BP: Profiling Vulnerabilities on the Attack Surface

Christopher Theisen, Hyunwoo Sohn, Dawson Tripp, and Laurie Williams
Computer Science
North Carolina State University
Raleigh, North Carolina 27695
Email: crtheise@ncsu.edu, hsohn3@ncsu.edu, dawsonrtripp@gmail.com, lawilli3@ncsu.edu

Abstract—**Security practitioners use the attack surface of software systems to prioritize areas of systems to test and analyze. To date, approaches for predicting which code artifacts are vulnerable have utilized a binary classification of code as vulnerable or not vulnerable. To better understand the strengths and weaknesses of vulnerability prediction approaches, vulnerability datasets with classification and severity data are needed. *The goal of this paper is to help researchers and practitioners make security effort prioritization decisions by evaluating which classifications and severities of vulnerabilities are on an attack surface approximated using crash dump stack traces*. In this work, we use crash dump stack traces to approximate the attack surface of Mozilla Firefox. We then generate a dataset of 271 vulnerable files in Firefox, classified using the Common Weakness Enumeration (CWE) system. We use these files as an oracle for the evaluation of the attack surface generated using crash data. In the Firefox vulnerability dataset, 14 different classifications of vulnerabilities appeared at least once. In our study, 85.3% of vulnerable files were on the attack surface generated using crash data. We found no difference between the severity of vulnerabilities found on the attack surface generated using crash data and vulnerabilities not occurring on the attack surface. Additionally, we discuss lessons learned during the development of this vulnerability dataset.**

I. Introduction

Security professionals use the attack surface of a software system to determine specific parts of a software system to target with security testing and review efforts. The Open Web Application Security Project (OWASP) defines the attack surface as follows [1]: The sum of all paths for data/commands into and out of the application, the code that protects these paths, valuable data used in the application (including secrets and keys, intellectual property, critical business data, and personal data and PII), and the code that protects data. Howard et al. defined the attack surface of a system along three different dimensions: targets and enablers, channels and protocols, and access rights [1].

Determining the complete attack surface of a system is computationally difficult [2]. One approach for approximating the attack surface of a software system is called Risk-Based Attack Surface Approximation (RASA) [3], [4]. 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 [5]. Some vulnerabilities may initially be declared as low severity, but are later determined to have wider consequences. One example of this is the default credentials vulnerability in Internet of Things (IoT) devices that led to the Mirai botnet [6].

RASA can be described as a vulnerability prediction model (VPM). VPMs are used to predict source code as either containing a vulnerability or not containing a vulnerability. Evaluating which classifications and severities of vulnerabilities are covered by a VPM is important to understand the benefits and limitations of approaches for prioritizing security effort. *The goal of this paper is to help researchers and practitioners make security effort prioritization decisions by evaluating which classifications and severities of vulnerabilities are on an attack surface approximated using crash dump stack traces*. 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 influence the next steps practitioners take to remedy those issues. In this paper, we will answer the following research questions:

RQ1: What vulnerability classifications are more likely to appear on a risk-based attack surface approximation compared to the entire set of code in a system?
RQ2: What severity of vulnerabilities are more likely to appear on a risk-based attack surface approximation compared to the entire set of code in a system?

In our study, we compare whether vulnerabilities of specific types and severities appear on, or are covered by, the approximated attack surface given by RASA. We analyzed vulnerabilities and source code from three different versions of Firefox: 57.0, 58.0b, and 59.0a. We obtained externally-reported vulnerabilities from the Mozilla organization's public reporting. We augmented the vulnerabilities with defects mined from Mozilla's Bugzilla repository of bugs, identifying defects with security consequences using a keyword search.

[1]https://www.owasp.org/index.php/Attack_Surface_Analysis_Cheat_Sheet

We then collected crash dump stack traces from Mozilla Crash Reports to generate the approximated attack surface from RASA.

After collecting a set of vulnerabilities to evaluate, we determine which vulnerabilities were on or off the attack surface approximated by RASA. We then determine whether specific classifications and/or severities of vulnerabilities are more or less likely to be covered by RASA. If RASA has specific strengths or weaknesses, these strengths and weaknesses can be augmented by other security effort prioritization approaches.

We contribute the following as part of this work:

- An analysis of how vulnerabilities of different classifications and severities are covered by the Risk-Based Attack Surface Approximation; and
- A process for mining vulnerability datasets from bug repositories for other software systems.

The rest of this paper is organized as follows. Section II describes related work to our project, Section III describes the process for generating an attack surface from RASA, Section IV describes our research methodology, Section V describes our data collection approach for Mozilla Firefox, Section VI describes our results, Section VII discusses the conclusions drawn from our results, and Section VIII describes the limitations of our work and potential future avenues of research.

II. Related Work

In this section, we present related research to our study.

A. Crash Dump Stack Trace Analysis

Analyzing crash dump stack traces to build profiles of software or identify defective code continues to be an active area of research. Castelluccio et al. [7] developed an approach for grouping crash dumps together for easier analysis on Mozilla Firefox, which is deployed in Mozilla's crash reporting system. Bianci et al. [8] use crash data to generate tests designed to reproduce the failures the data represents.

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. [9] built Retracer, an approach for reverse engineering the execution path from crash data. Chen et al. [10] developed an approach called Star, another approach for reverse engineering crashes using symbolic execution. In the space of security specific work on crash data, Credal, developed by Xu et al. [11] focuses on localizing memory corruption errors from memory dumps. The volume of recent work in the area of crash dump reproduction speaks to the value of this data when attempting to triage defects and vulnerabilities.

B. Vulnerability Prediction and Prioritization

Vulnerability Prediction Models (VPMs) are a subset of Defect Prediction Models (DPMs) focused specifically on vulnerabilities in software. The analysis of and improvement of DPMs using new datasets, new features, and new approaches for analysis is a popular subject of top software engineering conferences [12]. Text mining was explored for vulnerability prediction by Hovsepyan and Scandariato et al. [13], [14]. Shin et al. [15] and Zimmermann et al. [16] explored the use of software metrics like code churn and code complexity for vulnerability prediction. Theisen et al. [3] developed RASA, reproduced in this work, which is used for security effort prioritization. However, none of these approaches consider vulnerability classification or severity in their analysis.

C. Classifying Defects and Vulnerabilities

The classification of defects and vulnerabilities has a long history in the research community. Chillarege et al. developed a technique called Orthogonal Defect Classification (ODC) to develop classifications of defects [17]. Podgurski et al. [18] later developed an automated approach for classifying defects. In this work, we use existing classification data for defects generated by the Mozilla team to determine if any of those defects could also be considered as vulnerabilities.

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 [19]. Li et al. profiled defect characteristics in open source projects [20]. Khalid et al. found that complaints about software functionality were the most common complaint for mobile applications [21]. However, classification schemes for vulnerabilities is not as common. In vulnerability classification literature, researchers have mostly focused on whether code can be classified as vulnerable or not [3], [4], [22], [15], [16], [14].

Austin et al. [23] performed a comparison of vulnerability detection techniques and found that a single approach was not sufficient for vulnerability coverage. This work informs the need for a vulnerability dataset that not only considers a binary classification of vulnerabilities, but also a classification process for vulnerabilities. Bettenburg et al. provided a blueprint for the content of useful bug reports [24].

D. Attack Surface

Manadhata et al. [2] provided one of the earliest measures of attack surfaces in software systems, focusing on entry and exit points. Theisen et al. used crash dump stack traces to predict potentially vulnerable code [3], [4]. We replicate their approach, called Risk-Based Attack Surface Approximation (RASA), as part of this work. Vulnerability data collected as part of this work is shared with another report submitted to another conference [25]. Zhang et al. [26] used package dependencies to determine the attack surface of package based systems. In related domains such as networks and power grids, researcher look for ways to limit the exposure of critical infrastructure using the attack surface concept [27].

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. [28] focused on minimizing the attack surface of databases. Zhang et al. [29] developed an approach for reducing the attack surface of an operating system kernel. Kantola et al. [30] examined message handling between applications on the Android operating system in order to reduce the possibility of attack. While many approaches exist for approximating or reducing the attack surface of software systems, the generation of a ground truth attack surface for large software systems remains an open question.

III. Risk-Based Attack Surface Approximation

In this section, we describe Risk-Based Attack Surface Approximation (RASA) [3], [4], an approach for predicting what source code in a software system might have exploitable vulnerabilities.

Crash dump stack traces were used by Theisen et al. as a proxy for potentially vulnerable code. [3]. Crashing code has several properties that it shares with vulnerabilities. First, crashing code and vulnerable code can both be expressed as unexpected data flow and handling errors. In a crash, these data flow and handling errors result in a specific form of unintended behavior; a crash of the target system. Many vulnerabilities consist of data flow and handling errors, with the consequence of the violation of security properties such as confidentiality, integrity, and availability. The core assumption of the Theisen et al. approach is that code that has previously crashed is likely to have additional, separate data flow and handling issues. While previous research has shown that defects tend to correlate with one another, Camilo et al. found that this does not extend to vulnerabilities, necessitating new approaches for determining what code should be inspected for vulnerabilities [31].

Generating RASA For a software system consists of several data collection and analysis steps. A visualization of this process is found in Figure 1. First, we collect a set of crash dump stack traces from the software system. We find the crashing thread in the crash dump, along with the associated stack trace for that thread at the time of the crash. From the stack trace, we identify the individual code artifacts (binaries, files, functions) that occur on each frame (or line), of the stack trace in the crashing thread. Next, code artifacts from the crash data are compared against the contents of source control for the software system, with the goal of developing a one-to-one mapping of the code in the crash data to source control. Code artifact names can be reused or overloaded during the development process. For example, multiple source code file can be named "foo.cpp" in different directories of the system. Function name reuse is also common. The resultant mapping of crashing code to source control provides a list of code in source control that has crashed at least once. We treat this as an approximated version of the attack surface of of the software system, or the RASA of the software system. A visualization of the approximated attack surface of a software system can be found in Figure 2.

After generating RASA for a software system, we use security vulnerability data as an oracle for evaluating how well RASA covers vulnerable code.. Code artifacts with at least one vulnerability are considered vulnerable code, while code artifacts with no vulnerabilities are considered non-vulnerable code. We then calculate two metrics to evaluate the performance of RASA: Code Coverage (CC) and Vulnerability Coverage (VC) [4].

CC is the percentage of artifacts found on crash dump stack traces relative to all artifacts, as shown in via Equation 1 [4]:

$$CC = \frac{\#\ artifacts\ in\ crashes}{\#\ artifacts\ in\ the\ system} \tag{1}$$

VC is the percentage of artifacts found on crash dump stack traces with at least one or more security vulnerabilities relative to all artifacts with one or more vulnerabilities, as computed in Equation 2 [4]:

$$VC = \frac{\#\ artifacts\ with\ at\ least\ one\ vuln.\ in\ crashes}{\#\ artifacts\ with\ at\ least\ one\ vuln.} \tag{2}$$

In one RASA [3] using Windows data, 48.4% of binaries appeared on at least one crash dump stack trace (CC=48.4%), while 94.8% of post-release vulnerabilities (VC=94.8%) were found in that 48.4% subset of the binaries. Based on this result, Security professionals can prioritize their efforts on source code that has been seen in crash dump stack traces, as vulnerabilities are found to be more dense in those locations.

Another RASA study using Mozilla Firefox data [4] ran the process at the source code file level, rather than the binary level. In the Firefox study, the researchers found that randomly sampling stack traces down to 10% of the available crash data did not have an appreciable impact on RASA's ability to predict where exploitable vulnerabilities would be in either Firefox or Windows [4]. The process for approximating vulnerable code was run 10 times with 10% of the crashes, 20% of the crashes, et cetera for both systems. Between runs at the same sampling level, variation of vulnerability coverage of less than 1% was observed. In addition, only 3% (or fewer than eight) fewer vulnerabilities were covered using an order of magnitude less data (10% of the crashes versus 100% of the crashes in the collected dataset). Therefore, smaller organizations with significantly less crash data may see practical results similar to the results of RASA on Firefox and Windows for their own software systems.

IV. 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.

A. 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 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

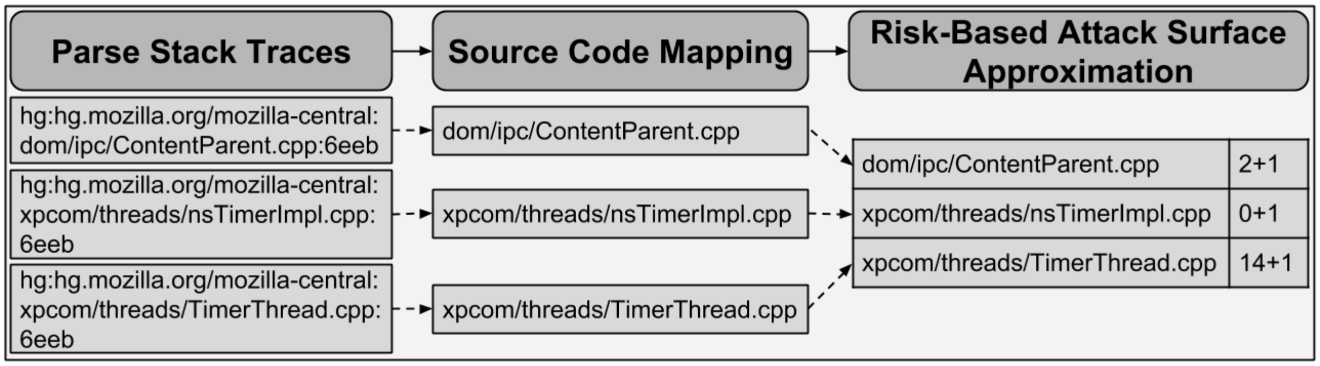
Parse Stack Traces
Source Code Mapping
Risk-Based Attack Surface Approximation
hg:hg.mozilla.org/mozilla-central: dom/ipc/ContentParent.cpp:6eeb
hg:hg.mozilla.org/mozilla-central: xpcom/threads/nsTimerImpl.cpp: 6eeb
hg:hg.mozilla.org/mozilla-central: xpcom/threads/TimerThread.cpp: 6eeb
dom/ipc/ContentParent.cpp
xpcom/threads/nsTimerImpl.cpp
xpcom/threads/TimerThread.cpp
dom/ipc/ContentParent.cpp 2+1
xpcom/threads/nsTimerImpl.cpp 0+1
xpcom/threads/TimerThread.cpp 14+1

Fig. 1. A visualization of the process used to generate a Risk-Based Attack Surface Approximation (RASA) for a software system, with a running example. Each occurrence of a file in a stack trace adds an additional count to the resultant attack surface.

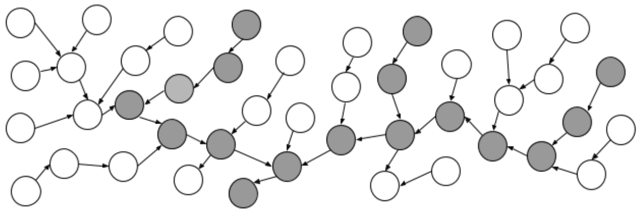

Fig. 2. A graph visualization of the attack surface generated by RASA. Individual nodes on the graph are source code artifacts. Edges indicate calls from one artifact to another. Grey circles are code that crashed at least once.

for security issues, such as "security" or "vulnerability". In addition to collecting explicitly defined vulnerabilities, we also check the rest of the bugs in the bug tracking system for more potential vulnerabilities, as research by Herzig et al. [32] 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 V.

B. Classifying Vulnerabilities

After identifying vulnerabilities for inclusion in our dataset, two raters then classify each vulnerability using an existing vulnerability classification scheme. We use the Common Weakness Enumeration (CWE) set of most commonly seen weaknesses in software [2]. We have added the "Other" classification to the list of classifications to cover the case where a vulnerability does not fall into a preexisting category. In the event that a significant amount of "Other" vulnerabilities are identified, further stratification may be needed. From CWE, we use the following 24 classifications of vulnerabilities:

- 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 dataset. 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 [33]. Cohen's Kappa provides a quantita-

[2]http://cwe.mitre.org/data/definitions/1003.html

tive measure of the agreement between classifiers or raters. Cohen's Kappa is useful for this process 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 for sorting vulnerabilities into classifications once the vulnerabilities have been identified.

C. *Vulnerability Severity*

We use system-specific severity ratings for the vulnerability dataset, rather than using a global measure like the Common Weakness Scoring System (CWSS). A system-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 system-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.

D. *Risk-Based Attack Surface Approximation*

We evaluate RASA, as described in Section III, in terms of RASA's ability to cover vulnerabilities of different classifications and severities. For RASA, crash dump stack traces from the system under analysis are collected and stored. The individual code artifacts, such as binaries, files, or functions, that appear on the crashing thread in the the crash dump stack traces are saved. If a binary, file, or function appears on at least one of these crashing threads, RASA declares that code to be on the attack surface of the software system.

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 in the previous sections 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 V.

V. Case Study

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

A. *Vulnerability Data*

Mozilla Firefox vulnerability data was first collected from Mozilla Foundation Security Advisories blog [34]. Vulnerabilities were collected by the authors. For each bug, we recorded the following information:

TABLE I
The list of keywords used to collect potential unlabeled vulnerabilities from Mozilla's Bugzilla database on Mozilla Firefox.

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

- *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.
- *Bugzilla Entry:* A link to the associated Bugzilla entry in the Mozilla repository.
- *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 Bugzilla repository [3] for bugs that were not reported publicly on the Mozilla Foundation Security Advisories blog. Some bugs were not reported publicly, but exhibited characteristics of a vulnerability. We used a list of keywords determined by Shin et al. [15] to mine these potential vulnerabilities. The list of keywords used for mining potential vulnerabilities is found in Table I. After collecting bugs that matched one or more of these keywords 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. We record the same vulnerability information for vulnerabilities mined from Bugzilla, with the exception of labeling the *Mozilla Security Report* field as Not Applicable.

To link source code files in Mozilla Firefox with a vulnerability, the authors inspected the *Bugzilla Entry* for each identified vulnerability in Mozilla's Bugzilla database. 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). In some cases, multiple diffs were included as part of the fix for a vulnerabilities. These diffs would be labeled "Part 1", "Part 2", et cetera in the bug report

[3]https://bugzilla.mozilla.org/

attachments. If all the diffs had reviews from a security team member or a release manager, we inspected them for source code files modified in fixing the vulnerability. If one of the parts did not have an associated review, it was not inspected.

Each reviewed diff was then run through a string parser which looked for strings formatted in source code format; *.cpp, *.c, *.h, et cetera. We recorded each of these occurrences and associated them with the Bugzilla entry the diff was attached to.

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. As an example: the source code file "foo.cpp" could be classified as being associated with both *Code Quality* and *Uncontrolled Resource Consumption* vulnerabilities, but would be assigned the highest severity score between the two vulnerabilities.

When inspecting individual diffs for each bugfix, 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. We removed these files because many bugs also featured updates to test cases for Firefox, which are not included in the scope of our study.

In some cases, files were moved or have 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 performed manual inspection to see if the lines of code modified was still in place in source control using grep and other search tools. 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.

We identified 63 bugs with 232 files with vulnerability fixes from the Mozilla Foundation Security Advisories blog from June 2016 to December 2016. From our Bugzilla 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 $\kappa = 0.6$. 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 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.

B. *Crash Dump Stack Trace Collection*

To collect crash dump stack trace data for RASA, we use Mozilla Crash Reports [35], a site for querying Mozilla's crash report database. Mozilla provides an API for bulk crash collection efforts, leveraging their SuperSearch feature for collection of crashes under specific parameters.

We used SuperSearch generate the URL for our 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 matching our search parameters, and then applied string parsing to the 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 from the "crashing thread" against source code files found in Mozilla Firefox source control. If a file was not found in source control, we discarded it. Some source code files were from other Mozilla products, indicating integration with other products. In some cases, crashes were classified incorrectly by Mozilla Crash Reports. Finally, some third party code was also included in crashes, such as driver files, source code from underlying software like Rust[4], or plugins for the Firefox browser. While vulnerabilities may also be present in other Mozilla products or third party code, we considered these source code files out of scope.

Once third party and unknown code was removed from the "crashing thread", we saved the remaining source code file names to a running "crashing code" list. If a particular source code file had not been seen yet in the "crashing code" list, we added that source code file to the list. If a particular source code file was already seen, we incremented a counter associated with that source code file. For files that occurred multiple times in the "crashing thread" of a crash, we only increment the associated "crashing code" counter once. After parsing the crashes, the "crashing code" list and the associated counters represent RASA for Mozilla Firefox.

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.

VI. Results

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

A. *Coverage of Vulnerability Classifications*

The distribution of coverage of different classifications of vulnerabilities by crashing code can be seen in Figure 3. 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

[4]https://www.rust-lang.org/en-US/

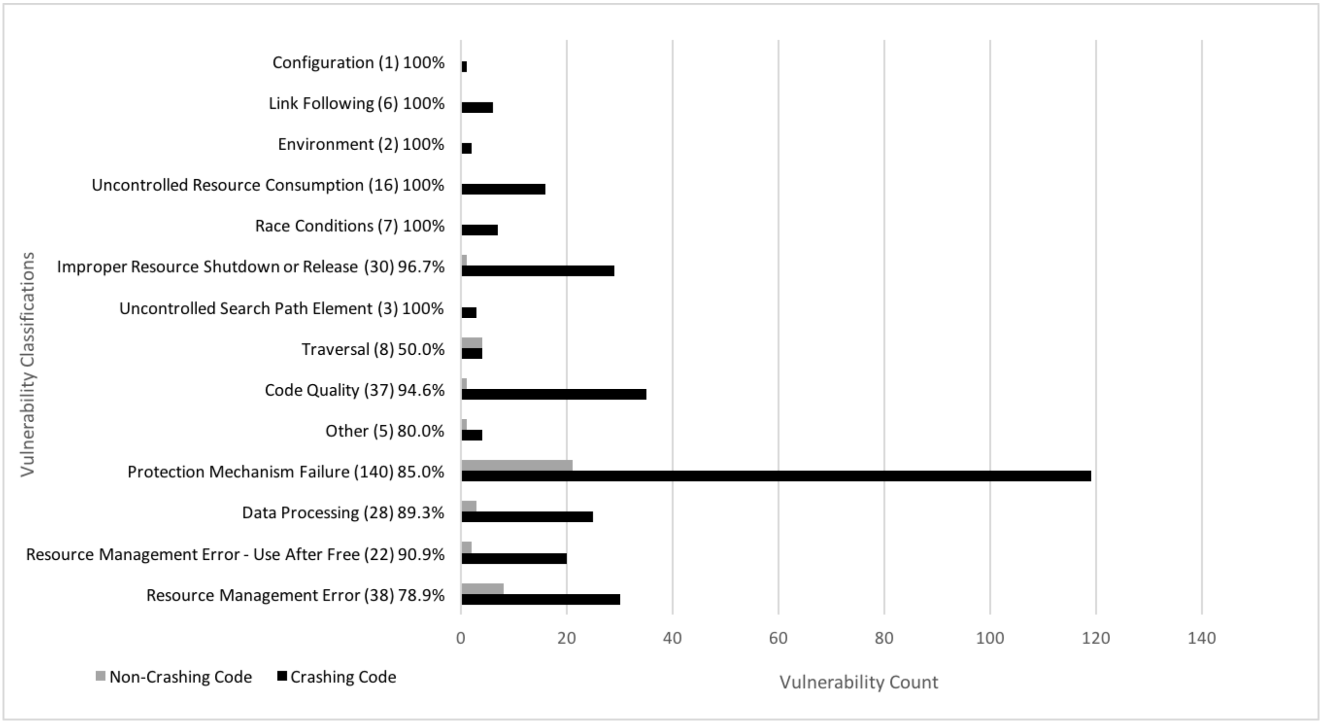

Fig. 3. Distribution of classifications of vulnerabilities occurring on crashing code versus classifications occurring on non-crashing code. The total number of vulnerabilities in this classification is next to the classification label in parenthesis. The percentage of vulnerabilities covered for a specific classification is next to the label. Vulnerability classifications with no entries in our dataset are omitted.

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.

B. *Coverage of Vulnerability Severity*

The distribution of coverage of different severities of vulnerabilities by crashing code can be seen in Figure 4. In our results, we see little variation between the four different categories of severity, based on our previously defined baseline of coverage at 85.3%. Vulnerabilities classified as *Critical* and *Low* featured higher coverage, while *High* and *Moderate* vulnerabilities featured lower coverage. 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.

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.

VII. Discussion

In this section, we discuss the results from our study on Mozilla Firefox, along with some conclusions drawn about the use of vulnerabilities as an oracle for vulnerability prediction models.

A. *Results*

Some of the results of comparing code coverage of crashes to vulnerabilities has unintuitive results. In particular, we were surprised to see that *Resource Management Errors* had a

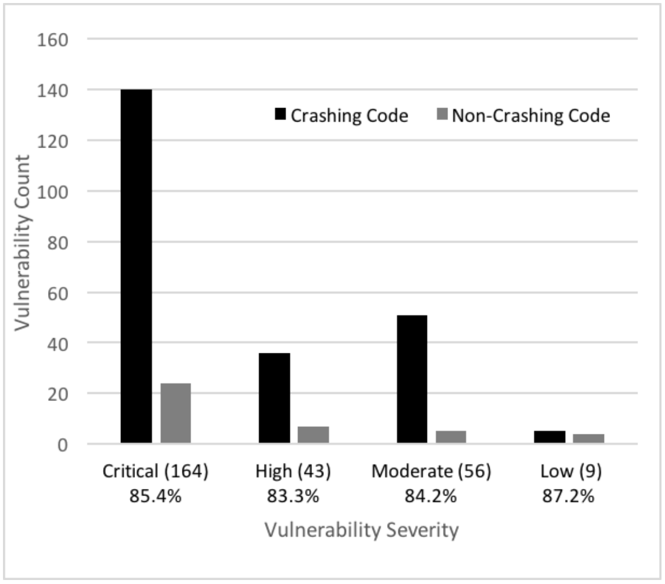

Fig. 4. Distribution of severity of vulnerabilities occurring on crashing code versus severities occurring on non-crashing code. The total number of vulnerabilities for a specific severity is in parenthesis next to the label. The percentage of vulnerabilities covered for a specific severity is next to the label.

lower coverage rate than the baseline (78.9% versus 85.3%). Upon further inspection, the subset of vulnerabilities based on memory corruption issues had a particularly low coverage rate, with only two of the seven memory corruption issues being in crashing code. Our expectation was that crashes would do particularly well with memory corruption issues, as memory corruption issues can also cause crashes. Another classification with a lower coverage rate than the baseline of 85.3% was *Traversal* issues, summarized by CWE as:

"Weaknesses in this classification can be used to access files outside of a restricted directory (path traversal) or to perform operations on files that would otherwise be restricted (path equivalence). Files, directories, and folders are so central to information technology that many different weaknesses and variants have been discovered. The manipulations generally involve special characters or sequences in pathnames, or the use of alternate references or channels."

As the use of special characters or sequences can cause crashes if they are not properly escaped, this was also a surprise. Part of this result could be explained by the low number of vulnerabilities in this classification, with only two having this property. For both of these failures in coverage, future work may be able to determine if this 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 this 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.

RASA covers vulnerabilities of a variety of different classifications. There is little variation in the coverage of vulnerabilities of different severity ratings compared to the baseline coverage, leading us to conclude that RASA does not perform particularly well or particularly poorly in the coverage of the most severe security issues for Mozilla Firefox. Based on these results, we can recommend the use of RASA for prioritizing security efforts for Firefox, and practitioners can have confidence 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 this result generalizes to other products.

B. *Oracles for Vulnerability Studies*

One consequence of this work is a closer examination of oracles for vulnerability prediction or attack surface modeling studies. Previous studies in the the vulnerability prediction and attack surface modeling space have used historical vulnerabilities as an oracle for representing known vulnerable code [3], [4], [16], [15], [14]. 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, differences between vulnerability datasets and defect datasets raises the question on whether the two approaches should be evaluated by the same metric. Kononenko et al. found that the chance for an individual file or function to have a defect in a software system can be as high at 50% [36]. 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.

A study by the authors on vulnerability severity using data from the Fedora operating system was discussed with practitioners at the Symposium and Bootcamp on the Science of Security (HoTSoS 2018) [37] 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 job 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 above discussion point suggests that using historical vulnerabilities as an oracle for VPMs may introduce bias in the validation process. In particular, the VPMs may not consider parts of a software system with the worst issues, because the team maintaining the software may be catching vulnerabilities before they are shipped. On one hand, this could be considered a feature: for teams who are already performing proactive security review and testing, highlighting areas of the codebase they are currently *not* inspecting and testing carefully could have immediate benefits. In that case, not covering the known dangerous areas of the codebase is fine, because the team already knows functions like the above 'Example' function are an issue.

However, a bias towards uninspected areas of the codebase has several negative consequences. First, for the experienced practitioner above, the hypothesized VPM missing a key function like the 'Example' function lowers their trust in the tool. If a VPM 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. Based on these observations, one important branch of future research may be on the oracles used to evaluate vulnerability prediction models.

One possibility is to model the impact of vulnerabilities if they occur in a specific part of the codebase or with specific resources in the system. The practice of *Threat Modeling* is widely used in industry to determine potential threats, and has been used by the research community for security requirements development [38]. Applying *Threat Modeling* to systems highlight areas of the system where a vulnerabilities would cause the most damage. For example, one organization has a policy that mandates security review and testing if code is modified or written that manipulates Personally Identifiable Information (PII). While a vulnerability may not have occurred in that place before, a vulnerability related to PII would be costly for their organization. Pairing data flow analysis while locating critical data in systems could result in another approach for prioritizing security testing and review efforts based on the potential consequences of a failure. For example, a function that handles critically sensitive information could be reviewed and tested after every change, while code that handles less sensitive data could be tested less often.

VIII. Limitations and Future Work

Several threats to validity exist for this 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 not mentioned on the Mozilla Security Advisories blog 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 this 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 used the top level CVE definitions of vulnerabilities for our study, as described in section IV. 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.

Our study describes the use of crash dump stack traces to approximate vulnerable code, and the coverage of various classifications of vulnerabilities. Our study only covers one such approach for approximating vulnerable code. Other approaches, such as the ones created by Munaiah et al. [22] and Younis et al. [39], may cover different classifications of vulnerabilities. One possible result from a replication of this study using other techniques is the discovery of combinations of approaches that result in better coverage, as different techniques might do a better job of covering different classifications of vulnerabilities.

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.

Acknowledgments

The authors would like to thank the Mozilla Firefox team for making crash and defect data available to the public, which made this research possible. We thank the many organizations that contribute to the CVE, and all those who contributed vulnerability classifications to the CVE classification system. We thank the Realsearch group for their feedback on the paper.

References

[1] M. Howard, J. Pincus, and J. M. Wing, "Measuring relative attack surfaces," in *Computer Security in the 21st Century*. Springer, 2005, pp. 109–137.

[2] P. K. Manadhata and J. M. Wing, "An attack surface metric," *IEEE Transactions on Software Engineering*, vol. 37, no. 3, pp. 371–386, 2011.

[3] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating attack surfaces with stack traces," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 2. IEEE, 2015, pp. 199–208.

[4] C. Theisen, K. Herzig, B. Murphy, and L. Williams, "Risk-based attack surface approximation: how much data is enough?" in *Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE/ACM 39th International Conference on*. IEEE, 2017, pp. 273–282.

[5] Z. Durumeric, J. Kasten, D. Adrian, J. A. Halderman, M. Bailey, F. Li, N. Weaver, J. Amann, J. Beekman, M. Payer *et al.*, "The matter of heartbleed," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 475–488.

[6] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "Ddos in the iot: Mirai and other botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[7] M. Castelluccio, C. Sansone, L. Verdoliva, and G. Poggi, "Automatically analyzing groups of crashes for finding correlations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 717–726. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106306

[8] F. A. Bianchi, M. Pezzè, and V. Terragni, "Reproducing concurrency failures from crash stacks," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 705–716. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106292

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

[10] N. Chen and S. Kim, "Star: Stack trace based automatic crash reproduction via symbolic execution," *IEEE transactions on software engineering*, vol. 41, no. 2, pp. 198–220, 2015.

[11] J. Xu, D. Mu, P. Chen, X. Xing, P. Wang, and P. Liu, "Credal: Towards locating a memory corruption vulnerability with your core dump," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 529–540.

[12] W. Fu and T. Menzies, "Revisiting unsupervised learning for defect prediction," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 72–83.

[13] A. Hovsepyan, R. Scandariato, and W. Joosen, "Is newer always better?: The case of vulnerability prediction models," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2016, p. 26.

[14] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.

[15] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[16] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 421–428.

[17] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification-a concept for in-process measurements," *IEEE Transactions on software Engineering*, vol. 18, no. 11, pp. 943–956, 1992.

[18] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 465–475.

[19] M. Sullivan and R. Chillarege, "A comparison of software defects in database management systems and operating systems." in *FTCS*, 1992, pp. 475–484.

[20] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*. ACM, 2006, pp. 25–33.

[21] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, vol. 32, no. 3, pp. 70–77, 2015.

[22] N. Munaiah and A. Meneely, "Beyond the Attack Surface: Assessing Security Risk with Random Walks on Call Graphs," in *Proceedings of the 2016 ACM Workshop on Software PROtection*, ser. SPRO '16. New York, NY, USA: ACM, 2016, pp. 3–14. [Online]. Available: http://doi.acm.org/10.1145/2995306.2995311

[23] A. Austin and L. Williams, "One technique is not enough: A comparison of vulnerability discovery techniques," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 97–106.

[24] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 308–318.

[25] C. Theisen, H. Sohn, D. Tripp, and L. Williams, "Better together: Combining vulnerability prediction models," in *Submitted to the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. IEEE, 2018.

[26] S. Zhang, X. Zhang, X. Ou, L. Chen, N. Edwards, and J. Jin, "Assessing attack surface with component-based package dependency," in *International Conference on Network and System Security*. Springer, 2015, pp. 405–417.

[27] J. C. Foreman and D. Gurugubelli, "Identifying the cyber attack surface of the advanced metering infrastructure," *The Electricity Journal*, vol. 28, no. 1, pp. 94–103, 2015.

[28] D. Geneiatakis, "Minimizing databases attack surface against sql injection attacks," in *International Conference on Information and Communications Security*. Springer, 2015, pp. 1–9.

[29] Z. Zhang, Y. Cheng, S. Nepal, D. Liu, Q. Shen, and F. Rabhi, "A reliable and practical approach to kernel attack surface reduction of commodity os," *arXiv preprint arXiv:1802.07062*, 2018.

[30] D. Kantola, E. Chin, W. He, and D. Wagner, "Reducing attack surfaces for intra-application communication in android," in *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 69–80.

[31] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? a study of the chromium project," in *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*. IEEE, 2015, pp. 269–279.

[32] K. Herzig, S. Just, and A. Zeller, "It&#039;s not a bug, it&#039;s a feature: How misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 392–401. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486840

[33] J. Cohen, "Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit." *Psychological bulletin*, vol. 70, no. 4, p. 213, 1968.

[34] (2017) Mozilla foundation security advisories. [Online]. Available: https://www.mozilla.org/en-US/security/advisories/

[35] (2017) Mozilla crash reports. [Online]. Available: https://crash-stats.mozilla.com/home/product/Firefox

[36] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 1028–1038. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884840

[37] C. Theisen and L. Williams, "How bad is it, really? an analysis of severity scores for vulnerabilities: Poster," in *Proceedings of the 5th Annual Symposium and Bootcamp on Hot Topics in the Science of Security*, ser. HoTSoS '18. New York, NY, USA: ACM, 2018, pp. 20:1–20:1. [Online]. Available: http://doi.acm.org/10.1145/3190619.3191694

[38] S. Myagmar, A. J. Lee, and W. Yurcik, "Threat modeling as a basis for security requirements," in *Symposium on requirements engineering for information security (SREIS)*, vol. 2005. Citeseer, 2005, pp. 1–8.

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