in refactoring and maintaining specific parts of their codebases that are
potentially susceptible to vulnerability introduction. A potential priority list, based on our findings,
is the following:

6.10 Limitations

Further exploration is needed of the use of stack traces from crash dumps as a potential source of
new software security and reliability metrics. Many companies already collect crash dump stack
-traces for their products, and reusing existing datasets that industry teams already maintain is one
potential avenue for providing immediate value to practitioners.

We have shown that RASA does change over time and from version to version for the Windows
product. However, we have shown the change over time of RASA on yearly cycles and major version
-releases. Looking at code changes from month to month or even week to week and seeing if it
-provides useful feedback for security professionals would be the next step to explore how effective
RASA could be, practically. Changing the version analysis from major version releases to minor
-changes (such as hotfixes) would also help in that respect.

Finally, building practical tools that plug into IDEs would be one way to get information from
RASA in front of developers and security professionals. However, surveys and interviews to clarify
the work environment of security professionals are needed to determine the best way to integrate
new information into security workflows.
Figure 6.2 A trendline of the distribution of files in Windows 8, based on the number of incoming and outgoing calls from a file.
Figure 6.3 A trendline of the distribution of files in Windows 81, based on the number of incoming and outgoing calls from a file.
Figure 6.4 A trendline of the distribution of files in Windows 10, based on the number of incoming and outgoing calls from a file.
We address the following research question with this chapter:

**Vulnerability Classifications (RQ5):** What classifications and severities of vulnerabilities appear on an attack surface approximated by crash dump stack traces in comparison to vulnerabilities that are not on crash dump stack traces?

### 7.1 Motivation

RASA uses crash dump stack trace information from target software systems to predict where potentially exploitable vulnerabilities might be. However, previous work in attack surface approximation and vulnerability prediction has treated vulnerable code as a binary classification: source code either has a vulnerability or it does not. Vulnerabilities come in different classifications, such as buffer overflow errors, memory corruption errors, or data processing errors. In addition to different classifications of vulnerabilities, vulnerabilities are not equal in terms of their impact. Some vulnerabilities have critical consequences for many users, such as the recent Heartbleed vulnerability [Dur14].

Evaluating which classifications and severities of vulnerabilities are covered by vulnerability prediction or prioritization approaches is important to understand the benefits and limitations of approaches for prioritizing security effort. Practitioners utilizing RASA for attack surface approximation could benefit from an understanding of what classifications of vulnerabilities are best detected by the model. Similar concerns exist for other approaches for attack surface generation.
and vulnerability prediction. Knowing the classifications of vulnerabilities on the attack surface may inform the proper context in which to use an approach to approximate the attack surface.

7.2 Methodology

In this section, we describe the methodology for identifying and classifying vulnerabilities and the use of crash dump stack traces to approximate the attack surface of a product.

7.2.1 Identifying Vulnerabilities

We identify vulnerabilities from the target software system in two stages. First, vulnerabilities explicitly defined as such by the maintainers of the software system are included in our dataset. Organizations that publicly report vulnerabilities the maintainers have found or were reported to them by outside entities can be found in several ways. First, such organizations may have a blog, newsletter, or mailing list for reporting security incidents, such as Mozilla’s Security Reports blog. Second, the organization's bug tracking repository may have a flag for security issues, such as “security” or “vulnerability”. Finally, researchers could consult the National Vulnerability Database (NVD), though they would need a way to map NVD entries to individual code artifacts related to the fix for that vulnerability. These bugs would be considered vulnerabilities for the dataset. We also mine potential vulnerabilities from the organization's bug tracking system, as research by Herzig et al. [Her13] indicates a high level of misclassification of bugs in open source projects. The specific keywords and process used for mining will depend on the software system in question. For details specific to our study, reference Section 7.3.

7.2.2 Classifying Vulnerabilities

After identifying the vulnerabilities for inclusion in the dataset, two raters then classify each vulnerability using an existing vulnerability classification scheme. For our approach, we use the Common Weakness Enumeration (CWE) set of most commonly seen weaknesses in software. We use, at maximum, the third level classification of weaknesses. Further specificity would result in too many classifications for meaningful results, as the classification tree has a maximum depth of eight at the time of this writing (June 13, 2018). We add the "Other" classification to the list of most common classifications of vulnerabilities to cover the case where a vulnerability does not fall into these categories. In the event that a significant amount of "Other" vulnerabilities are identified, further stratification may be needed. Using the above criteria, we narrowed our set of classifications from 113 to the following 24 classifications of vulnerabilities:

1\footnote{http://cwe.mitre.org/data/definitions/1003.html}
• Configuration

• Code

• Code - Data Processing Errors

• Code - Pathname Traversal and Equivalence Errors - Path Traversal

• Code - Pathname Traversal and Equivalence Errors - Link Following

• Code - 7PK Security Features - Credentials Management

• Code - 7PK Security Features - Permissions, Privileges, and Access Controls

• Code - 7PK Security Features - Improper Certificate Validation

• Code - 7PK Security Features - Cryptographic Issues

• Code - 7PK Security Features - Use of Insufficiently Random Values

• Code - 7PK Security Features - Insufficient Verification of Data Authenticity

• Code - 7PK Security Features - Improperly Implemented Security Check for Standard

• Code - 7PK Security Features - Protection Mechanism Failure

• Code - 7PK Time and State

• Code - 7PK Code Quality

• Code - Resource Management Error

• Code - Resource Management Error - Uncontrolled Resource Consumption

• Code - Resource Management Error - Improper Resource Shutdown or Release

• Code - Resource Management Error - Use After Free

• Code - Resource Management Error - Double Free

• Code - Channel and Path Errors - Uncontrolled Search Path Element

• Code - Channel and Path Errors - Unquoted Search Path or Element

• Environment

• Other
Further explanation of each classification can be found on the CWE website. The two raters individually classify each vulnerability in the targeted dataset. The rater looks at the description of each bug, along with the comment history and the diffs describing the fix for the vulnerability to determine a classification. After classifying each vulnerability, the raters then convene and resolve any differences that have occurred between the two of them. In the event that the two raters cannot come to a consensus, a third party arbitrator may be used to resolve the conflict. The initial independent classification of each vulnerability is preserved so the initial agreement between the two raters can be calculated.

We use Cohen's Kappa to evaluate the agreement between the two raters [Coh68]. Cohen's Kappa provides a quantitative measure of the agreement between classifiers or raters. Cohen's Kappa is useful because it takes into account the possibility of agreement by random chance, rather than properties of the object being rated or classified. We report Cohen's Kappa for sorting vulnerabilities into classifications once the vulnerabilities have been identified.

7.2.3 Vulnerability Severity

We use software-specific severity ratings for the vulnerability dataset, rather than using a global measure like the Common Weakness Scoring System (CWSS). A software-specific severity rating is a rating assigned by the maintainers of a software system in their own bug tracking system, such as Bugzilla. We use software-specific severity ratings based on the observation that security is contextual: a severe vulnerability for one organization may be a minor one for another, despite the mechanics of the vulnerability being identical. The severity of the vulnerability is linked to the consequences of the vulnerability, such as critical data leaking for one group while trivial data is leaked for another. Based on these observations, using internal severity measurements is superior to global measurements, such as the ones provided by CWSS. For our study, we use the severity ratings provided by the product engineers at Mozilla, according to a four point scale: critical, high, moderate, and low. These ratings are taken directly from the bug report entries in the Mozilla Bugzilla repository.

7.2.4 Risk-Based Attack Surface Approximation

We evaluate RASA in terms of RASA's ability to cover vulnerabilities of different classifications and severities. RASA is described in Chapter 4.

After a set of crashes have been collected for the target system and code artifacts have been identified as being on the attack surface of the system or off the attack surface, we then map individual vulnerabilities as collected to individual code artifacts in the software system. The process of identifying specific code elements and tying them to specific code artifacts for our case is described in detail in Section 7.3.
7.3 Case Study

In this section, we describe the steps taken to mine vulnerabilities and crash dump stack traces for Mozilla Firefox.

7.3.1 Vulnerability Data

Mozilla Firefox vulnerability data was first collected from Mozilla Foundation Security Advisories [Mozb]. Vulnerabilities were collected by the authors. For each bug, we recorded the following information:

- **Mozilla Security Report**: The short URL where the bug report was found. Multiple bugs were sometimes reported in a single blog post on the website.

- **File**: The source code file(s) modified to fix the bug.

- **Description**: A summary of the bug by the Mozilla security team.

- **Severity**: The severity of the defect. We used the internal classification system from the Mozilla team: critical, high, moderate, and low.

In addition, we mined Mozilla’s bug database directly for bugs that were not reported publicly on the blog as vulnerabilities, but exhibited characteristics of a vulnerability. We used a list of keywords determined by Shin et al. [Shi11] previously to perform the mining activity. The list of keywords used for mining potential vulnerabilities is found in Table 7.1. After collecting our potential vulnerabilities from Bugzilla, two of the authors independently classified each bug as a vulnerability or as not a vulnerability. If a bug was classified as a vulnerability, it was then added to the set of vulnerabilities taken from the Mozilla Security Advisories.

To determine what files were involved with a particular bug, the authors inspected the linked bug entries attached to each security report linking to Mozilla’s Bugzilla database2. For each bug, we specifically looked for a diff attached to the bug that had been positively reviewed by a security team member (indicated by the sec-approval+ tag), a release manager (indicated by approval-X or review+ tags), or both. In many cases, we chose the diff with the most reviews attached to the bug report. One exception is bugs labeled specifically with multiple diffs. These diffs would be labeled "Part 1", "Part 2", et cetera in the bug report attachments. If all the diffs had reviews from a security team member or a release manager, we included them in our vulnerability list.

In the case that the same file appears in multiple vulnerabilities, we took the following steps for classification and severity analysis. For classification, we recorded the list of all classifications for that particular file. For severity, we recorded the highest observed severity classification seen in our dataset.

---

2https://bugzilla.mozilla.org/
Table 7.1 The list of keywords used to collect potential unlabeled vulnerabilities from Mozilla's Bugzilla database on Mozilla Firefox.

<table>
<thead>
<tr>
<th>Search Keywords</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>secure</td>
<td>shutdown</td>
<td>integer</td>
<td>XSS</td>
</tr>
<tr>
<td>security</td>
<td>regression</td>
<td>signedness</td>
<td>denial service</td>
</tr>
<tr>
<td>vulnerable</td>
<td>incorrect</td>
<td>widthness</td>
<td>DOS</td>
</tr>
<tr>
<td>vulnerability</td>
<td>memory corruption</td>
<td>underflow</td>
<td>crash</td>
</tr>
<tr>
<td>fail</td>
<td>race</td>
<td>improper</td>
<td>deadlock</td>
</tr>
<tr>
<td>failure</td>
<td>racy</td>
<td>unauthenticated</td>
<td>SQLI</td>
</tr>
<tr>
<td>bug problem</td>
<td>buffer</td>
<td>gain access</td>
<td>SQL</td>
</tr>
<tr>
<td>error</td>
<td>overflow</td>
<td>permission</td>
<td>injection</td>
</tr>
<tr>
<td>crash</td>
<td>stack</td>
<td>cross site</td>
<td>format</td>
</tr>
<tr>
<td>string</td>
<td>CSS</td>
<td>attack</td>
<td>overrun</td>
</tr>
<tr>
<td>printf</td>
<td>bypass</td>
<td>CSRF</td>
<td>breach</td>
</tr>
<tr>
<td>scanf</td>
<td>backdoor</td>
<td>XSRF</td>
<td>violate</td>
</tr>
<tr>
<td>request forgery exploit</td>
<td>threat</td>
<td>forged</td>
<td>fatal</td>
</tr>
<tr>
<td></td>
<td>expose</td>
<td>hole</td>
<td>blacklist</td>
</tr>
</tbody>
</table>

When inspecting the individual diffs for each bugfix, we looked for C/C++ source code files in the diff. We specifically removed files from the "test" directory or with "test" in the name unless the bug specifically dealt with user access to test materials for Mozilla Firefox.

In some cases, files may have moved or been refactored out of Mozilla Firefox since the bug report was issued. For files flagged in bug reports that were not found in source control, we looked to see if the lines of code modified was still in place in source control using grep. If the lines of code were found in another file, we changed the entry in the vulnerable code list to match the current file in source control. If it was not found, we discarded that file from the vulnerable code list.

From our mining effort using keywords, we identified 308 bugs with their related source code files for potential inclusion in our set of vulnerabilities. Each rater looked at the information in the bug reports, such as the title of the bug, the change that fixes the bug, the description of the bug, or the comments in the bug. For each bug, the rater specified whether the bug should also be considered a vulnerability (TRUE/FALSE), and the classification of the vulnerability if it was declared as such, from the set of classifications above. After classifying these files, we were left with 88 bugs, associated with 111 source code files changed to fix a vulnerability. The inter-rater reliability between the two authors when classifying vulnerable files was \( \kappa = 0.6 \). Based on a scale for qualifying these figures, the two raters had Moderate to Substantial agreement [LK77]. We added these 111 files to the 232 files found on the Mozilla Security Advisories blog, resulting in a final list of 343 source code files that were involved with a vulnerability. We discarded 14 source code files from our list.
Figure 7.1 Distribution of classifications of vulnerabilities occurring on crashing code versus classifications occurring on non-crashing code. The percentage of vulnerabilities covered for a specific classification is next to the label. Vulnerability classifications with no entries in our dataset are omitted.

and changed 8 source code files based on the movement of code to another source code file or the removal of code from the system. Finally, we removed duplicate source code file entries in our list, resulting in 271 files with at least one vulnerability.

7.3.2 Crash Dump Stack Trace Collection

To collect crash dump stack trace data for RASA to use to find potentially exploitable source code in Mozilla Firefox, we use the site provided by the Mozilla team on their public crash dump stack trace database, Mozilla Crash Reports [Moza]. Mozilla provides an API for bulk crash collection efforts, leveraging their SuperSearch feature for collection of crashes under specific parameters.

We used the SuperSearch landing page to generate our URL for API calls. We searched for Firefox crashes for the nightly build of Firefox along with the public releases at the time of our study (59.0al 58.0b, and 57.0). We first collected a set of crash ID’s to parse, then parsed individual crash ID’s. For each crash ID, we checked the crashing thread for the crash and recorded all source code files involved with that thread. We then checked the list of source code files associated with the crash against source code files found in Mozilla Firefox source control. If a file was not found in source control, we discarded it. These discards represented a variety of different programs. Some source code files were from other Mozilla products, indicating integration with other products or, in some cases, crashes that were classified incorrectly by Mozilla Crash Reports. Some third party code
was also included in crashes, such as driver files or source code from underlying software like Rust\(^3\). While vulnerabilities may also be present in other Mozilla products or third party code, we considered these source code files out of scope.

Finally, the list of source code files that were found to have crashed, including the number of times a source code file was seen in a crash, was added as a feature to our three VPMs.

We started collection of historical crash data on November 12, 2017 and finished on November 22, 2017. A total of 1,141,519 crashes were parsed while collecting crash dump stack traces.

7.4 Results

In this section, we present the results of our study.

7.4.1 Coverage of Vulnerability Classifications

The distribution of coverage of different classifications of vulnerabilities by crashing code can be seen in Figure 7.1. The coverage of all vulnerabilities for our data set was 85.3%, meaning 85.3% of vulnerabilities in our dataset occurred in crashing code. Therefore, 85.3% was used as a baseline for each individual classification of vulnerability. Classifications of vulnerabilities with a higher percentage of coverage would represent strengths of the RASA approach, while categories with a lower percentage of coverage would represent weaknesses of the approach. In our study, four of the vulnerability classifications are lower than the baseline value of 85.3%: Traversal, Protection Mechanism Failure, Resource Management Error and Other. Based on this result, these four classifications would represent weaknesses of the RASA approach, or classifications of vulnerabilities the approach is more likely to miss. The remaining categories would represent strengths of the approach, or classifications of vulnerabilities the approach is more likely to cover.

7.4.2 Coverage of Vulnerability Severity

The distribution of coverage of different severities of vulnerabilities by crashing code can be seen in Figure 7.2. In our results, we see little variation between the three of the four different categories of severity, based on our previously defined baseline of coverage at 85.3%. Vulnerabilities classified as Critical and Moderate featured higher coverage, while High and Low vulnerabilities featured lower coverage, Low significantly so. Based on these results, using RASA does not seem to offer any benefits or drawbacks in terms of finding more severe vulnerabilities. The vulnerabilities not covered by RASA are relatively evenly distributed in terms of severity. While the Low category featured a significantly reduced coverage rate compared to the baseline, this is not a concern as this is the least severe category.

\(^3\)https://www.rust-lang.org/en-US/
Figure 7.2 Distribution of severity of vulnerabilities occurring on crashing code versus severities occurring on non-crashing code. The percentage of vulnerabilities covered for a specific severity is next to the label.

We also split our vulnerability data into two sets; the publicly reported vulnerabilities on the Mozilla Security Advisories blog, and the mined vulnerabilities from Bugzilla. We then counted the number of vulnerabilities in each severity level for each set of vulnerabilities to see if any differences existed in severity distribution between the two sets. Of the 232 vulnerable files from the security reports blog, 136 of them were classified as critical, 59 were classified as high, 27 were classified as moderate, and 10 were classified as low. Of the 111 vulnerable files from keyword-based mining, 55 were classified as critical, 4 were classified as high, and 51 were classified as normal/moderate. There was an initial expectation that vulnerabilities classified by the Mozilla team as Critical would be more likely to be reported on the blog, but this does not seem to be the case.

Based on the overall distribution of severity ratings for vulnerabilities done by the Mozilla Firefox team, there seems to be a bias towards rating vulnerabilities as Critical. Over 60% of the files associated with vulnerabilities were classified as Critical at least once. The lack of stratification of severity ratings for vulnerabilities suggests a possible issue with the scale used by the Mozilla team for rating severity.

7.5 Discussion

In this section, we discuss the results from our study on Mozilla Firefox, along with some recommendations on how these results apply to studies that use vulnerability data.

Several classifications of vulnerabilities were covered less frequently than the average coverage rate for crash dump stack traces. In particular, we were surprised to see that Resource Management Errors had a lower coverage rate than the average coverage rate (78.9% versus 85.3%). Upon further
inspection, vulnerabilities based on memory corruption issues as a subset of Resource Management Errors had a particularly low coverage rate, with only two of the seven memory corruption issues also being in crashing code. Our expectation was that crashes would cover the memory corruption classification of vulnerability at a higher rate than other classifications. The other classification with a lower coverage rate than the base line was code traversal issues. Part of that result could be explained by the low number of vulnerabilities in the traversal classification, with only two entries. For both of these failures in coverage, future work may be able to determine if the poor coverage of traversal issues is statistical noise or an issue with using crash dump stack traces as a security metric.

One interesting observation of our results is that 162 of the 271 vulnerabilities in our study were classified as critical, the highest possible level of severity. The high number of critical vulnerabilities indicates a significant skew in terms of the rating of vulnerabilities by the Mozilla team. If our observation holds for other software projects, it could indicate a potential issue with the rating of vulnerabilities in terms of their overall impact. If most vulnerabilities are rated as critical, then vulnerabilities are harder to distinguish from one another. Ideally, vulnerability severity classification should be more evenly distributed to draw meaningful conclusions about the distribution of vulnerabilities in a software system. However, practitioners are likely to rate any vulnerability found as more severe than a regular defect, as security vulnerabilities are considered significant events regardless of their magnitude. Therefore, researchers may be better served by ignoring severity ratings for vulnerabilities because of the problems in generating ratings in the first place.

Based on our results, our results indicate is that RASA covers vulnerabilities of a variety of different classifications, with the underrepresented categories featuring a small enough sample size that it may be a poor sampling. The coverage rate for specific severities of vulnerabilities does not vary significantly compared to the average coverage rate, leading us to observe that RASA does not miss the most severe security issues for Mozilla Firefox. We can recommend the use of RASA for prioritizing security efforts for Firefox, and practitioners can have confidence based on these results that they will not disproportionally miss the most severe issues in their product. The development of additional vulnerability datasets with classification and severity considerations would help determine whether our result generalizes to other products.

7.6 Limitations

Several threats to validity exist for our study. First, our classification system for vulnerabilities could be flawed, as it relies on the opinion of the people profiling the vulnerabilities. The approach for identifying potential vulnerabilities in Mozilla's Bugzilla database is based on keyword-based and human classification. There may be inaccuracies in that dataset, such as missing vulnerabilities or bugs misclassified as vulnerabilities. We mitigated the misclassification possibility by using two
raters, and having the two raters resolve their differences in classification. In the case where the raters could not come to a consensus, we excluded that potential vulnerability from the dataset.

We specifically only consider code that is considered first party Mozilla Firefox code, rather than any external libraries. The lack of consideration of external libraries was due to the lack of crash data available for those libraries. Additionally, the ability to generalize these results is limited to other software systems with similar functions, such as alternate web browsers.

We used the top level CVE definitions of vulnerabilities for our study. As the landscape of security vulnerabilities changes, these classifications of vulnerabilities may also change. Vulnerability datasets should be periodically updated as changes are made to vulnerability classification systems. In addition, analysis of the distribution of severity scores in a publicly reported system as the National Vulnerability Database (NVD) would confirm or deny the concept that vulnerability severity ratings tend to be skewed, as seen in our Mozilla Firefox dataset.

Some of the bugs described Mozilla Security Advisories blog are marked as private in the Mozilla Bugzilla database. Marking a vulnerability as private hides the vulnerability from public view, as only authorized accounts can see information about the vulnerability. Without the diff that fixes the vulnerability, we cannot include that vulnerability in our dataset.
We address the following research question with this study:

**VPM Comparison (RQ6):** Does risk-based attack surface approximation improve the precision and recall values of vulnerability prediction models when predicting the location of vulnerabilities in software system?

### 8.1 Motivation

Vulnerability Prediction Models (VPMs); e.g. [Zim10; Shi11; Sca14; The15c] are an approach for prioritizing security inspection and testing efforts to find and fix vulnerabilities. Predicting which specific source code files or functions contain vulnerabilities would be a valuable tool for development teams looking to secure their products. To date, low precision and recall values of VPMs have prevented their practical use [Mor15].

Users of Defect Prediction Models (DPMs) and VPMs balance data collection effort, missing vulnerabilities (false negatives), and wasted effort (false positives) when weighing which model to use. Organizations will value these factors differently; a national defense organization may want an approach with the smallest false negative rate, regardless of the extra effort wasted in the process. Outlining these tradeoffs in advance is important, as practitioners will typically only choose a single
prediction model for their product.

Here is an example of these usability tradeoffs. A local startup is considering using VPMs to find vulnerabilities in their product before release. VPM X claims to have accuracy far exceeding its competitors, but it requires 60 features, 50 of which the startup is not currently collecting. On the other hand, VPM Y has worse accuracy measures, but only requires three easily-collected features for prediction. Understanding performance tradeoffs beyond runtime and accuracy is key for technique adoption in practice. Available VPMs should, therefore, be compared to one another on the same target software system to determine the strengths and weaknesses of each approach. Once each of these VPMs have been evaluated, combining the features of each VPM may result in a model with better accuracy. An assessment of which approaches to vulnerability prediction result in the best accuracy, alongside resource usage considerations, is important for the practical implementation of VPMs.

This chapter describes two studies. The first study is a comparison of the Zimmermann et al. VPM [Zim10] with and without crash data to augment the predictions made by the model. The second study compares four different vulnerability prediction approaches, along with combinations of each approach. The four approaches involved a broad sampling of software metrics, churn and complexity software metrics, a source code text mining approach, and RASA.

8.2 Vulnerability Prediction Model - Windows 8

First, we explore how limiting VPMs to analyzing the approximated attack surface affects the accuracy of VPMs. As discussed earlier, VPMs predict where vulnerabilities might appear in a particular codebase based on software engineering metrics, such as code churn, code complexity, code dependencies, and pre-release vulnerabilities, among others. Performance of these VPMs [Zim10; Shi11; Sca14; The15c] have not reached the accuracy levels needed (in terms of precision and recall) to be used practically on software projects for vulnerability prediction. One of the reasons for the lack of accuracy is the unbalanced nature of the vulnerability prediction problem. Vulnerabilities are often found in less than 1% of the codebase [Shi11]. Adding an attack surface approximation technique such as RASA to these VPMs may have a significant effect on the accuracy of these models, as the approximated attack surface would eliminate irrelevant code from consideration for containing vulnerabilities.

To compare the effect of adding RASA to an existing VPM, we replicate a VPM both with and without crash data to determine whether limiting a VPM to the approximated attack surface can improve predictive performance. We replicated a VPM for Windows, as published by Zimmermann et al. [Zim10]. The experiments conducted by Zimmerman et al. were conducted on datasets collected for Windows 8, a product of significant size. Choosing the same product as the original study enables a comparison with the original study, giving insight in to how vulnerability prediction metrics in a
codebase change over time.

The VPM described by Zimmermann et al. [Zim10] is based on static code metrics and pre- and post-release vulnerabilities. As an analogue to the original study, we gathered these measurements using CODEMINE [Cze13]. Microsoft developed CODEMINE to allow the company to monitor the development attributes of its products during development and after product release. The CODEMINE process provides a central repository of development and vulnerability metrics that were used within our research study. We refer to the VPM as Software Metrics: Broad (SM:B).

The SM:B VPM using has several limitations. Many of the features collected for Microsoft Windows do not generalize outside of Microsoft, particularly metrics related to the organization of software teams. Even within Microsoft, reorganization of software teams would require the new measurements of team organization to be validated again against vulnerability data. Without these organizational metrics, the VPM is almost overlapped entirely by the Churn and Complexity approach, as seen in Tables 8.1 and 8.2.

### 8.2.1 Crash Dump Stack Traces

The stack traces used in our Windows study are from the Windows 8.1 operating system. Stack traces for the Windows study come from three complementary sources: fuzzing crashes, user mode crashes, and kernel mode crashes. We keep track of the source of each crash as we parse each stack trace. We now describe each crash source.

Microsoft’s security testing teams generate fuzzing crashes. Fuzzing is a testing strategy revolving around sending random or deliberately malformed/malicious data to the input points on a system. The goal of fuzzing is to simulate attacks and to get the system to behave in an unexpected manner. Typically, any response from the system that differs from a standard error message is flagged and investigated. Possible results from fuzz testing include crashes, memory leaks, and security bugs such as data loss or improper access. Fuzzing crashes are a useful data source as any input generated and fed into the system under test could also originate from a user and thus from a potential hacker. Fuzzing aims to discover security vulnerabilities, and any irregular fuzzing results are relevant for determining what code areas were involved in handling the malformed input.

Windows users generate user mode crashes. User mode crashes are field/customer crashes that are not due to hardware failures. Crashes in applications running without administrator rights triggers collection of crash information that may be sent to Microsoft. The system responsible for collecting the data, *Microsoft Error Reporting*, is executed under the same user privileges as the user that ran the crashed application. However, running in user mode limits the stack trace generator's ability to access and resolve which resources were involved in the crash. Thus, user mode crashes may only identify code areas that are accessible by the user in that mode.

Kernel crashes occur on field and customer machines, but unlike user crashes, they occurred
within the Windows kernel. In general, kernel crashes indicate more severe failures and usually include full resolutions of code artifacts, as long as they are part of the Windows system. The kernel is running under administrative privileges allowing the stack trace generator to gather more details about the system's granular details.

8.2.2 Code Metrics

The VPMs developed by Zimmerman et al. [Zim10] and replicated in our study are based on 29 metrics broadly classified into 6 categories:

- Churn metrics [NB05]. Churn measures are relative to a time period; the period for all presented calculations is between the start and RTM date of the project.

- Complexity metrics [EE01]. More complicated code is more likely to exhibit errors.

- Dependency metrics [Geg08]. The degree to which a piece of code is depended upon, or depends upon other code, influences its impact on software vulnerabilities.

- Legacy metrics. Metrics of particular interest to Microsoft. The importance of security in the development of software at Microsoft began receiving increased attention after the Bill Gates' 2002 Trustworthy Computing Memo [Ass03], with significant investments made in security training, tools, and process [HL06]. Code written after these processes were put in place has had a higher, more process-driven, level of attention to security applied in its design, construction and testing. These metrics verify the theory that code written before the security reset may be more likely to contain vulnerabilities.

- Size metrics. Larger source files are more difficult to mentally manage, and, therefore, are more prone to defects and vulnerabilities.

- Pre-Release vulnerabilities. For VPMs predicting post-release vulnerabilities, we used pre-release vulnerabilities to model usual suspects.

Depending on the metric, data was available at either the source file level or at the function level. In cases where function level data was present, amounts were aggregated up to the file level via averages, totals and maximums. Binary-level data was obtained by aggregating source-file level data up to the binary in which each source file is used. Our study uses additional metrics that were not available at the time of the original study by Zimmermann at al. [Zim10]. The table identifies which metrics are common between the two studies and which are unique to our study.

All size, churn, complexity, and dependency metrics were measured as of each releases’ Release to Manufacturing date.
8.2.3 Pre and Post-Release Vulnerabilities

As dependent variables, we used the number of post-release security vulnerabilities detected and fixed within the corresponding source files and code binaries respectively. Pre-release vulnerabilities were used as independent variables. Pre-release vulnerabilities are issues that are identified and fixed during software development. A post-release vulnerability is a security issue detected and corrected after releasing the corresponding software product to the public. Pre-release vulnerabilities of product version N may also be post-release vulnerabilities for product version N-1. Post-release security changes can be considered as 'escapes' from the software development lifecycle (SDL). Escapes may be worthy of investigation for SDL application in future releases.

To identify post-release vulnerability fixes, we counted the number of code changes applied in Windows service pack branches marked as security fix. These branches serve as a sink of defect fixes that will eventually be shipped to customers as part of a service pack or hot-fix. No feature development is permitted on these branches. Pre-release vulnerabilities were identified by bug reports marked as security vulnerabilities that resulted in changed source files and binaries.

8.2.4 Prediction Models

We ran our prediction models at both the binary and source level granularity. For both levels of granularity, we build classification models to identify code entities that had at least one vulnerability. For each level of granularity, we split the overall data collection into a training dataset that contains 2/3 of all data points. The remaining data is used for testing purposes. To split the data, we used stratified sampling; the ratio of code entities associated with vulnerabilities from the original dataset is preserved for both subsets. We repeatedly sampled the original dataset 100 times (100-cross-fold-validation). In total, we generated 200 independent training and testing sets: two levels of granularity and 100-cross folds each (similar to [Zim10; ZN08]).

We remove all code entities that are not part of the attack surface of Windows 8.1 from both training and testing sets before sampling takes place. Thus, classification models referring to attack surfaces contain less data points. We conducted the experiments using the R statistical software [Tea13] (Version 3.10). Instead of using the original feature vectors provided by the raw metric values, we applied R's prcomp [VR13] procedure to our data to produce principal components. Principal Component Analysis (PCA) [KPF01] reduces redundancy in our matrix of metrics and observations by maximizing the variance of linearly independent variables. Deciding how many of these variables to use in model building typically takes one of two forms; either a limit on the number of terms in the model is set, or some total amount of variance to be accounted for by the model is set. We selected principal components that accounted for 95% of variance.

In pursuit of high prediction performance, we used Max Kuhn's R package caret [Kuh15] to build VPMs based on the components selected by PCA. We used a Random Forest [Bre01] machine
learning technique to generate our prediction. Random Forest is a variant of decision trees that can be represented as a binomial tree, and is popularly used for classification tasks. We chose Random Forest because vulnerability datasets are highly unbalanced (many entities have no vulnerability, very few have some) and Random Forest is particularly good in handling unbalanced datasets.

### 8.2.5 Data Sources

The stack traces used in our study are from the Windows 8 operating system. Stack traces for our study come from three complementary sources: fuzzing crashes, user mode crashes, and kernel mode crashes. We keep track of the source of each crash as we parse each stack trace. We now describe each crash source.

Microsoft’s security testing teams generate fuzzing crashes. Fuzzing is a testing strategy revolving around sending random or deliberately malformed/malicious data to the input points on a system. The goal of fuzzing is to simulate attacks and to get the system to behave in an unexpected manner. Typically, any response from the system that differs from a standard error message is flagged and investigated. Possible results from fuzz testing include crashes, memory leaks, and security bugs such as data loss or improper access. Fuzzing crashes are a useful data source as any input generated and fed into the system under test could also originate from a user and thus from a potential hacker. Fuzzing aims to discover security vulnerabilities, and any irregular fuzzing results are relevant for determining what code areas were involved in handling the malformed input.

Windows users generate user mode crashes. User mode crashes are field/customer crashes that are not due to hardware failures. Crashes in applications running without administrator rights triggers collection of crash information that may be sent to Microsoft. The system responsible for collecting the data, *Microsoft Error Reporting*, is executed under the same user privileges as the user that ran the crashed application. However, running in user mode limits the stack trace generator's ability to access and resolve which resources were involved in the crash. Thus, user mode crashes may only identify code areas that are accessible by the user in that mode.

Kernel crashes occur on field/customer machines, but unlike application crashes, they occurred within the Windows kernel. In general, kernel crashes indicate more severe failures and usually include full resolutions of code artifacts, as long as they are part of the Windows system. The kernel is running under administrative privileges allowing the stack trace generator to gather more details about the system's granular details.

To measure the effectiveness of our approach, we required a set of vulnerabilities to compare our approach against. We therefore use the set of vulnerabilities seen in Windows 8 both pre- and post-release of the product. We then check to see if these vulnerabilities appear on our attack surface approximation to determine whether they would be missed by our approach.
8.2.6 Results

We first calculate the precision and recall of our VPM runs when considering the entire codebase and only code on our attack surface approximation. Compared with vulnerability prediction models (VPMs) run on the entire codebase, VPMs run on the attack surface approximation improved recall from 0.07 to 0.1 for binaries and from 0.02 to 0.05 for source files. Precision remained at 0.5 for binaries, while improving from 0.5 to 0.69 for source files. While these figures are low due to the scarcity of vulnerabilities, our result represents improvement over the state of the art in an open research area.

The initial results of applying the attack surface concept to VPMs are encouraging. We have demonstrated a statistical improvement in the accuracy of VPMs when the set of artifacts is narrowed down at both the binary and file level, and continual improvement based on better targeting of test data is possible. By continually improving the accuracy of these VPMs, we hope to allay developer concerns about them and hope to see additional use of them to detect vulnerabilities.

8.3 Vulnerability Prediction Models Comparison - Firefox

In this section, we describe two additional VPMs that will be compared to the Zimmermann and RASA approaches on Mozilla Firefox.

8.3.1 Software Metrics - Churn and Complexity

Shin et al. [SW08; Shi11] focus on churn and complexity when predicting which parts of a codebase are likely to have vulnerabilities. We refer to this VPM as Software Metrics: Churn and Complexity (SM:CC). Shin et al. [Shi11] use linear discriminant analysis, Bayesian network modeling, and Random Forest to build the prediction models based on the features in Table 8.2.

Shin et al. were able to reduce the amount of code inspected for security efforts by 71% for Mozilla Firefox and 28% for the RedHat Enterprise Linux kernel. In another study on Mozilla Firefox, they report 12% precision and 83% recall using similar metrics [SW13]. Shin et al. explored execution complexity metrics versus static complexity metrics, and found that the execution complexity metrics outperformed their counterpart for Mozilla Firefox and Wireshark in terms of File Inspection Reduction (FIR and LOC Inspection Reduction (LIR) [SW11]. For Firefox, execution metrics resulted in FIR of 0.65 versus 0.63 for static metrics. Execution metrics had an improved LIR value compared to static metrics (0.41 versus 0.37). For Wireshark, execution metrics resulted in FIR of 0.62 versus 0.31 for static metrics. Execution metrics had an improved LIR value compared to static metrics (0.67 versus -0.04). The Churn and Complexity VPM’s limitations are similar to the Broad VPM’s limitations in that some features may not generalize to other products. The Churn and Complexity
studies have reported high recall values but poor precision values, meaning organizations that value finding as many vulnerabilities as possible without cost concerns would choose the Shin approach.

### 8.3.2 Text Mining

Scandariato and Walden et al. [Sca14; Wal14] compared text mining and software metric approaches to vulnerability prediction. In particular, Scandariato and Walden use a “bag of words” approach to text mining [Zha10]. The bag-of-words model tokenizes source code, splitting the frequency in which individual strings appear in a source code file into a single feature in the software system. For example, if the value “NULL” appears three times in a source code file, the “NULL” feature for that file would be three.

Scandariato and Walden found that text mining tokens resulted in better precision and recall for vulnerability prediction for three projects. For Drupal, text mining had higher mean precision (.57 for text mining versus .52 for software metrics) and recall (.81 versus .77) values when compared to software metric based approaches. Similarly, for PHPMyAdmin, text mining had higher mean precision (.14 versus .13) and recall (.74 versus .66) values. However, Drupal and PHPMyAdmin are relatively smaller projects than Moodle, which had five times as many tokens (18,306) as Drupal (3,886) and PHPMyAdmin (5,252). Text mining still outperformed software metrics in terms of recall (.8 versus .7), but had relatively equal precision values (.02 for both approaches). Text mining and software metrics both suffered from a heavy loss of precision as the product size increased in their study. The authors have made two replication sites available which informed our construction of a C/C++ version of their approach [Scaa; Scab].

The text mining approach to vulnerability prediction has several limitations. First, one of the studies on the topic [Sca14] used all (unfiltered) static analysis alerts as a proxy for potentially vulnerable code which resulted in higher precision and recall than other VPM studies. However, researchers and practitioners have shown that static analysis have varying accuracy, particularly for security related issues [Mos07; Kra05]. Austin et al. found that vulnerability analysis tools produce a high number of false positives [AW11]. Later studies on text mining used author-validated vulnerability sets [Wal14]. Additionally, text mining takes a significant amount of time to run and has a large disk space requirement, as discussed later in section 7.4. The time taken to tokenize code into features is of concern, as it means that a text mining approach to vulnerability prediction would be incompatible with a continuous deployment workflow. Setting a maximum number of unique tokens to consider as features could solve the disk space issue, but not the computation time issue.

### 8.3.3 Methodology

In this section, we describe our methodology for comparing VPMs.
8.3.3.1 VPM Comparison

Classification of vulnerable code is performed at the source code file level. Our classification model attempts to split the dataset into two parts: one part with source code files that have at least one vulnerability, and one part with source code files that have no vulnerabilities. We used a training/testing approach to testing our model, with our training set containing 70% of our data points. The remaining data is used for testing purposes.

We conducted the experiments using SciKit Learn for Python [Ped11]. We ran each model from section 8.3, against our target system to predict vulnerabilities. We ran each model using X different classifiers: Random Forest, Naive Bayes, Logistic Regression, and Classification and Decision Trees (CART). We used Random Forest and Naive Bayes as our two classifiers, as these two classifiers outperformed Logistic Regression and CART in all cases. In addition to precision and recall, we also measure the F-Score of the VPM, or the harmonic average of precision and recall. Higher indicates better overall accuracy, assuming that precision and recall are weighted equally.

These measures taken together represent the accuracy of each VPM in our study. We present these measures separately so the strengths of each model can be determined in lieu of a single accuracy figure. For example, one model may have high recall but low precision, indicating that the model has a high false positive rate. Another model may have high precision but low recall, indicating that the model rarely gives false positives but misses many vulnerabilities.

We collect the following performance measures for each VPM:

- **Data Collection:** The amount of time required to collect data for the VPM.
- **RAM:** The maximum amount of RAM used during the classification process.
- **Runtime (Classifier):** The time it takes to run the VPM for a given classifier after data has been collected.
- **Disk Space:** The amount of space the data needed for a particular VPM takes up on hard drives.

We note the number of metrics or features that have to be collected to run the VPM. Note that for the text mining approach, we do not consider every feature string an individual feature; the feature count is one, as the tokenizer only needs to be run once. We also note the number of metrics or features that were able to be measured for our replication. For example, if an original study had 30 metrics but only 20 could be collected for our replication, that may be a consideration for practitioners looking to implement that VPM.

We also discuss the degree of difficulty in mining each feature in Section 7.5, including any particular roadblocks we discovered.
8.3.3.2 Combining Features

After constructing each VPM independently, we explore how additional features may effect the accuracy of each model. Specifically, we combine the features from the four VPMs in our study and run the Random Forest and Naive Bayes classifiers on those features. We take the same prediction measures from our initial run of each VPM and compare them to each combined result, giving us a total of seven different permutations to compare:

- Software Metrics - Broad [Zim10]
- Software Metrics - Churn and Complexity [Shi11]
- Text Mining [Sca14]
- Crashes [The15c]
- Software Metrics (All) and Text Mining
- Software Metrics (All) and Crashes
- Text Mining and Crashes
- Software Metrics (All), Text Mining, and Crashes

8.3.4 Mozilla Firefox Case Study

In this section, we describe the specific steps taken to mine features for each of our four VPMs on Mozilla Firefox. Our study focuses on C and C++ files in Mozilla Firefox. Features were collected for 28,750 unique source code files in the project.

8.3.4.1 Software Metrics

Depending on the metric, data is presented at either the source code file level or the function/method level. In the case of function/method level metrics, metric values were aggregated up to the source code file level to provide a consistent point of comparison across the four VPMs.

We use Understand [Sci17] to generate software metrics that were the closest match to metrics from the original study for each of the software metrics VPMs. Table 8.1 provides our best mapping for each metric for the Zimmermann et al. approach, while Table 8.2 provides our best mapping for each metric in the Shin et al. approach. All software metrics were collected using Understand on Mozilla Firefox as of November 21st, 2017.

Understand has been used previously to generate metrics for software researchers for a variety of purposes [MK13; RD13]. The mapping of complexity metrics from Understand to metrics from
the Development Metrics and Complexity Metrics VPMs can be found in Tables 8.1 and 8.2. Shin et al. [Shi11] used Understand in 2011 as their tool for collecting complexity metrics for their study. Some of the metrics that were used by Shin in 2011 are deprecated in newer versions of Understand in favor of existing or newer metrics: we have noted these changes in Table 8.2. Of note is the name changes to SumFanIn and SumFanOut, along with the collapse of SumMaxNesting and MaxMaxNesting into a single aggregated metric called MaxNesting. For our replication, we consider MaxNesting to be a single feature of the VPM.

Churn metrics were collected for Mozilla Firefox from Mozilla's source control system, Mercurial [O’S09]. Three churn metrics were collected from source control:

- **No. of Engineers (Zimermann et al. [Zim10]):** The number of unique developers who edited a source code file, or "No. of Engineers" from the Software Metrics VPM.
- **NumChanges (Shin et al. [Shi11]):** Total number of changes committed to source control for a specific source code file.
- **LinesChanged (Shin et al. [Shi11]):** Total number of lines of code changed for a specific source code file across all commits.

Metrics we do not include in our study on Mozilla Firefox, along with an explanation of why they were excluded, are listed below.

- **Repeat Frequency:** While Mozilla Firefox does have different versions we could consider for the Repeat Frequency metric, builds are done nightly and the resultant metric for Firefox would be Frequency again.
- **Block Coverage:** To our knowledge, no public tool exists that provides an equivalent to this metric.
- **Arc Coverage:** To our knowledge, no public tool exists that provides an equivalent to this metric.
- **Incoming closure:** To our knowledge, no public tool exists that provides an equivalent to this metric. This metric is covered partially by SumFanIn.
- **Outgoing closure:** To our knowledge, no public tool exists that provides an equivalent to this metric. This metric is covered partially by SumFanOut.
- **Number of Ex-Engineers:** For an open source project, employment as described in the original study is not relevant concept. We do calculate Number of Engineers based on the commit history, but we cannot determine whether an “employee” has left the project; time between commits for an individual user could indicate someone taking a break from open source development, not necessarily an “ex-contributor.”
Table 8.1 List of features included in our best-effort replication of the VPM described by Zimmermann et al. [Zim10]

<table>
<thead>
<tr>
<th>Code Churn Features</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Churn</td>
<td>Total number of lines changed for the source code file.</td>
<td>hg churn -v</td>
</tr>
<tr>
<td>Frequency</td>
<td>Number of times the source code file was changed.</td>
<td>hg churn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Complexity Features</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cyclomatic Complexity</td>
<td>The number of linearly independent paths through the source code file.</td>
<td>Understand - Cyclomatic</td>
</tr>
<tr>
<td>Fan-in</td>
<td>Number of incoming calls to the source code file.</td>
<td>Understand - CountInput</td>
</tr>
<tr>
<td>Fan-out</td>
<td>Number of outgoing calls to the source code file.</td>
<td>Understand - CountOutput</td>
</tr>
<tr>
<td>LoC</td>
<td>Number of lines of code in the source code file.</td>
<td>Understand - CountLineCode</td>
</tr>
<tr>
<td>Weighted Methods per Class</td>
<td>Number of declared methods per class in the file, if applicable.</td>
<td>Understand - CountDeclMethod</td>
</tr>
<tr>
<td>Depth of Inheritance</td>
<td>The size of the inheritance tree for the methods and classes in the source code file, aggregated.</td>
<td>Understand - MaxInheritanceTree</td>
</tr>
<tr>
<td>Coupling between Objects</td>
<td>The coupling of the classes in the source code file.</td>
<td>Understand - CountClassCoupled</td>
</tr>
<tr>
<td>Number of subclasses</td>
<td>The number of subclasses found in the source code file.</td>
<td>Understand - CountClassBase</td>
</tr>
<tr>
<td>Total Global Variables</td>
<td>The number of global variables declared in the source code file.</td>
<td>Understand - CountDeclInstanceVariablePublic</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Organizational Features</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Engineers</td>
<td>The number of unique engineers that edited the source code file.</td>
<td>hg churn</td>
</tr>
</tbody>
</table>
Table 8.2 List of features included in our best-effort replication of the VPM described by Shin et al. [Shi11]. Descriptions for complexity metrics from SciTools [Sci17].

<table>
<thead>
<tr>
<th>Code Complexity Features</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>CountLineCode</td>
<td>Number of lines containing source code. [aka LOC]</td>
<td>CountLineCode</td>
</tr>
<tr>
<td>CountDeclClass</td>
<td>Number of declared classes in the source code file.</td>
<td>CountDeclClass</td>
</tr>
<tr>
<td>CountDeclFunction</td>
<td>Number of Declared functions in the source code file.</td>
<td>CountDeclFunction</td>
</tr>
<tr>
<td>CountLineCodeDecl</td>
<td>Number of lines containing declarative source code.</td>
<td>CountLineCodeDecl</td>
</tr>
<tr>
<td>CountLinePreprocessor</td>
<td>Number of preprocessor lines.</td>
<td>CountLinePreprocessor</td>
</tr>
<tr>
<td>SumEssential</td>
<td>Sum of essential complexity of all nested functions or methods.</td>
<td>SumEssential</td>
</tr>
<tr>
<td>SumCyclomaticStrict</td>
<td>Sum of strict cyclomatic complexity of all nested functions or methods.</td>
<td>SumCyclomaticStrict</td>
</tr>
<tr>
<td>SumMaxNesting</td>
<td>Maximum nesting level of control constructs.</td>
<td>MaxNesting</td>
</tr>
<tr>
<td>MaxCyclomaticStrict</td>
<td>Maximum strict cyclomatic complexity of nested functions or methods.</td>
<td>MaxCyclomaticStrict</td>
</tr>
<tr>
<td>MaxMaxNesting</td>
<td>Maximum nesting level of control constructs.</td>
<td>MaxNesting</td>
</tr>
<tr>
<td>SumFanIn</td>
<td>Number of immediate base classes. [aka IFANIN]</td>
<td>CountClassBase</td>
</tr>
<tr>
<td>SumFanOut</td>
<td>Number of called subprograms plus global variables set. [aka FANOUT]</td>
<td>CountOutput</td>
</tr>
</tbody>
</table>

**Code Churn Features**

<table>
<thead>
<tr>
<th>NumChanges</th>
<th>Number of total changes to the source code file.</th>
<th>hg churn</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinesChanged</td>
<td>Number of total lines changes in the source code file across all edits.</td>
<td>hg churn -v</td>
</tr>
</tbody>
</table>

83
• *Edit Frequency*: For Mozilla Firefox, this metric is identical to Frequency, as Firefox is developed on a continuous basis.

• *Depth of Master Ownership (DMO)*: As this is an open source project, no organizational tree exists for individual components of Firefox. There is an owner of the Firefox project along with peers on the project, but not for individual components of Firefox. In addition, this concept would only contain sufficient data in the case of software products that are mature and have been through many releases, invalidating it for newer and smaller projects.

• *Percentage of Org contributing to development*: Because the DMO concept does not exist, this metric does not exist.

• *Level of Organizational Code Ownership*: Because the DMO concept does not exist, this metric does not exist.

• *Organization Intersection Factor*: Because the DMO concept does not exist, this metric does not exist.

For our replication, we have determined that 11 of the 29 metrics used in Zimmermann et al. [Zim10] are not relevant for Mozilla Firefox, and are likely not relevant for the majority of open source projects. We discuss the issue of metric replication in Section 7.5. Each of these metrics were included as features in models using software metrics.

### 8.3.4.2 Text Mining

The original studies and replication packages by Scandariato et al. [Sca14; Scaa; Scab] are available for PHP and Android projects. For our replication, we had to build a custom tokenizer for C/C++ code. We wrote custom Python scripts for tokenizing C/C++ code, making use of PyTokenize [Pyt]. The scripts used for our replication can be found on GitHub\(^1\). We used the SciKit Learn implementations of Naive Bayes and Random Forest for consistency with the other VPMs in our study, as different implementations of algorithms in different machine learning packages may have different performance.

### 8.3.4.3 Crash Dump Stack Traces

To collect crash dump stack trace data for Mozilla Firefox, we use the site provided by the Mozilla team on their public crash dump stack trace database, Mozilla Crash Reports [Moza]. Mozilla provides an API for bulk crash collection efforts, leveraging their SuperSearch feature for collection of crashes under specific parameters.

---

\(^1\)https://github.com/theisencr/c-code-tokenizer
We used the SuperSearch landing page to generate our URL for API calls. We searched for Firefox crashes for the nightly build of Firefox along with the public releases at the time of our study (59.0a1 58.0b, and 57.0). We first collected a set of crash ID’s to parse, then parsed individual crash ID’s. For each crash ID, we checked the crashing thread for the crash and recorded all source code files involved with that thread. We then checked the list of source code files associated with the crash against source code files found in Mozilla Firefox source control. If a file was not found in source control, we discarded it. These discards represented a variety of different programs. Some source code files were from other Mozilla products, indicating integration with other products or, in some cases, crashes that were classified incorrectly by Mozilla Crash Reports. Some third party code was also included in crashes, such as driver files or source code from underlying software like Rust. While vulnerabilities may also be present in other Mozilla products or third party code, we considered these source code files out of scope.

Finally, the list of source code files that were found to have crashed, including the number of times a source code file was seen in a crash, was added as a feature to our three VPMs.

We started collection of historical crash data on November 12, 2017 and finished on November 22, 2017. Crashes were collected over the period of a year matching our vulnerability data, starting in January 2017 to the beginning of our collection period. A total of 1,141,519 crashes were parsed while collecting crash dump stack traces.

### 8.3.4.4 Vulnerability Data

Mozilla Firefox vulnerability data was first collected from Mozilla Foundation Security Advisories [Mozb]. Vulnerabilities were collected by the authors. For each bug, we recorded the following information:

- **Mozilla Security Report**: The short URL where the bug report was found. Multiple bugs were sometimes reported in a single blog post on the website.

- **CVE-ID**: The CVE-ID³ associated with each vulnerability.

- **File**: The source code file(s) modified to fix the bug.

- **Description**: A summary of the bug by the Mozilla security team.

- **Type**: The type of security flaw the bug represents. Type information is usually in the first few words of the description.

  In addition, we mined Mozilla's bug database directly for bugs that were not reported publicly on the blog as vulnerabilities, but exhibited the characteristics of a vulnerability. We used a list of

---

³https://cve.mitre.org/
keywords determined by Shin et al. [Shi11] previously to perform the mining activity. After collecting our potential vulnerabilities from Bugzilla, two of the authors independently classified each bug as a vulnerability or as not a vulnerability. If a bug was classified as a vulnerability, it was then added to the set of vulnerabilities taken from the Mozilla Security Advisories.

First, we pulled down bugs which contain any of keywords in their descriptions into XML files to reduce loads generated by the crawling activities. We searched bugs with the status “Resolved”, “Verified”, “Closed”, and “Fixed”. After identifying potential vulnerabilities, we extracted detailed information of each bug using a secondary parser. To determine what files were involved with a particular bug, the authors inspected the linked bug entries attached to each security report linking to Mozilla’s Bugzilla database\(^4\). For each bug, we looked for an “diff”, or record of code changes, attached to the bug that had been positively reviewed by a security team member (indicated by the sec-approval+ tag), a release manager (indicated by approval-X or review+ tags). In most cases, we chose the most recently reviewed diff attached to the bug report. One exception is bugs labeled specifically with multiple diffs. These diffs would be labeled “Part 1”, “Part 2”, et cetera in the bug report attachments. If all the diffs had reviews from a security team member or a release manager, we included them in our vulnerability list.

When inspecting the individual diffs for each bugfix, we looked for C/C++ source code files in the diff. We specifically removed files from the "test" directory or with "test" in the name unless the bug specifically dealt with user access to test materials for Mozilla Firefox. In some cases, files may have moved or been refactored out of Mozilla Firefox since the bug report was issued. For files flagged in bug reports that were not found in source control, we looked to see if the lines of code modified was still in place in source control using grep. If the lines of code were found in another file, we changed the entry in the vulnerable code list to match the current file in source control. If it was not found, we discarded that file from the vulnerable code list.

From our mining effort, we identified 308 files for potential inclusion in our set of vulnerabilities. After classifying these files, we were left with 111 files involved with a vulnerability. The inter-rater reliability between the two authors when classifying vulnerable files was \(\kappa = 0.6\). We added these 111 files to the 247 files found on the Mozilla Security Advisories blog, resulting in a final list 358 source code files that were involved with a vulnerability. We discarded 14 source code files from our list and changed 8 source code files based on the movement of code to another source code file or the removal of code from the system. Finally, we created a list of the unique files in the list of 344 remaining files, resulting in 271 files with at least one vulnerability.

\(^4\)https://bugzilla.mozilla.org/
8.3.4.5 Prediction Model Comparison

For our replication, we used the SciKit Learn [Ped11] implementations of Gaussian Naive Bayes, Random Forest, Logistic Regression, and CART. for a fair comparison of each model. For Random Forest, we used a forest size of 100 and a maximum tree depth of 10 to prevent overfitting for the text mining model. While tuning parameters may result in prediction performance for certain models under certain conditions, we chose to use a consistent set of parameters across each model. We chose to use consistent parameters for two reasons: first, different parameters for the classifier for different models would result in a potentially unfair comparison of the models; second, practitioners using a model that we have trained without prior vulnerability data of their own will not have the opportunity to tune further, as they will not have an oracle to tune on.

For each model, we split the dataset into a training set containing 70% of the data and a testing set containing 30% of the data. We repeatedly sampled the original dataset 100 times. When computing precision, recall, and F-Scores for our models, we append the resultant prediction array to the rest of our runs to compare against the actual state of each test set. The result is the mean performance in terms of precision, recall and F-Score for each model.

Based on our replications of each model, results from Random Forest and Gaussian Naive Bayes outperformed Logistic Regression and CART in all of our scenarios. While the underperformance of these models does not mean that Logistic Regression and CART are irrelevant for VPMs, they were outclassed by Random Forest and Gaussian Naive Bayes for our case study.

8.3.5 Results

In this section, we present the results of our case study on vulnerability prediction models run against Mozilla Firefox.

8.3.5.1 VPM Comparison

**RQ6.1: What is the accuracy of Vulnerability Prediction Models based on software metrics, text mining, and crash dump stack traces on the same product?**

The results of each VPM, using a Random Forest Classifier and a Gaussian Naive Bayes Classifier, can be found in Table 8.3 and Table 8.4, respectively. We refer to the VPM generated by Zimmermann et al. as Software Metrics: Broad (SM:B) [Zim10]. We refer to the VPM generated by Shin et al. as Software Metrics: Churn and Complexity (SM:CC) [Shi11]. Box plots of the variation between runs for Gaussian Naive Bayes can be found in Figures 8.1 and 8.2. For the crashes VPM, the precision and recall numbers featured in the tables are the treatment of any source code file that crashes as a vulnerable file, as the SciKit Learn implementations of Random Forest and Naive Bayes had inconsistent results when a single feature was used as an input.
Table 8.3 Median Precision, Recall, and F1 score for each Vulnerability Prediction Model for Mozilla Firefox, using a Random Forest Classifier.

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theisen (Crashes)</td>
<td>0.05</td>
<td>0.86</td>
<td>0.09</td>
</tr>
<tr>
<td>Scandariato (Text Mining)</td>
<td>0.47</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>Shin (SM - Churn/Complexity)</td>
<td>0.46</td>
<td>0.04</td>
<td>0.07</td>
</tr>
<tr>
<td>Zimmermann (SM - Broad)</td>
<td>0.47</td>
<td>0.05</td>
<td>0.09</td>
</tr>
<tr>
<td>Crashes+Text Mining</td>
<td>0.53</td>
<td>0.04</td>
<td>0.07</td>
</tr>
<tr>
<td>Crashes+Software Metrics</td>
<td>0.43</td>
<td>0.05</td>
<td>0.09</td>
</tr>
<tr>
<td>Text Mining+Software Metrics</td>
<td>0.46</td>
<td>0.02</td>
<td>0.04</td>
</tr>
<tr>
<td>All</td>
<td>0.52</td>
<td>0.04</td>
<td>0.07</td>
</tr>
</tbody>
</table>

For Random Forest, accuracy is poor. While the precision numbers reported for Text Mining, SM:B, and SM:CC could be acceptable (0.47, 0.47, 0.46), the recall for each approach is low (0.02, 0.04, 0.05). In practical terms, the model is only predicting 4—8 vulnerable source code files, but that the model is making successful predictions almost half of the time.

We also collected feature importance data for Random Forest on the All Model, as shown in Table 8.5. Feature importance data was not available for the Naive Bayes classifier. Crashes were the strongest performing feature in the model with an importance of 0.016. Fan-out, or outgoing calls (CountOutput) from a specific source code file, was consistently the next best performing feature with an importance of 0.013. The string “nullptr” from the text mining model was next, with a feature importance of 0.012.

For Naive Bayes, accuracy worsens for the Text Mining approach while the SM:B and SM:CC approaches improve as compared to the Random Forest runs. For all three approaches, precision is noticeably worse in favor of larger recall. For text mining, the Naive Bayes classifier had a median precision of 0.01, versus 0.47 for Random Forest. Recall for Naive Bayes was 0.74, compared to 0.02 for the Random Forest classifier. Similar inverse relationships between precision and recall are observed for SM:B and SM:CC. The crash dump stack trace approach has the highest recall, but both the broad and the churn and complexity approaches have fewer false positives.

8.3.6 VPM Combinations

RQ6.2: What kinds of accuracy improvements are seen when combining features from different models?

The results of each combined VPM, using a Random Forest Classifier and a Gaussian Naive Bayes Classifier, can be found in Table 8.3 and Table 8.4, respectively. Box plots of the variation between runs for Gaussian Naive Bayes can be found in Figures 8.1 and 8.2. Because of the similarity
Table 8.4 Median Precision, Recall, and F1 score for each Vulnerability Prediction Model for Mozilla Firefox, using a Gaussian Naive Bayes Classifier.

<table>
<thead>
<tr>
<th>Model</th>
<th>Precision</th>
<th>Recall</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Theisen (Crashes)</td>
<td>0.05</td>
<td>0.86</td>
<td>0.09</td>
</tr>
<tr>
<td>Scandariato (Text Mining)</td>
<td>0.01</td>
<td>0.74</td>
<td>0.02</td>
</tr>
<tr>
<td>Shin (SM - Churn/Complexity)</td>
<td>0.12</td>
<td>0.46</td>
<td>0.19</td>
</tr>
<tr>
<td>Zimmermann (SM - Broad)</td>
<td>0.13</td>
<td>0.42</td>
<td>0.20</td>
</tr>
<tr>
<td>Crashes + Text Mining</td>
<td>0.22</td>
<td>0.34</td>
<td>0.27</td>
</tr>
<tr>
<td>Crashes + Software Metrics</td>
<td>0.18</td>
<td>0.33</td>
<td>0.23</td>
</tr>
<tr>
<td>Text Mining + Software Metrics</td>
<td>0.01</td>
<td>0.85</td>
<td>0.02</td>
</tr>
<tr>
<td>All</td>
<td>0.23</td>
<td>0.36</td>
<td>0.28</td>
</tr>
</tbody>
</table>

Table 8.5 The relative importance of the top 10 features used in the trees featured in the Random Forest Classifier in the “All” model. Quotes indicate a token from the text mining approach.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crashes</td>
<td>0.016</td>
</tr>
<tr>
<td>CountOutput</td>
<td>0.013</td>
</tr>
<tr>
<td>&quot;nullptr&quot;</td>
<td>0.012</td>
</tr>
<tr>
<td>CountClassDerived</td>
<td>0.012</td>
</tr>
<tr>
<td>CountDeclMethod</td>
<td>0.010</td>
</tr>
<tr>
<td>&quot;MOZ_ASSUME&quot;</td>
<td>0.010</td>
</tr>
<tr>
<td>Churn</td>
<td>0.010</td>
</tr>
<tr>
<td>CountClassBase</td>
<td>0.010</td>
</tr>
<tr>
<td>Cyclomatic</td>
<td>0.009</td>
</tr>
<tr>
<td>UniqueDevelopers</td>
<td>0.008</td>
</tr>
</tbody>
</table>

between the SM:B and SM:CC models, we combine all the metrics found in both models into a single “Software Metrics” model for combining software features with the other models.

For Random Forest, combining features from different approaches did not have a noticeable effect on the accuracy of the combined model. The best performing overall approach was the combination of software metrics and crash data, with a precision of 0.43 and recall of 0.05. In all cases, the inclusion of text mining data in a combined model for Random Forest resulted in an overall decrease in accuracy.

However, Naive Bayes showed improvement in accuracy with a combination of features from all three models. Using Naive Bayes, all four combined models showed an overall increase in accuracy. Individual losses in either precision or recall for a specific combined model was offset by a larger increase in the other statistic.
8.3.7 VPM Resource Usage

RQ3: What is the resource usage of Vulnerability Prediction Models based on software metrics, text mining, and crash dump stack traces on the same product?

Resource usage statistics for each individual VPM are presented in Table 8.6. All performance statistics were captured on a 2013 MacBook Pro using single threaded scripts.

The crash data, by definition, consisted of a single feature and was the smallest prediction model in our set. We capped our text mining feature set at 1000 for practicality, as collecting an unlimited number of features would have required hundreds of GB’s of space and would be impractical to classify with. Regardless of the restriction, the text mining features were orders of magnitude larger than the other models.

Collecting crash data took the longest amount of time for our case study, running over a period of 10 days with a rate limit enforced by the Mozilla API. Without a rate limit in place, we estimate that the mining of the 1,141,519 crashes included in our study would have taken approximately 10 hours, based on the rate of collection of crashes before the rate limit was reached. Using prefabricated tools for software metrics (Understand) and text mining (PyTokenizer) sped up the implementation process. Text mining took approximately 21 hours. The set of software metrics used for both the Shin and Zimmermann VPMs took 4.5 hours. Software metric data took the most memory to collect, using approximately 6GB of RAM at peak usage.

Software metrics also took up the least space on disk of the approaches, with the text mining data taking up 45.43GB in a .csv file. More efficient data formats may reduce the file size, but .csv was used for all of our datasets for easy comparison of file sizes.

Text mining, due to the sheer number of features, took significantly longer to run than all of the other approaches and permutations thereof. While the software metrics approaches took less than two minutes to run 100 times, the text mining approach took over 15 minutes. However, none of these runtimes would present a significant drawback in terms of practical usage unless the data was being rendered in real time for developers. The instant feedback from the crash dump stack trace approach could allow for more real time feedback.

8.4 Discussion

In this section, we discuss the implications of the above results for researchers and practitioners.

8.4.1 VPM Comparisons

In our results, the SM:B and SM:CC models had the best overall precision using both Random Forest and Naive Bayes, with the text mining approach equaling them using Random Forest. The crashes approach consistently had the best recall of the four models. Of note is that Random Forest
Figure 8.1 Precision values for each Vulnerability Prediction Model, along with the combinations of each model.
Table 8.6 Resource usage statistic for each Vulnerability Prediction Model on Mozilla Firefox.

<table>
<thead>
<tr>
<th></th>
<th>Data Collection</th>
<th>RAM</th>
<th>Runtime (RF)</th>
<th>Runtime (GNB)</th>
<th>Disk Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>RASA</td>
<td>10d (Limited)</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>3.43GB, 387KB compressed</td>
</tr>
<tr>
<td>Text Mining</td>
<td>20h36m</td>
<td>0.5GB</td>
<td>16m41s</td>
<td>14m5s</td>
<td>45.43GB, 58.9MB for top 1000</td>
</tr>
<tr>
<td>Software Metrics</td>
<td>4.5h</td>
<td>6GB</td>
<td>1m49s</td>
<td>17s</td>
<td>2MB</td>
</tr>
<tr>
<td>(Shin)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Software Metrics</td>
<td>4.5h</td>
<td>6GB</td>
<td>1m57s</td>
<td>15s</td>
<td>2MB</td>
</tr>
<tr>
<td>(Zimmermann)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

consistently produced stronger precision numbers, while Naive Bayes produced stronger recall numbers. As the cost of missing a vulnerability would typically outweigh the cost of reviewing a file that did not have a vulnerability, we would expect that models that prioritized recall results would be more practical for implementation. While our results have low precision numbers, our results still represent a significant reduction in the code to be analyzed.

8.4.2 VPM Combinations

Combining features from multiple models did not necessarily result in an increase in accuracy, and sometimes resulted in a degradation in performance. While we thought it was possible that the models could cover some of the same vulnerabilities, the fact that some features could prove to be distractors, or have a negative effect on accuracy, was unexpected. Further research into what specific properties of these features correlate with vulnerable code is needed for proper feature selection.

Using Naive Bayes, we found the best accuracy for a vulnerability prediction model run on a large (10,000+ source code files) software project. While better results have been seen for smaller projects using the text mining approach [Sca14], text mining had worse performance on a large one. While the results in Table 8.4 represent a step forward in vulnerability prediction, further work is needed to develop new features for input into classifiers, along with new analytic approaches for VPMs.

8.4.3 Feature Generation and Practical Usage

Our results indicate two approaches to improve performance of VPMs: better selection of classifier to use for prediction; and better features to input into said classifiers. The strength of crashes as a feature for vulnerability prediction, as shown by our results, indicates that features that incorporate
Figure 8.2 Recall values for each Vulnerability Prediction Model, along with the combinations of each model.
paths through software systems and the usage of those paths may have a strong relationship to vulnerabilities. To that end, dynamic metrics, which have been explored for a variety of subjects including productivity and time complexity [SS11], could be useful to map pathways that are actually executed. Recent work by Sharma et al. [SC15] indicates promise in the area of maintainability when it comes to the use of dynamic metrics generated at runtime for code measurement.

While models like the ones discussed in our study are useful for validating whether these features are consistent with the presence of vulnerabilities, providing a practical answer to the question "Where should I look first for vulnerabilities?" should be a question that we can answer. For the prioritization question, we turn to Table 8.5. For one of our best performing classifiers, crash data, the number of output calls from a file, and the string “nullptr” had the strongest effect on the decisions made in the trees for Random Forest. Based on these results, security teams should focus on these features (or in the case of nullptr, an equivalent reference to null) to make decisions on what code to review. These features are also intuitive: code that crashes more frequently is more likely to have errors; code with more outgoing calls is more likely to have a bad hand-off of data between methods; and code that uses null values explicitly runs the risk of setting something to null when it is inappropriate to do so or in an untested edge case.

### 8.5 Limitations

While we posit that availability of features is an issue when running VPMs, no study has been done to our knowledge to survey the software engineering community to find what software metrics are already generated by organizations. Understanding what metrics and features organizations have available already, even if they do not seem relevant to security at first glance. A list of metrics and features that organizations have readily available could drive both DPM and VPM research towards the use of these “low-lying” data sources, or provide an opportunity for researchers to recommend that organizations collect data that they currently consider to be of limited use.

The similarity of the features included in the Zimmermann and Shin VPMs is of particular note. The primary difference between the two approaches lies in how the models manipulate the input data. While Random Forests, linear discriminant analysis and Bayesian Network Modeling are all powerful techniques that were chosen because of the unique properties found in vulnerability prediction, exploring more prediction methods using the same input features may result in the discovery of other techniques well suited for VPMs.

Our study represents a comparison of different VPMs on the same software system. Firefox is a particularly large system for a prediction study on average; while the Zimmermann study was performed on Windows [Zim10], both Shin and Scandariato ran their approaches on systems of varying sizes, including PHPMyAdmin and Drupal [Shi11; Sca14]. For these systems both models performed better in a smaller environment. We were unable to find a small software system that
allowed for the collection of all the features needed for our study, as an appropriate software system had to be written in C/C++, have public source code, have public vulnerability data, and have a public repository of crashes. Further replications of our study would be useful for comparing these models, but publicly available data is a significant limitation.

A single missing vulnerability can be catastrophic for an organization. Because no published VPM to our knowledge claims 100% coverage of historical vulnerabilities, practitioners may decide not to take the results of such models into consideration. One of our observations while explaining RASA to practitioners is that framing the approach as a prioritization scheme rather than a prediction model improved practitioner reaction to the tool. Further research into the reaction of practitioners to vulnerability prediction models when framed as absolute measures versus prioritization methods could help determine what barriers are preventing prediction model adoption in industry.
We address the following research question with this study:

**Attack Surface Comparison (RQ7):** How does risk-based attack surface approximation compare with the state of the art of attack surface approximation in terms of computation time, size of the resultant attack surface, and presentation of feedback?

### 9.1 Motivation

Other researchers have developed approaches that use the attack surface concept. Munaiah et al. [MM16] developed a technique for determining the attack surface of a target system via random walks on call graphs of the target system. Younis et al. [You14] developed an approach to measure the risk of specific vulnerabilities based on their determination of the attack surface of a system. Understanding the benefits and drawbacks of each of these approaches compared to RASA is important for determining the cases in which each approach should be used. Metrics used for comparison include the size of the resultant attack surface and the percentage of vulnerabilities covered by each approach.
9.2 Munaiah Attack Surface

The attack surface measurement system designed by Munaiah et al. [MM16] makes use of GNU CFlow\(^1\) to generate a call graph of the targeted software system. The call graph of a system is used to generate a *Risky Walk* of the software system. The *Risky Walk* is based on the idea of an attacker exploring or probing a system for potential vulnerabilities. The attacker is assumed to have no knowledge of the paths and flows through a system, and so explores the system similarly to a web user exploring the Internet. Therefore, Munaiah et al. use the PageRank algorithm [Pag99] to simulate attacker behavior in exploring the software system. Both the call graph and the *Risky Walk* are used to generate a series of metrics profiling the attack surface of the software system:

- **Entry Points**: Places where data is passed to the system from an outside entity.
- **Exit Points**: Places where data is sent to outside entities form the software system.
- **Dangerous Points**: Functions in the software system that attackers are likely to target with attacks, based on PageRank metrics.
- **Proximity Metrics**: A measurement of the distance between entry or exit points and dangerous points in the software system.
- **Risky Walks**: A probabilistic measurement of the behavior of a typical attacker using random executions of the system with known malicious inputs.

Munaiah et al. have made a replication package for consuming Cflow-generated call graphs on Github\(^2\). In summary, the Munaiah approach not only models data flows through a software system from the call graph of the system, but also attacker behavior related to potentially dangerous functions in a system through the *Risky Walk*. The approach assumes that attackers have some insider knowledge of the software system to determine the dangerous points in the system for computing the risky walk metric.

GNU CFlow is restricted to use on programs written in C and C++. GNU CFlow therefore cannot build complete call graphs on software systems written wholly or partially in any other languages. As many systems are written in several languages, this is a significant restriction. Another limitation of the use of GNU CFlow to measure the attack surface of software is a restriction on the size of the software system that can be analyzed using the tool. We attempted to use Cflow on Mozilla Firefox in a variety of different configurations, but were unable to capture a set of flows for the software system. The Fedora and Ubuntu operating systems were also run with Cflow with negative results. Our result suggests that attack surface measurement systems based on Cflow will fail to return results.

---

\(^1\)http://www.gnu.org/software/cflow
\(^2\)https://github.comandymeneelyattack-surface-metrics

97
on larger software systems. Other tools for generating call graphs exist, such as Egypt, but these tools (including Egypt) only provide visual representations of call graphs or other non-parseable formats. If another tool that generated text output similar to CFlow could be run on large systems, the Munaiah approach may be able to be properly replicated on those large systems.

9.3 Younis Attack Surface

The attack surface measurement approach proposed by Younis et al. [You14] is a subset of the attack surface measurement proposed by Munaiah et al. The Younis et al. approach also uses GNU CFlow, as mentioned in Section 9.2. However, the Younis approach only determines the entry and exit points described by Munaiah et al. in their approach. Younis then evaluates the resultant attack surface by whether vulnerabilities are found along the paths from entry to exit points, as determined by CFlow. The Munaiah approach identifies entry and exit points as well, but offers the additional step of determining the proximity of vulnerabilities to these paths.

Based on our observation, the Munaiah approach can be considered a superset of the Younis approach. Any code artifact reported by Munaiah as having any value of the proximity metric greater than zero would have been reported as part of the attack surface by Younis. We saw no advantage to replicating the Younis approach in addition to Munaiah based on these observations, as 1) to our knowledge, the Younis approach has no public reproduction package; and 2) we would be replicating work already available in the Munaiah reproduction package. To our knowledge, no direct comparison exists between the Younis and Munaiah approach. We continued our analysis strictly comparing the RASA approach and the Munaiah approach.

9.4 Methodology

To compare the RASA and Munaiah approach, we replicate both approaches on a software system where both approaches can be performed. For RASA, the software system must have available crash dump stack trace information with binary, file, and/or function information available for analysis. A mapping of the code information to source control is also necessary. For the Munaiah approach, the GNU CFlow tool must be able to be successfully run against the software.

After selecting a software system, we collect the following metrics for each approach:

- **Data Collection Time**: The amount of time it took to collect the data necessary to run the approach.

- **Data Size**: The amount of disk space the approach's data takes up.

• **Attack Surface Size:** The size of the resultant attack surface, as measured by the approach.

• **Precision:** The likelihood for a particular code artifact to have a vulnerability if it is on the attack surface measured by the approach, and

• **Recall:** The likelihood that a particular vulnerability is included on the attack surface measured by the approach.

Once we have collected these metrics, we compare the two approaches across these five metrics.

### 9.5 Case Study - Wireshark

To compare attack surface measurement approaches, we first needed to determine a software system or set of software systems to use as the basis for comparison. We used the following criteria to exclude software systems from our study:

• The system must not be written and documented in a language other than English.

We used the following inclusion criteria to select software systems for possible inclusion into our study:

• The system must have publicly available source code.

• Assuming access to source code, the GNU CFlow tool must be able to be run successfully against the system.

• The product must have publicly available vulnerability data.

• The product must have publicly available crash dump stack trace data from end users.

We looked at several different software systems for potential inclusion in our study. The following systems were considered but disqualified:

• **Windows:** While crash data was available for Windows, source code was not available, so GNU CFlow could not be run against Windows.

• **Firefox:** Crash data was available from previous studies [The17; The18b]. However, attempts to run GNU CFlow resulted in incomplete call graphs of the system that did not pass manual inspection for possible paths. As an example: the call graph generated by CFlow was 50% larger on disk than the call graph generated for wireshark, but Firefox has over 10 times as much source code as Wireshark.
• **Ubuntu**: As a larger software system than Firefox, GNU CFlow also failed to generate a correct call graph. In addition, Ubuntu restricts access to crash data on an as-needed bases for developers of applications for the Ubuntu operating system.

• **Fedora**: As a larger software system than Firefox, GNU CFlow also failed to generate a correct call graph.

After these failures, we found that Wireshark had data that would support both attack surface measurement systems. Crash data was available via Wireshark's Bugzilla tracker when searching for the term "Crash" \(^4\). Source code was freely available, as Wireshark is open source. Finally, Munaiah et al. had used Wireshark for their initial study of their approach [MM16]. Their previous success gave strong confidence that the approach could be replicated.

We were able to identify 19 vulnerabilities that had code fixes associated with them. These 11 vulnerabilities were flagged with the “security” flag, and had at least one diff associated with them. We parsed each diff attached to each vulnerability, and extracted the source code files changed as part of the diff that fixed the vulnerability. A total of 11 unique files were associated with at least one vulnerability, with one files being part of multiple vulnerabilities in several cases.

### 9.6 Results and Discussion

After collecting our crash data and Wireshark vulnerabilities, we compare the correlation between crashing code and vulnerable code per the methodology in Chapter 4. We found 38 crashes in Wireshark's Bugzilla database that had file and function data available in the stack trace associated with the crash. Of the 3,580 source code files in Wireshark, 65 of them were on the 38 crashes from the system. The attack surface generated by RASA for Wireshark resulted in a precision of 0.05 and a recall of 0.27. By comparison, the Munaiah approach featured precision of 0.11 and 0.76 for locating vulnerable files. The RASA attack surface covered less vulnerabilities with worse accuracy.

From a resource usage perspective, data collection took approximately four hours for the manual collection of crashes. The crash data took less than 1MB of space on disc. Generating a control flow graph for Wireshark took 45 minutes on a 2013 MacBook Pro, with the resultant Cflow size being 60MB. Calculating precision and recall took trivial amounts of time for both approaches. In addition to these comparison, we also looked at the sizes of the attack surfaces generated by each approach. The resultant attack surface generated by RASA covered 167 unique functions in Wireshark. The Munaiah approach covered 2308 functions, with 48 entry points and 161 exit points. The RASA attack surface was significantly smaller, which may explain the differences in results in terms of precision and recall.

\(^4\)https://bugs.wireshark.org/bugzilla/buglist.cgi?quicksearch=crash
9.7 Limitations

Our study was performed on the Wireshark software system, which does not have a central repository of crashes from customer data. The lack of central repository means that crashes were only usable if 1) they were attached to a bug report; and 2) the system the crash is fun also had symbol data installed as part of the Wireshark installation package. Symbols are not included in a default Wireshark installation, meaning only users who explicitly wanted to install them to help with debugging efforts will have them. Lack of symbol data meant that approximately 66% of the crashes in Wireshark's Bugzilla database were unable to be used in our study. Several orders of magnitude less crash data was included as part of our study, which likely had a significant effect on the results of RASA. The lack of overlap between software systems that RASA and the Munaiah approach can be run on means that it is difficult to draw meaningful conclusions about the predictive power of each approach. The fact that both approaches cover distinctly different cases of software may be an advantage, as more teams have attack surface approaches available.

Overall, the Munaiah approach [MM16] outperformed RASA in terms of vulnerability coverage, but the results are inconclusive. Our study confirmed that RASA needs quality crash data to accurately approximate the attack surface of software, and that RASA will struggle to perform well when it does not have quality crash data. We therefore conclude that an automatic crash reporting system where symbol data can be applied immediately is a requirement for systems wishing to implement RASA.
In this chapter, we summarize the contributions of the previous studies and discuss potential future work in the area of RASA and attack surface measurement. We also discuss limitations of RASA.

### 10.1 Contributions

We make the following contributions as part of this work:

We have developed an approach for approximating the attack surface of a software system using crash dump stack trace metrics as an empirical metric for vulnerable code called Risk-Based Attack Surface Approximation. RASA has been validated as an effective means for approximating the attack surface of several different software systems and shown to work with a variety of different magnitudes of crash dump stack trace data. We also provide the framework for running RASA against other systems via scripts for parsing crash dump stack traces. Additionally, we have developed a series of additional metrics based on the concepts of frequency of crashes, complexity of graphs generated by stack traces, and the boundary of a software system to further improve prioritization efforts.

We have compared RASA against the state of the art in VPMs and as an input into other VPMs in an attempt to improve performance of VPMs against the state of the art. Adding crash dump stack trace data to existing VPMs based on software metrics and text mining resulted in a threelfold

\[1\text{https://github.com/theisencrstack-miner}\]
increase in accuracy over any one VPM.

We have profiled vulnerabilities used as an oracle for VPMs, with analysis of the distribution of various classifications based on CWE and severity scores given by developers to bugs in Mozilla Firefox. A better understanding of classifications and severities of vulnerabilities could guide future efforts in improving VPMs.

We also compared RASA against another approach for measuring attack surfaces in Munaiah et al.’s GNU CFlow based approach [MM16]. RASA has specific advantages over the CFlow based approach in that it works on multiple languages and on much larger software systems that CFlow cannot analyze.

10.2 Summary

The studies in this document have explored the feasibility [The15c] and applicability [The17] of the RASA approach, as well as comparing RASA to different techniques [The18b].

RASA has been run against several different systems, including Windows, Firefox, and Wireshark. For the larger software systems, RASA provides a cost-effective, resource efficient way of determining the attack surface of a software system. For smaller systems such as Wireshark, the quality of the resultant attack surface measurement is dependent on the quality of crash data available. Crash data from bug reports, rather than a datastore specifically designed for crash data, resulted in a less complete attack surface measurement. RASA has been run successfully at the binary and file levels of granularity for source code. Files are a more efficient unit of measure for locating vulnerabilities as compared to binaries.

RASA optimizes well for recall when treated as a prediction model, or stated alternatively, performs well when the goal is to cover as many vulnerabilities as possible. The underlying crash metric of RASA also augments existing VPM well, resulting in state of the art performance in vulnerability prediction when combined with text mining and software metrics data.

RASA has specific advantages for practitioners looking for an approach for prioritizing security effort with minimal effort. Because of the low cost in terms of technical expertise and data requirements for RASA, teams of varying sizes can implement RASA on their own systems. Teams should be aware of several restrictions during the implementation of RASA. One, quality crash dump stack trace data is a requirement, and may require teams to collect crash data via automated functionality. Frameworks for automated crash collection are available, as Mozilla Crash Reports is open source. Our comparison against other attack surface measurement approaches, however, validated that a curated crash dataset is a requirement for effective measurement using RASA.
10.3 Future Work

In this section, we describe several different approaches for further expanding on our work.

10.3.1 Crash Data Quality

Chapter 9 describes the result of running RASA on Wireshark. Our study highlighted some limitations of the RASA approach, specifically on smaller software systems. While prediction of vulnerable code is possible on smaller systems as outlined in Chapter 5, the quality of the crash data is important. In the case of Wireshark, available crash data was limited as symbol data is not available by default, and no central repository of crashes exists for Wireshark.

For teams looking to implement RASA on their own system, the quality of the crash data they have must be considered. If the organization does not ship debugging symbols with their software by default, they will need to be able to apply symbol data post crash at a central repository, such as in the approaches by Microsoft and Mozilla.

10.3.2 Dynamic Software Metrics

The software metrics used in Chapter 8 represent a class of metrics known as static software metrics. Static software metrics are defined by the fact that they do not change no matter the execution profile of the software. For example, the number of lines of code that are contained in a source code file does not change from user to user when that user executes software. Previous work with DPMs indicated that dynamic, or runtime metrics for software offered no significant advantage over static metrics in terms of defect prediction, despite providing information about how the software is actually used. Chhabra et al. [CG10] surveyed the differences between dynamic and static metrics, and found that while dynamic metrics were more difficult to obtain, they provided unique information that static metrics do not, such as behavioral data about users, more precise measurements of software in practice compared to static metrics, and an ability to identify and deal with object-oriented features as well as dead code.

The behavioral metrics mentioned by Chhabra et al. [CG10] would be of particular interest to security practitioners and researchers. Security problems are partially behavior problems; they are examples of a system performing actions they are capable of performing, but were not designed to perform, such as unauthorized exfiltration of personal data. While dynamic metrics offered minimal advantages over static metrics in the defect prediction space, they may offer advantages in the vulnerability prediction space thanks to these contextual metrics.
10.3.3 Oracles for Vulnerability Studies

One consequence of our work is a closer examination of oracles for vulnerability prediction or attack surface modeling studies. Previous studies in the vulnerability prediction and attack surface modeling space have used historical vulnerabilities as an oracle representing known vulnerable code [The15c; The17; Zim10; Shi11; Sca14]. The use of historical vulnerabilities as an oracle for vulnerable code follows from defect prediction research, which uses historical defect data as an oracle.

However, the differences between vulnerability datasets and defect datasets raises the question on whether the two approaches should be evaluated by the same metric. A study by Kononenko et al. found that the chance for an individual file or function to have a defect can be as high at 50% [Kon16]. Our Mozilla Firefox vulnerability dataset has a vulnerability rate of less than 1%. As historically defective code is significantly more prevalent than historically vulnerable code, the data imbalance problem means that vulnerability prediction work using a historical oracle may never rise to the predictive power of defect prediction models.

Part of our evaluation process included time spent talking to industry professionals about the specific results of the RASA studies in this document. During these discussions, a few practitioners raised the concern that historical vulnerability datasets have a discovery problem. To quote one practitioner:

“If my team has always known that the ‘Example’ function could have a vulnerability introduced by a developer, we’re extra careful to check that function and the associated functionality for a possible vulnerability whenever it’s changed. If we do our due diligence and keep a vulnerability from being shipped to users in that function, then a prediction model based on observed vulnerabilities is never going to include the ‘Example’ function.”

The potential bias against frequently inspected areas of the codebase has several negative consequences. First, for the experienced practitioner above, the fact that the hypothesized vulnerability prediction approach misses a key function like the above ‘Example’ function lowers their trust in the tool. If a vulnerability prediction tool misses the “obvious” things, why should they trust the rest of the results? For less experienced professionals or teams just starting secure engineering processes, missing the obvious could have immediate critical consequences for the team. If you rely on these prediction models to build your basic understanding of the security concerns in a system, then the team may not realize what parts of their systems have critical security issues until those issues have been exploited. Understanding the scarcity of vulnerability data is critical for interpreting the results of VPM studies, as the accuracy of VPMs is effected by the scarcity.
10.3.4 Framing Results

Based on these observations, one important branch of research may be on the oracles used to evaluate vulnerability prediction models. One possibility raised by practitioners was modeling impact of vulnerabilities if they occur in a specific part of the codebase. The practice of Threat Modeling is widely used in industry to determine potential threats to a software system. Applying Threat Modeling to source code would mean highlighting areas of the codebase where a vulnerability would cause the most damage. For example, one practitioner described an engineering policy that mandated security review and testing if code is modified or written that manipulates PII. While a vulnerability may not have occurred in that place before, a vulnerability in such code would be costly for their organization. Pairing data flow and manipulation information with historical vulnerability data could highlight areas that are critical and have seen issues in the past, providing a more stratified priority queue for reviewing and testing parts of the system. For example, a function that has had vulnerabilities in the past and handles critically sensitive information could be reviewed and tested after every change, while code that has had previous vulnerabilities but handles less sensitive data could be tested less often. Introducing impact scores and tuning vulnerability prediction models around these scores may increase practitioner trust in vulnerability prediction toolsets.

Practitioners also raised concerns about cases where a prediction model misses a vulnerability. A single missing vulnerability can be catastrophic for an organization. Because no published model to our knowledge claims to catch 100% of historical vulnerabilities, practitioners may decide not to take the results of such models into consideration. One of our casual observations when explaining RASA to practitioners is that framing the approach as a prioritization scheme rather than a prediction model improved practitioner reaction to the tool. For one practitioner we spoke to, a precision model implied “only look at these functions or files,” while a prioritization scheme suggested them to look at those functions or files first. Further research into the reaction of practitioners to vulnerability prediction models when framed as absolute measures versus prioritization methods could help determine what barriers are preventing prediction model adoption in industry.


[Coh68] Cohen, J. “Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit.” Psychological bulletin 70.4 (1968), p. 213.


