Beyond the Attack Surface

Assessing Security Risk with Random Walks on Call Graphs

Nuthan Munaiah
nm6061@rit.edu

Andrew Meneely
axmvse@rit.edu

Department of Software Engineering
Rochester Institute of Technology, Rochester, New York, USA

ABSTRACT

When reasoning about software security, researchers and practitioners use the phrase "attack surface" as a metaphor for risk. Enumerate and minimize the ways attackers can break in then risk is reduced and the system is better protected, the metaphor says. But software systems are much more complicated than their surfaces. We propose function- and file-level attack surface metrics—proximity and risky walk—that enable fine-grained risk assessment. Our risky walk metric is highly configurable: we use PageRank on a probability-weighted call graph to simulate attacker behavior of finding or exploiting a vulnerability. We provide evidence-based guidance for deploying these metrics, including an extensive parameter tuning study. We conducted an empirical study on two large open source projects, FFmpeg and Wireshark, to investigate the potential correlation between our metrics and historical post-release vulnerabilities. We found our metrics to be statistically significantly associated with vulnerable functions/files with a small-to-large Cohen's *d* effect size. Our prediction model achieved an increase of 36% (in FFmpeg) and 27% (in Wireshark) in the average value of F_2-measure over a base model built with SLOC and coupling metrics. Our prediction model outperformed comparable models from prior literature with notable improvements: 58% reduction in false negative rate, 81% reduction in false positive rate, and 548% increase in F_2-measure. These metrics advance vulnerability prevention by (a) being flexible in terms of granularity, (b) performing better than vulnerability prediction literature, and (c) being tunable so that practitioners can tailor the metrics to their products and better assess security risk.

Keywords

metric; vulnerability; attack surface; page rank; risk

1. INTRODUCTION

In our digital world, software must be secure. As customers, patients, and citizens, we rely on our software to provide confidentiality, integrity, and availability. The responsibility to engineer secure software rests on developers, who must understand complex security risks that change with each new line of code they write. Developers inevitably make mistakes, and the risk of those mistakes can be quantitatively tracked with metrics.

The *attack surface* metaphor is often invoked as a way to assess the security risk of large systems [14, 18, 15, 20, 19]. The metaphor goes like this: if an attacker has more avenues to enter the system, then the system is at a higher risk. Software systems provide these avenues in their inputs and outputs, where inputs can take in potential exploits and outputs provide information on how data is processed (e.g. lack of sanitization). Thus, broadly speaking, developers can understand their security risks by understanding their inputs and outputs. This metaphor has been the inspiration for security risk metrics such as the number of *entry points* and *exit points* [20, 21, 22] and is even in practice at Microsoft [18].

Historically, attack surface metrics have focused on the system's "perimeter" and hence fail to capture what happens beyond the entry and exit points. A simple source code change, such as the addition of an API call deep within the system, can have a drastic effect on security risk because large software systems are vastly interconnected and the API call could be connecting two subsystems that were earlier disconnected. If the attack surface metaphor is about examining avenues of attack, then risk analysis should be about how attacker *traverses* the software system.

The *call graph* [13, 12, 1, 7] can be used as a basis for a model of attacker behavior because it contains the overall system structure. Random walks through a call graph can serve as an approximation for attacker searching (manually or automatically) for vulnerabilities. With network analysis techniques [3, 30, 39, 28] such as centrality, random walks, and geodesic paths, we can provide developers with fine-grained risk metrics at the function- or file-level that can be used to continually assess the impact of a source code change on security risk.

The goal of this research is to assess security risk through an empirical understanding of the relationship between vulnerabilities, individual functions or files, and the attack surface of a software system. We apply the attack surface metaphor described in prior literature [14, 15, 22] to the call graph and propose function- and file-level metrics—proximity and risky walk—that simulate random walks across functions/files to account for system structure beyond the surface. We empirically analyze our metrics in two large open source projects:

SPRO'16, October 28 2016, Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-4576-7/16/10...$15.00
DOI: `http://dx.doi.org/10.1145/2995306.2995311`

the FFmpeg media transcoder and the Wireshark network protocol analyzer to understand if the metrics are associated with historical vulnerabilities. We also investigate the sensitivity of the metrics to call graph collection approach (static-only vs. static+dynamic), as well as parameter tuning for our random walk metric.

We address the following research questions:

RQ1 Association Is a function/file more likely to be fixed for a post-release vulnerability if:
- (a) it is near the attack surface or dangerous points?
- (b) it has a higher probability of being traversed on a random walk from the attack surface?

RQ2 Prediction (Base) Do proximity and risky walk metrics improve the performance of a base prediction model built with SLOC and coupling metrics?

RQ3 Prediction (Prior) How do the prediction models built with proximity and risky walk metrics compare with prior vulnerability prediction literature?

The research contributions of this work are:

- A method for applying the attack surface metaphor to individual functions/files so that developers can quantitatively assess security risk;
- Implementation of our method as an open source project [34];
- Empirical evaluation of our method in two large open source projects.

The remainder of the paper is organized as follows: in Section 2, we present the motivation for our metrics. In Section 3, the metrics are defined and the approach to collect the metrics is presented. We present the methodology used in the empirical evaluation of the metrics in Section 4 and outline our results in Section 5. We discuss our limitations in Section 6, present an overview of the state of the art in attack surface research in Section 7, and conclude with a summary in Section 8.

2. METRIC MOTIVATION

In this section we discuss some key decisions that led to the formation of our risk metrics. Modeling attacker behavior using graphs is not new [42, 25], nor is using call graphs [20, 21, 51], but our approach of combining attack surface and call graph is unique to our knowledge.

Historically [14], software attack surface researchers have concluded that reducing points of entry/exit should reduce the number exploits attackers can use. We expand the scope of that argument by applying the principle of *defense in depth*. We wish to consider risks of attack by aggregating how the system is interconnected, still using the attack surface as a starting point, to model attacker behavior.

Consider the attack surface metaphor as it applies to medieval castles. If a castle has many different entrances and exits, attackers will also have many different ways of invading the castle. Reduce entrances and the security will improve. But, an inner courtyard might also be particularly risky because it is the point of convergence for multiple pathways into the castle. A renovation within the castle might change attacker behavior without changing the external entry and exit points. By combining both structure and surface, we can better understand our system from the inside out.

Why use call graphs?

Call graphs provide an inexpensive, automated way of measuring how the system is interconnected. Most programming languages support call graph collection, so these metrics could be easily adopted into, say, a continuous integration build. Call graphs are also the next natural step from the attack surface methods because entry/exit points in prior attack surface literature [20, 21, 22] are actual methods.

While call graphs are simple to collect they are, in practice, imperfect representations of what can happen at runtime because of pointer manipulations. In our approach discussed in the subsequent section, we discuss how one could mitigate this concern and how, in our empirical study, we found our call graph to be a close representative of potential attacker pathways. We also examined the sensitivity of static-only vs. static+dynamic call graph collection in our empirical study in Section 4.2.

Once customized to the build process of a software system, our proposed approach is entirely automated. We envision developers using these fine-grained metrics as an informative part of their everyday development workflow to prioritize their software protection efforts such as code reviews and penetration testing.

Why use random walk metrics?

Many network analysis applications rely upon *geodesic* path (i.e. "shortest path") metrics to provide analysis. Geodesic path metrics are useful in social situations where distance is to be measured and the attempt to find a shortest path (e.g. two humans who are "friends" are also "friends-of-friends", but the shortest path makes the most sense in that domain). We use geodesic paths whenever we want to gauge potential distance (e.g. "distance to the attack surface").

Computers, however, do not execute methods based on shortest paths, they execute methods based on inputs. For attackers searching for a vulnerability, perhaps via a fuzz testing tool or through manual experimentation, traversals of the network would look more random. An attacker, via his inputs, would execute commonly-used methods, leading to a higher risk of attack if those methods had a vulnerability. Thus, we use random walk metrics to simulate the attacker behavior of exploring the system.

Why use PageRank and not regular Random Walk?

The standard Random Walk metric is useful when discussing how someone traverses a network infinitely (e.g. car traffic patterns). Attackers, however, are constantly starting over their traversals–at the attack surface. The PageRank metric, based on Google's algorithm of modeling web surfing behavior, is a Random Walk with additional parameters of (a) damping factor (a probability of "starting over"), and (b) personalization vector (the probability of starting at a particular method). The concept of constantly starting over at the attack surface and then conducting a random walk fits with the scenario of an attacker exploring for a vulnerability. Thus, to incorporate the notion of attack surface into call graph centrality, we found PageRank to be the most appropriate metric.

3. PROPOSED METRICS

In this section, we introduce the proximity and risky walk metrics in the context of the attack surface metaphor introduced earlier. A unique (read desirable) feature of our metrics is that they can be defined at either function- or file-level, enabling developers to choose the level of granularity in risk assessment.

The high-level approach to collecting the metrics, at either the function- or file-level, may be summarized as follows:

Step 1: Obtain the call graph
Step 2: Identify Entry, Exit, and Dangerous Points
Step 3a: Compute Proximity metrics
Step 3b: Compute Risky Walk metric

We have developed an open source tool, called *Attack Surface Meter* [34], that enables the collection of the metrics proposed in this paper for a software system written in the C programming language. As of this writing, *attack surface meter* is capable of parsing call graphs obtained from GNU cflow and GNU gprof. The *attack surface meter* may be extended to measure software systems written in other programming languages by defining a parser for the call graph generated by an appropriate language-specific utility.

Step 1: Obtain the call graph

The proximity and risky walk metrics are defined on the call graph representation of a software system. The call graph represents a series of steps that an attacker effectively takes when attempting to exploit a vulnerability. In our definition, a call graph is a collection of directed relationships from *caller* functions to *callee* functions, represented as a symmetric directed graph. That is, if an attacker's exploit accesses a callee, it can potentially access the caller, and vice versa. We use a symmetric directed graph instead of an undirected graph so that we can weight returns differently than calls. (Note: our predecessors [20, 21, 51] have used directed but not symmetric call graphs, which we believe does not fully account for function returns.)

The caller—callee relationships may be deduced in two ways: (1) *static analysis*, where the source code is parsed and analyzed, or (2) *dynamic analysis*, where the software system is profiled during execution. Both techniques are imperfect: static analysis has limited support for language features such as function pointers and polymorphism, and dynamic analysis requires software to compile and all possible execution paths to be exercised. Thus, the call graph is always an approximation as Grove et al. mention when describing *soundness* of call graphs [12]. We hypothesize that the level of approximation achieved by the call graph may be improved if static and dynamic analyses are used in unison. In Section 4.2, we present the sensitivity of our metrics to static-only vs. static+dynamic analysis.

Developers familiar with their own systems can manually inspect the call graph for soundness, as we did in both of our historical studies. Based on applying our technique to historical case studies, we also used two heuristics for gauging if the collected call graph has a representative set of edges. The two heuristics we used were:

Number of Fragments (f) The number of strongly connected components in the call graph

Monolithicity (m) The percentage of total functions that is in the largest strongly connected component.

We note that these heuristics are useful in systems that are "monolithic", that is, systems that are intended to compile into one massive call graph. While this happened to be true in our empirical studies, it may not be in, say, an API with intentionally disconnected subsystems such as `glibc`.

In practice, software systems often have functions that are never invoked or functions that are invoked only when testing. These functions will appear as islands in the call graph, so the ideal m is not necessarily 100% and the ideal f is not necessarily one. The ideal value of m and f vary between systems, and must be taken into consideration.

The call graph described so far is conventional in that the edges in the graph represent function call/return. The edges in the conventional call graph may be used to deduce call/return relationships between files as well. The *file-level call graph* enables the proximity and the risky walk metrics to be collected at a file-level, providing an alternate approach to security risk assessment.

Step 2: Identify Entry, Exit, and Dangerous Points

Attackers need places to send their attacks, or places where they might start the reconnaissance for their attacks. From Manadhata et al. [22], we defined these functions as *entry points* (where data enters in) and *exit points* (where data exits out). Additionally, attackers may be more likely to target functions that make system calls deemed dangerous. We call these functions as *dangerous points*. There may be other criteria in which a function could be deemed dangerous (e.g. function handling sensitive information specific to an application), however, we restrict ourselves to only functions that make dangerous system calls.

The language used in the development of a system and its operating environment determine how inputs, outputs, and the dangerous system calls are identified. For instance, in the context of a web application, an entry point could be a method that saves form-posted data to a database, whereas, an exit point could be a method that formats data for a web page. In the context of a C program: any function invoking a C standard input function (e.g. `scanf`, `getc`, etc.) is an entry point, whereas, any function invoking a C standard output function (e.g. `printf`, `putc`, etc.) is an exit point. Again, in the context of a C program, any function making a dangerous system call (e.g. `chown`, `fork`, etc.) is a dangerous point. Defining entry and exit points for all available technologies may be an open problem, but the most popular set of functions for the C standard library is in Appendix A of Manadhata and Wing [19]. Similarly, the set of available system calls is dependent on the version of the C standard library used during the development of a software system. We have used the system calls with threat level 1 through 3 enumerated by Bernaschi et al. [2].

The notion of entry, exit, and dangerous points may be extended to the file-level by applying the following heuristic: a file is an entry point, exit point, or dangerous point if it contains at least one function that is an entry point, exit point, or dangerous point, respectively.

Step 3a: Compute Proximity Metrics

As an attacker's exploit enters the system, functions/files that are near entry and exit points are likely to be involved with handling user data. In the call graph, nearby ancestors (i.e. functions/files that can reach a given function/file) may be more likely to have security risks, so we use an unweighted shortest path algorithm to determine the distance from a given function/file to the attack surface. Since functions/files may be reachable from multiple entry and exit points, we average the shortest path lengths. We chose to use the average of the shortest path lengths to better approximate the reality of function/file invocation pattern. In addition to measuring the distance to the attack surface, we also measure the distance of a function/file to dangerous points that were defined in the previous step.

Our three proximity metrics (i.e. proximity to entry, proximity to exit, and proximity to dangerous) are defined as:

Definition 1. *Proximity of a function/file `foo` to entry points, exit points, or dangerous points is the mean of the shortest unweighted path lengths to all functions/files reachable from `foo` that are entry points, exit points, or dangerous points, respectively.*

The power of the proximity metric is in its sensitivity to the (unforeseen) ripple effect of a source code change on functions that may have not been directly modified by the developer. For example, suppose a developer working on a function, `readMessage`, adds a call to `pingServer`. The proximity of `readMessage`, and all its descendants (i.e. functions reachable from `readMessage`), to the entry surface would decrease if `pingServer` is near an entry point. Conversely, a refactoring effort on `pingServer` that separates concerns of input validation or secure memory management would increase the proximity of `readMessage` to the entry surface without a direct change to it.

Step 3b: Compute Risky Walk Metric

Attackers looking for vulnerabilities may have limited knowledge of the system's source code and are essentially exercising different execution paths in the system hoping to find a vulnerability. This behavior of the attacker is similar to that of a World Wide Web user ("surfer") searching for a piece of information. The surfer starts at, say, the results from a search engine and follows a series of links until she finds the information she was looking for. Or perhaps she deems the search futile, at which point she returns back to the starting point and follows a different series of links. The starting point of the surfer is analogous to the entry points of the software system and the act of following a link is analogous to invoking a function in the software system. However, the attacker has no direct control over the series of function calls that the system makes in response to a particular input. As a result, the attacker resorts to trying several entry points with varying inputs.

In addition to ranking web pages [38], twitter users [16], and improving recommender systems [23], PageRank algorithm has found application in the realm of security as well [47, 25].

The PageRank algorithm uses three configurable parameters in addition to the call graph, they are:

- a *personalization vector*, v, that contains the probability that a random walk starts at a given node,
- a *damping factor*, α, that defines the probability that an attacker will continue the random walk across the call graph without abandoning the current walk and starting over, and
- an *edge weights vector*, w, that contains the edge weights used to derive the probability that a random walk traverses one of the many possible edges from a given node.

Wills [50] presents an elaborate description of the mathematics behind the computation of the page rank and the role of these parameters in the algorithm.

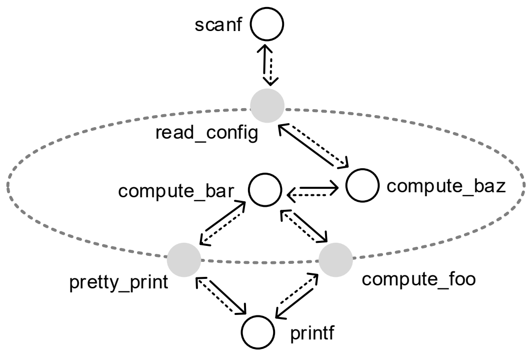

Figure 1: Attack surface visualization of a sample C program

In the context of security risk assessment, these parameters must be chosen with the intent of reflecting attacker mindset. For instance, we may want to assign a higher weight to edges terminating at functions that were fixed for vulnerabilities in the past to model the likelihood that an attacker may try to attack past vulnerable functions hoping to uncover a new vulnerability. Similarly, some software systems may have defenses in place to wraparound standard library functions known to be used incorrectly by developers. In such cases, we may want to assign a lower weight to edges terminating at such defensive functions.

In the empirical analysis of the risky walk metric in FFmpeg and Wireshark, we carried out an extensive parameter tuning exercise (detailed in Appendix A) to identify a robust set of parameters. Users of our approach may choose to use the parameters we arrived at in our study as our parameters ended up being similar across case studies. Alternatively, users may choose to use our weights as a starting point and adapt them to their own software systems.

Our Risky Walk metric is defined as follows:

Definition 2. *Risky walk of a function/file is the PageRank of that node in the call graph computed with a personalization vector, a damping factor, and an edge weights vector tuned to simulate attacker behavior.*

The power of risky walk metric is that it aims to simulate, by means of probabilities, the behavior of a typical attacker, specifically during the reconnaissance phase of an attack. The risky walk of a function/file is the probability that a random execution of the system, with inputs tailored to uncover vulnerabilities, will result in the function/file being invoked. In practice, risky walk of a function/file may be extremely small in a system with large number of functions/files, so viewing the logarithm of the risky walk or simple a ranking can make the values easier to interpret.

3.1 An Example

Figure 1 shows the call graph of a sample C program with the attack surface highlighted with dotted ellipse. The nodes represent the functions in the program. The solid directed edge represents call to a function, whereas, the dotted directed edge represents return from a function. The function `read_config` is an entry point because it calls an input function `scanf`. The functions `compute_foo` and `pretty_print` are exit points because they call the output function `printf`. The entry and exit points are shaded gray. For simplicity, the graph does not show any dangerous points.

The proximity to entry for `compute_baz` is 1 and for `compute_bar` it is 2. The proximity to exit for `compute_baz` is $(2+2)/2 = 2$ and for `compute_bar` it is $(1+1)/2 = 1$. Let $\alpha = 0.85$, vector v contain 0.3125 for entry and exit points and 0.03125 for the other functions (i.e. attacker is 10 times more likely to start at entry or exit point), and the vector w contain 10 for call edges and 5 for return edges. For risky walk, the function `compute_bar` would have a total weighting of $10+10+5 = 25$ for the outgoing edges. Each edge weight is computed as a proportion of the total, for example, the `compute_bar` to `compute_baz` edge probability would be $10/25 = 0.4\%$. The page rank of `compute_baz` will be 0.29 and `compute_bar` will be 0.27.

4. METHODOLOGY

In this section, we describe the methodology used in the empirical evaluation of the proximity and risky walk metrics in the context of two large open source projects: FFmpeg and Wireshark. At a high-level, the empirical evaluation was conducted in four phases, they are:

Phase I: Metric Collection
Phase II: Function/File Labeling
Phase III: Association Analysis
Phase IV: Regression Analysis

All statistical tests were executed on `R` version 3.2.3 [40].

4.1 Study Subjects

We chose two large open source projects as subjects of study in the empirical evaluation of our metrics. The motivation for choosing our subjects of study were: (a) large, popular, and open source projects, (b) well-kept vulnerability fix records, (c) substantial development history for tracking vulnerabilities over time, and (d) automated regression test suites to compare the sensitivity of static-only vs. static+dynamic analysis.

FFmpeg is a popular open source media transcoding library that is capable of encoding, decoding, multiplexing, demultiplexing, streaming, filtering, and playing an enormous variety of media. FFmpeg is used by other projects such as Google Chrome and the VLC Media Player. Since 2009, FFmpeg has had 19 major releases, 217 patch releases, and 237 vulnerabilities. In this study, we collected metrics for 16 of the 19 major releases of FFmpeg, mining 675 vulnerability-fixing commits to identify 280 unique functions that were fixed for a post-release vulnerability. On average, each major release of FFmpeg has 536k source-lines-of-code (SLOC) and 12,908 functions spread across an average of 1,865 files. In terms of SLOC, the average length of each file is 306 and that of each function is 30. FFmpeg has an extensive, automated regression test suite call FATE[1], which we leveraged for dynamic analysis.

Wireshark is an open source network protocol analyzer that has come to be the de facto standard for sniffing network data. The current release of Wireshark supports the analysis of 2,000 network protocols. Since 2008, Wireshark has had 8 major releases, 118 patch releases, and 312 vulnerabilities. In this study, we collected metrics for 7 of the 8 major releases of Wireshark, mining 590 vulnerability-fixing commits to identify 1,705 unique functions that were fixed for a post-release vulnerability. On average, each major release of Wireshark has 2,081 kSLOC and 53,350 functions spread across an average of 2,593 files. In terms of SLOC, the average length of each file is 855 and that of each function is 24. We collected dynamic analysis data by developing a simple test runner script to invoke the Wireshark GUI (and its command line variant, TShark) for a set of 1,877 packet capture files typically used for regression testing by the Wireshark team.

[1]`https://ffmpeg.org/fate.html`

4.2 Phase I: Metric Collection

In this section, we apply the method introduced in Section 3 to collect the proximity and risky walk metrics from the releases of FFmpeg and Wireshark considered in our study. In addition to *attack surface meter*, we have developed another open source application, called *Attack Surface Evolution*,[2] to facilitate the automated (and parallel) collection of the metrics for multiple releases of a software system. The metrics collected are saved to a database for further analysis.

We note that we collected, and analyzed, the metrics at both function- and file-level to evaluate the utility of the metrics at different levels of granularity.

Step 1: Obtain the call graph

We chose two popular call graph generation utilities: GNU cflow,[3] a static call graph generation utility, and GNU gprof,[4] a dynamic profiling utility, to obtain the call graphs of FFmpeg and Wireshark. We refer to these as `cflow` and `gprof` in the remainder of the section, respectively.

In order to determine the need for dynamic analysis, we use the heuristics—number of fragments and monolithicity—from Section 3. We obtain static and dynamic call graphs for the most recent release of FFmpeg and Wireshark considered in our study and use the heuristics to determine if there is a need for dynamic analysis or not. The most recent releases of FFmpeg and Wireshark considered in this study are 2.5.0 and 1.12.0, respectively. Since FFmpeg and Wireshark are intended to compile into a single system, we expect number of fragments to be low and monolithicity to be high. The number of fragments (f) and monolithicity (m) of the FFmpeg version 2.5.0 and Wireshark version 1.12.0 are given in the Table 1.

`cflow` and `gprof` call graphs are saved to the disk as plain-text files. The *attack surface meter* parses the textual call graph files and produces a single call graph that represents the software system. The *attack surface meter* uses NetworkX[5] version 1.9.1 to represent the call graph. The nodes in the call graph represent the functions in the software system and the edges represent transfer of control.

Step 2: Identify Entry, Exit, and Dangerous Points

The C standard input and output functions, used to identify the entry points and exit points in FFmpeg and Wireshark, are the same as those listed in Appendix A of previous work [19] by Manadhata. The dangerous system calls used in our study are the system calls with threat level 1 through 3 enumerated by Bernaschi et al [2].

With the entry, exit, and dangerous points identified, functions belonging to the C standard library were removed from the call graph to prevent these functions from being accounted for in the computation of our metrics.

[2]`https://github.com/nuthanmunaiah/attack-surface-evolution`
[3]`http://www.gnu.org/software/cflow/`
[4]`https://sourceware.org/binutils/docs/gprof/`
[5]`http://networkx.github.io/`

Table 1: Number of Fragments (f), Monolithicity (m), and mean value of Proximity to Entry (p_{en}), Proximity to Exit (p_{ex}), Proximity to Dangerous (p_{da}), and Risky Walk (rw) from static and static+dynamic analysis of FFmpeg version 2.5.0 and Wireshark version 1.12.0

Subject (Version)	Static						Static+Dynamic					
	f	m	$\mu_{p_{en}}$	$\mu_{p_{ex}}$	$\mu_{p_{da}}$	μ_{rw}	f	m	$\mu_{p_{en}}$	$\mu_{p_{ex}}$	$\mu_{p_{da}}$	μ_{rw}
FFmpeg (2.5.0)	182	0.979	3.669	3.807	3.498	6.530E-05	124	0.985	3.666	3.832	3.546	5.727E-05
Interpretation: Appending call graphs obtained through dynamic analysis decreased the number of fragments and marginally increased the monolithicity. Furthermore, there was a non-trivial change in the mean value of risky walk metric. Hence, we use dynamic analysis when obtaining the FFmpeg call graph.												
Wireshark (1.12.0)	78	0.998	4.197	4.408	4.335	1.527E-05	82	0.998	4.195	4.403	4.334	1.525E-05
Interpretation: Appending the call graphs obtained through dynamic analysis neither decreased the number of fragments nor increased the monolithicity. Furthermore, there was a trivial change in the mean value of the metrics. Hence, we do not use dynamic analysis when obtaining the Wireshark call graph.												

Step 3a and 3b: Compute Proximity and Risky Walk Metrics

The *attack surface meter* has methods to compute the proximity and risky walk metrics for a given function or file. These methods use the `shortest_path_length` and `page_rank` methods from the NetworkX API.

4.3 Phase II: Function/File Labeling

To evaluate the efficacy of our metrics as an indicator of vulnerabilities, we must understand *fixes* to historical post-release vulnerabilities and identify those functions/files that were vulnerable in the past. We labeled functions/files that were fixed for post-release vulnerability as *vulnerable* and all other functions/files as *neutral*. We refer to these two groups of functions/files as vulnerable functions/files and neutral functions/files, respectively.

We begin by collecting a list of publicly disclosed vulnerabilities from Common Vulnerabilities and Exposures (CVE)[6], National Vulnerability Database (NVD)[7], or the security advisories section on the project's website. The security advisories section on the project's website is more suited for our purposes as it contains the versions of software affected by a vulnerability and information that helps trace vulnerability fixes to the source code. Furthermore, the security advisories are released only when a publicly-disclosed vulnerability is acknowledged and fixed by the project team. The FFmpeg and Wireshark project teams post their security advisories at `https://www.ffmpeg.org/security.html` and `https://www.wireshark.org/security/`, respectively. In our study, for each historical vulnerability fixed by the development team, we collected the commit identifier that the project team reports as containing the fix. For each vulnerability-fixing commit identifier collected, we generated a patch using the `git` client and parsed the patch output (similar to [27]) to identify the name of the functions/files affected by the commit. We manually examined a random sample of the patches to ensure that the fix commit was not combined with other changes. In our sample, we found none of these cases to be true for FFmpeg and Wireshark.

4.4 Phase III: Association Analysis

We used the non-parametric Mann-Whitney-Wilcoxon (MWW) test to understand how well the proximity and risky walk metrics reflect the reality of historical post-release vulnerabilities. We consider the association between a given metric and post-release vulnerabilities to be statistically significant if the p-value is less than 0.05. We use the population median to determine if a metric is higher (or lower) for vulnerable functions/files when compared with neutral functions/files.

Association analysis merely reveals if there is a statistically significant difference between the distribution of the metric values collected from a population of vulnerable functions/files from that of the metric values collected from a population of neutral functions/files. We complemented the association analysis with Cohen's d effect size evaluation to assess the *strength* of association (if any, as revealed by MWW test). We used the heuristics proposed in Cohen's d literature [6] when interpreting the effect size. According to the heuristic, an effect is considered large if $|d| \geq 0.8$, medium if $|d| \geq 0.5$, small if $|d| \geq 0.2$, negligible otherwise.

4.5 Phase IV: Regression Analysis

In the regression analysis phase, the goal is to assess if the proximity and risky walk metrics can be used in building a regression model capable of predicting the likelihood of a function/file needing a fix for a post-release vulnerability in the future. A necessary condition in the evaluation of the efficacy of such a model is to assess if it performs better than a base prediction model built with SLOC and coupling metrics. The coupling metrics used in this study are the structural variant of fan in and fan out of a function/file. *Fan in* is the number of functions/files that call a given function/file and *fan out* is the number of functions/files that a given function/file calls. We chose to use SLOC, fan in, and fan out in building the base model because these metrics have been shown to be good predictors of vulnerabilities [54, 43].

While fan in and fan out were collected directly from the call graph, SLOC was measured using Scitools Understand[8]. Functions/files for which SLOC was not available from Understand were omitted when training and testing the model. We note that SLOC was available for all functions/files that were fixed for a post-release vulnerability.

We used two approaches in the training and testing of the regression models: (a) Cross-validation and (b) Next release validation. We used precision, recall, and F_2-measure to evaluate the performance of a model against the base model. In contrast to F_1-measure, the F_2-measure weights

[6]`https://cve.mitre.org/`

[7]`https://nvd.nist.gov/`

[8]`https://scitools.com/understand/`

recall higher than precision. A model that exhibits a higher recall is desirable in vulnerability prediction [31, 5, 43]. We note that the performance of a model was evaluated if and only if the model had at least one statistically significant (p-value ≤ 0.05) feature.

In *cross-validation*, a model is repetitively trained and tested with random splits of the data from a single release. We used stratified sampling when randomly splitting the data set to ensure equal proportion of vulnerable and neutral functions/files was maintained between the training and testing splits. In our study, we performed 10 repetitions of a 10-fold cross-validation. In other words, we trained and tested 100 models in each of the 23 releases of FFmpeg and Wireshark. The performance metrics—precision, recall, and F_2-measure—were aggregated across the 100 models.

In *next release validation*, a model is trained with historical vulnerability data and tested by attempting to predict *known* future vulnerabilities. For example, consider a scenario where FFmpeg version 1.1.0 is being prepared for release. Retrospectively, all functions fixed for a post-release vulnerability in releases leading up to, and including, 1.1.0 are used in training the model. The model is then used to predict functions that are likely to require a fix for a post-release vulnerability in the future. The predictions are validated by comparing them against all functions known (in the context of this study) to be fixed for a post-release vulnerability in patch releases from the 1.1.x branch (i.e. FFmpeg releases 1.1.1 to 1.1.16).

While cross-validation is an acceptable, and commonly used [46, 54, 44, 9], approach, next release validation is intuitive and closer to reality. We chose to use both approaches to ensure that the performance of our models can be compared with those from prior vulnerability prediction literature.

Vulnerabilities are rare; the act of predicting vulnerabilities in software systems has been compared with searching for a needle in a haystack [54]. As an example, on average, a mere 0.67% of functions in FFmpeg were fixed for a vulnerability. At a file-level, however, an average of 3.47% of files in FFmpeg were fixed for a vulnerability. Predicting at a higher level of granularity may partially alleviate the problem of disproportionately sized populations of vulnerable and neutral entities. However, even at the file-level, the number of vulnerable entities are so few that the prediction models may be biased toward neutral entities resulting in a considerably high false negative rate. We have used a popular approach to dealing with class imbalanced data sets called SMOTE [4]. SMOTE uses synthetic over-sampling of the minority class (vulnerable functions/files) and a random under-sampling of the majority class (neutral functions/files). In our study, we have over-sampled vulnerable functions/files by 200% and under-sampled neutral functions/files by 200%.

5. RESULTS

In the subsections that follow, we present the results from the empirical evaluation of our metrics.

5.1 RQ1: Association

Question: *Is a function/file more likely to be fixed for a post-release vulnerability if:*

(a) it is near the attack surface or dangerous points?

(b) it has a higher probability of being traversed on a random walk from the attack surface?

In this question, we wanted to understand if the proximity and risky walk metrics are capable of explaining the reality of historical post-release vulnerabilities. A statistically significant association between the metrics and historical post-release vulnerabilities support the utility of these metrics as early warning indicators of vulnerability likelihood.

We found a statistically significant association between the proximity metrics and vulnerable functions in 15 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's d effect size evaluation was predominantly *medium* in FFmpeg and *large* in Wireshark. The association results were consistent at the file-level as well, with the metrics being associated, to a statistically significant extent, with vulnerable files in 14 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's d effect size was predominantly *small* in both FFmpeg and Wireshark.

At both the function- and file-level, the median values of proximity metrics collected from vulnerable functions/files were lesser than that collected from neutral functions/files. In other words, vulnerable functions/files tend to be near the attack surface of a software system and also other functions/files regarded as dangerous.

Vulnerable functions/files tend to be near the attack surface and/or dangerous points.

The association analysis also revealed a statistically significant association between the risky walk metric and vulnerable functions in 13 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's d effect size evaluation was predominantly *small* in both FFmpeg and Wireshark. The association results were consistent with the file-level as well, with the metric being associated, to a statistically significant extent, with vulnerable files in 15 of the 16 FFmpeg releases and in all releases of Wireshark. The Cohen's d effect size was predominantly *large* in both FFmpeg and Wireshark.

At both the function- and file-level, the median value of risky walk metric collected from vulnerable functions/files was higher than that collected from neutral functions/files indicating that vulnerable functions/files tend to have a higher probability of being traversed by a random walk starting at the attack surface.

Vulnerable functions/files have a higher probability of being traversed on a random walk from the attack surface.

5.2 RQ2: Prediction (Base)

Question: *Do proximity and risky walk metrics improve the performance of a base prediction model built with SLOC and coupling metrics?*

In this question, we wanted to understand if the proximity and risky walk metrics can be used in building a predictive model that can predict the likelihood of a function/file being vulnerable better than a base model built with SLOC and coupling metrics.

We begin the regression analysis by assessing the correlation between the different metrics: proximity, risky walk, SLOC, fan in, and fan out. We used the non-parametric Spearman's Rank Correlation Coefficient, ρ, to asses the correlation between metrics. We observed a very high positive correlation ($\rho \geq +0.97$) between the proximity metrics. The high correlation suggests that entry points, exit points, and dangerous points tend to be close to one another. As a consequence, only one of the three proximity metrics may be sufficient in explaining all three phenomena, rendering the other two metrics redundant. The correlation analysis also revealed a strong positive correlation ($\rho \approx 0.76$) between SLOC and fan out. The positive correlation between SLOC and fan out is understandable in that a function that has more SLOC is likely to have more function calls. We also observed a moderate negative correlation ($\rho \approx -0.65$) between fan out and the proximity metrics. All correlations were statistically significant with p-value < 0.05.

We chose to not remove the redundant features manually but to use regression analysis approaches that are capable of dealing with multicollinearity. We explored several parametric and non-parametric regression analysis approaches and found the random forest machine learning approach to perform the best. The model performance results presented here, and in RQ3, are from the random forest model. Furthermore, the performance metrics—precision, recall, and F_2-measure—presented here are the mean values of those obtained from the models using the cross-validation and next release validation approaches.

FFmpeg

File-level: The average values of precision, recall, and F_2-measure of the base model were 0.0467, 0.7938, and 0.1840, respectively. The random forest model outperformed the base model model with an average precision of 0.1138 (an increase of 143.47%) and average F_2-measure of 0.3196 (an increase of 73.69%). However, the average recall of the random forest model was 0.5753 (a decrease of 27.52% from that of base).

Function-level: The average values of precision, recall, and F_2-measure of the base model were 0.0114, 0.7200, and 0.0533, respectively. The random forest model outperformed the base model with an average precision of 0.0156 (an increase of 36.33%) and average F_2-measure of 0.0725 (an increase of 36.07%). However, the average recall of the random forest model was 0.5493 (a decrease of 23.71% from that of base).

Wireshark

File-level: The average values of precision, recall, and F_2-measure of the base model were 0.1147, 0.7574, and 0.3399, respectively. The random forest model outperformed the base model model with an average precision of 0.1841 (an increase of 60.53%) and average F_2-measure of 0.3646 (an increase of 7.25%). However, the average recall of the random forest model was 0.5250 (a decrease of 30.68% from that of base).

Function-level: The average values of precision, recall, and F_2-measure of the base model were 0.0245, 0.5259, and 0.1023, respectively. The random forest model outperformed the base model with an average precision of 0.0333 (an increase of 31.11%), average recall of 0.5991 (an increase of 13.93%), and average F_2-measure of 0.1294 (an increase of 26.55%).

The random forest model outperformed the base model in terms of F_2-measure at both function- and file-levels.

As seen above, lowering the granularity of the metrics from file-level to function-level makes finding the "needle in a haystack" an order of magnitude more difficult. While function-level prediction has its challenges, the benefits warrant the need for more research at function-level prediction. For instance, in FFmpeg, if a file is predicted as vulnerable, developers must audit 306 SLOC, on average, however, if a function is predicted as vulnerable, developers must audit 30 SLOC, on average–a considerable reduction in effort.

5.3 RQ3: Prediction (Prior)

Question: *How do the prediction models built with proximity and risky walk metrics compare with prior vulnerability prediction literature?*

While traditional bug prediction is related to this question, our recent work has shown that the best bug prediction models would perform poorly when predicting vulnerabilities [33]. Further discussion on this is in Related Work in Section 7. Furthermore, vulnerability data used in building the model is one of the most important aspects in vulnerability prediction [24]. There have been many studies [10, 48, 11, 8] that use warnings from static code analyzers as indicators of vulnerability in a file. While these studies show that there is a correlation between warnings from static code analyzers and vulnerabilities, the correlation is moderate at best. In our prediction models, we have used real-world vulnerabilities i.e. those that were publicly disclosed, acknowledged, and fixed by the development team. In contrast, a recent work by Scandariato et al. [41] used files marked as vulnerable based on warnings from a static code analyzer. While the precision and recall of the model was shown to be high (≥ 0.8), the response from the model was not the likelihood of a file being vulnerable but its likelihood of having a static analysis warning.

Prediction models from prior vulnerability prediction literature have operated at file-level [36, 29], component-level [37], or binary-level [54, 46]. We found only one vulnerability prediction model that attempted to predict vulnerabilities at a function-level [44]. As a consequence of the disparity in granularity, choice of study subjects, and vulnerability data used in building the model, a direct comparison of the performance of the models in terms of metrics like precision, recall, etc., may be unfair. However, since we have collected our metrics, and built the prediction models, at both the function- and file-levels, we have partially alleviated the limitation of direct comparison imposed by granularity. To ensure fairness, we compared our function- and file-level models with other function- and file-level models from prior literature, respectively. We also collected and compared the same metrics (e.g. precision, recall, etc.) that were used in the performance evaluation of the models from prior literature.

At the function-level, the random forest model, fitted to data from both FFmpeg and Wireshark, outperformed the logistic regression model proposed by Shin and Williams [44]. The model proposed by Shin and Williams [44] achieved an almost zero average false positive rate (FPR) but a considerably high average false negative rate (FNR). A low FPR and a high FNR suggests that the model may have marked almost all functions as neutral. The best of our random forest models achieved a considerably lower average FNR of 0.3610 (a 58.02% decrease from 0.86) while maintaining an acceptable level of average accuracy at 0.8995 (a 4.31% decrease from 0.94).

Our file-level random forest model, fitted to data from both FFmpeg and Wireshark, outperformed the file-level prediction model proposed by Theisen et al. [46]. The best of our random forest models had an average recall that was considerably higher at 0.5796 (a 1059.24% increase from 0.05), the precision was 0.1984 (a 71.24% decrease from 0.69).

However, in vulnerability prediction we prefer higher recall over a higher precision [31]. Furthermore, the model proposed by Theisen et al. was built using only those files that ever appeared on stack traces from system crashes. There may be files with latent vulnerabilities (see [26]) that may have never crashed but their model does not consider these files. Putting such a model into operation means that the system must be in production, and potentially vulnerable, for a long time to get the stack trace data in the first place.

The file-level random forest models, fitted to data from both FFmpeg and Wireshark, outperformed the component-level prediction model proposed by Gegick et al. [9]. The CART model proposed by the authors achieved a recall of 0.57 but suffered a FPR of 0.48. The best of our random forest models had an average recall of 0.5796, which is similar to the model proposed by Gegick et al., however, the average FPR was 0.0876 (a 81.37% decrease).

In summary, both the function-level and file-level prediction models proposed in our work outperformed comparable models from existing vulnerability prediction literature. The true value of our metrics is in providing fine-grained, actionable, and interpretable intelligence about potential security risks that tend to vary with everyday changes to the source code. Furthermore, while performance metrics such as precision and recall provide a common ground to compare models, they fail to capture the nuances of the model building process.

The random forest model, at both function- and file-levels, outperformed comparable models from prior literature.

6. LIMITATIONS

Our empirical analysis is based on historical vulnerabilities, which are by no means comprehensive. Thus, many vulnerabilities may exist in our systems that have not been found. This is a common limitation in empirical security research, and is the reason we use the word “neutral” instead of “not vulnerable”.

A call graph is only an approximation of the system's function calls because ensuring all possible paths of control flow are represented is, at best, time consuming, and, at times, impossible. Program analysis researchers [12, 13] have proposed several call graph construction algorithms that produce call graphs with varying levels of precision. Researchers have used missing functions and/or function calls when comparing two call graphs [17] or two call graph generation tools [35]. In our study, we conducted manual inspection of the call graph to ensure its accuracy, and we suggest some added heuristics—number of fragments and monolithicity—to help users of our metrics understand how “close” they may be to getting as many graph edges and nodes as they will get.

The proposed metrics, especially the risky walk metric, depends on several parameters that, if not tuned properly, may result in poor risk analysis. We have conducted sensitivity analysis via parameter tuning across our models (See Appendix A). We found similar parameter values across our two case studies, indicating that our discovered parameters may generalize. Nonetheless, we recommend careful consideration be given to parameters when deploying these metrics.

Furthermore, the risky walk metric is sensitive to the weight assigned to edges in the call graph. While the sensitivity may seem to be a limitation of the metric, it indeed presents the users with an opportunity to configure the metric to better reflect the structure and history of a software system. For instance, in our empirical evaluation, we have chosen to increase the weight of edges terminating at historically vulnerable functions/files with the assumption that an attacker may attempt to start the reconnaissance by looking at such functions/files. We did, however, explore the impact of *not* weighting the edges terminating at historically vulnerable functions/files and found that the prediction models suffered a small (2.70% in FFmpeg and 7.84% in Wireshark) decrease in the average F_2-measure, while still outperforming the base model. The exploration revealed that the prediction performance of the models can be improved by assigning appropriate weights to the call graph edges thus allowing the users to customize the metric to their software systems.

7. RELATED WORK

The attack surface as a metaphor for risk is far from new. Michael Howard of Microsoft proposed the idea of quantifying security of a software system by measuring its *attack profile* [14]. Michael Howard's proposal was to reduce the attack profile of a product by having only the most commonly used features enabled by default. He introduced the notion of *attackability* of a product as a measure of its exposure to an attack. He computed the Relative Attack Surface Quotient (RASQ) to compare the *attackability* of seven versions of the Windows operating system to assess the relative security between them. The notion of *attackability* was redefined by Howard et al. along three dimensions: targets and enablers, channels and protocols, and access rights [15]. Howard et al. also proposed a formal method of measuring attack surface of software system in terms of its attack vectors (features that may be used in an attack).

The granularity of attack surface measurement was lowered from a system-level to a design-level by defining entry points and exit points to identify resources that compose the attack surface [22]. The notion of *size* of the attack surface emerged as measured by number of entry and exit points. We propose the refinement of the attack surface metrics by extending them to individual functions (and files) and taking into account the structure as they connect to the system's entry and exit points.

In addition to interpreting the attack surface as being the outer shell of a system, we could also consider all resources “exposed” through the surface as being part of the attack surface as well. Younis et al. [52, 51] used reachability analysis to assess the severity of a vulnerability and the probability that a vulnerability will be exploited. While this work used call graphs, they did not apply attack surface metrics to individual functions as we did. We also take dangerous system calls into account in our empirical analysis. Theisen et al. [46] used the attack surface metaphor to improve existing vulnerability prediction models. The authors have shown that approximating the attack surface of a software system using functions from stack traces improves the performance of existing vulnerability prediction models. The prediction models are at the source file and compiled binary levels. While their study focuses on prediction improvement, our study focuses on producing a more lightweight approach to collecting metrics that rely on the call graph and entry/exit points, and does not require a database of millions of stack traces from production usage data.

A vulnerability is a special kind of a software bug, one that has security consequences. Naturally, one may assume that bug/defect prediction models [49, 32, 53] may be used in vulnerability prediction. While an empirical connection has been observed [45], we have shown that bugs do not foreshadow vulnerabilities [33]. Thus, while the vulnerability prediction methods may resemble bug prediction, the models do not directly translate.

8. SUMMARY

The goal of this study is to assess security risk through an empirical understanding of the relationship between vulnerabilities, individual functions/files, and the attack surface of a software system. We proposed novel attack surface metrics—proximity and risky walk—defined on the call graph representation of a software system. Our empirical analysis revealed a statistically significant association with historical vulnerabilities (RQ1), and that prediction models outperformed a base prediction model built with SLOC and coupling metrics (RQ2). Prediction models that leverage our metrics, at both function- and file-levels, outperformed comparable models from prior vulnerability literature (RQ3). We envision the metrics to be beneficial to both researchers and practitioners as they are simple to collect, intuitive to understand, and flexible to apply.

In the future, we will explore the space of personalization and edge weighting schemes for the call graph as that affords an enormous opportunity for configuration and room for innovation. Future studies can also examine how these metrics fare on non-monolithic systems, such as APIs.

9. ACKNOWLEDGEMENTS

This research was sponsored by the U.S. National Security Agency Science of Security Lablet at North Carolina State University. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity. We thank the Research Computing Group at Rochester Institute of Technology for helping us migrate our analysis to the compute cluster.

10. REFERENCES

[1] K. Ali and O. Lhoták. *Application-Only Call Graph Construction*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[2] M. Bernaschi, E. Gabrielli, and L. V. Mancini. Operating System Enhancements to Prevent the Misuse of System Calls. In *Proc. 7th Conf. Computer and Communications Security*, pages 174–183, New York, NY, USA, 2000. ACM.

[3] U. Brandes and T. Erlebach. *Network Analysis: Methodological Foundations*, volume 3418. Springer-Verlag New York, Inc., 2005.

[4] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[5] I. Chowdhury and M. Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[6] J. Cohen. *Statistical power analysis for the behavioral sciences*. Academic press, 2013.

[7] A. Feldthaus, M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Efficient Construction of Approximate Call Graphs for JavaScript IDE Services. In *Proc. 35th Int. Conf. Software Engineering*, pages 752–761, May 2013.

[8] M. Gegick, P. Rotella, and L. Williams. Predicting Attack-prone Components. *Proc. Int. Conf. Software Testing Verification and Validation*, pages 181–190, 2009.

[9] M. Gegick, P. Rotella, and L. Williams. *Toward Non-security Failures as a Predictor of Security Faults and Failures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[10] M. Gegick and L. Williams. Toward the use of automated static analysis alerts for early identification of vulnerability- and attack-prone components. *Proc. 2nd Int. Conf. Internet Monitoring and Protection*, 2007.

[11] M. Gegick, L. Williams, J. Osborne, and M. Vouk. Prioritizing Software Security Fortification through Code-Level Metrics. *Proc. 4th workshop Quality of Protection*, pages 31–38, 2008.

[12] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call Graph Construction in Object-oriented Languages. In *Proc. 12th Conf. Object-oriented Programming, Systems, Languages, and Applications*, pages 108–124, New York, NY, USA, 1997. ACM.

[13] M. W. Hall and K. Kennedy. Efficient Call Graph Analysis. *Letters on Programming Languages and Systems*, 1(3):227–242, Sep 1992.

[14] M. Howard. Fending Off Future Attacks by Reducing Attack Surface. `http://msdn.microsoft.com/en-us/library/ms972812.aspx`, 2003.

[15] M. Howard, J. Pincus, and J. M. Wing. *Measuring Relative Attack Surfaces*. Springer, Boston, MA, 2005.

[16] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a Social Network or a News Media? In *Proc. 19th Int. Conf. on World Wide Web*, pages 591–600, New York, NY, USA, 2010. ACM.

[17] O. Lhoták. Comparing Call Graphs. In *Proc. 7th Workshop Program Analysis for Software Tools and Engineering*, pages 37–42, New York, NY, USA, 2007. ACM.

[18] S. Lipner. The Trustworthy Computing Security Development Lifecycle. In *Proc. 20th Computer Security Applications Conference*, pages 2–13, Dec 2004.

[19] P. Manadhata. *An Attack Surface Metric*. PhD thesis, Carnegie Mellon Univ., 2008.

[20] P. Manadhata, J. Wing, M. Flynn, and M. McQueen. Measuring the Attack Surfaces of Two FTP Daemons. In *Proc. 2nd Workshop Quality of Protection*, pages 3–10, New York, NY, USA, 2006. ACM.

[21] P. K. Manadhata, Y. Karabulut, and J. M. Wing. Report: Measuring the Attack Surfaces of Enterprise Software. In *Int. Symp. on Engineering Secure Software and Systems*, pages 91–100. Springer, 2009.

[22] P. K. Manadhata and J. M. Wing. An Attack Surface Metric. *Transactions on Software Engineering*, 37(3):371–386, May 2011.

[23] P. Massa and P. Avesani. Trust-aware Recommender Systems. In *Proc. 1st Conf. Recommender Systems*, pages 17–24, New York, NY, USA, 2007. ACM.

[24] F. Massacci and V. H. Nguyen. Which is the Right Source for Vulnerability Studies?: An Empirical Analysis on Mozilla Firefox. In *Proc. 6th Int. Workshop Security Measurements and Metrics*, pages 4:1–4:8, New York, NY, USA, 2010. ACM.

[25] V. Mehta, C. Bartzis, H. Zhu, E. Clarke, and J. Wing. *Ranking Attack Graphs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[26] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *Proc. Int. Symp. Empirical Software Engineering and Measurement*, pages 65–74. ACM, Oct 2013.

[27] A. Meneely, A. C. R. Tejeda, B. Spates, S. Trudeau, D. Neuberger, K. Whitlock, C. Ketant, and K. Davis. An Empirical Investigation of Socio-technical Code Review Metrics and Security Vulnerabilities. *Proc. 6th Int. Workshop Social Software Engineering*, pages 37–44, 2014.

[28] A. Meneely and L. Williams. Secure Open Source Collaboration: An Empirical Study of Linus' Law. In *Proc. 16th Conf. Computer and Communications Security*, pages 453–462, New York, NY, USA, 2009. ACM.

[29] A. Meneely and L. Williams. Strengthening the Empirical Analysis of the Relationship Between Linus' Law and Software Security. In *Proc. Int. Symp. Empirical Software Engineering and Measurement*, pages 9:1–9:10, New York, NY, USA, 2010. ACM.

[30] A. Meneely, L. Williams, W. Snipes, and J. Osborne. Predicting Failures with Developer Networks and Social Network Analysis. In *Proc. 16th Int. Symp. Foundations of Software Engineering*, pages 13–23, New York, NY, USA, 2008. ACM.

[31] T. Menzies, J. Greenwald, T. Menzies, A. Dekhtyar, J. Distefano, and J. Greenwald. Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'". 33:7–10, Nov 2007.

[32] T. Menzies, Z. Milton, B. Turhan, B. Cukic, Y. Jiang, and A. Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[33] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project. *Empirical Software Engineering*, pages 1–43, 2016.

[34] N. Munaiah and A. Meneely. Attack Surface Meter. `https://github.com/andymeneely/attack-surface-metrics`. Accessed: 2016-01-31.

[35] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An Empirical Study of Static Call Graph Extractors. *Transactions on Software Engineering and Methodology*, 7(2):158–191, Apr 1998.

[36] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting Vulnerable Software Components. In *Proc. 14th Conf. Computer and Communications Security*, pages 529–540, New York, NY, USA, 2007. ACM.

[37] V. H. Nguyen and L. M. S. Tran. Predicting Vulnerable Software Components with Dependency Graphs. In *Proc. 6th Int. Workshop Security Measurements and Metrics*, pages 3:1–3:8, New York, NY, USA, 2010. ACM.

[38] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, Nov 1999.

[39] M. Pinzger, N. Nagappan, and B. Murphy. Can Developer-module Networks Predict Failures? In *Proc. 16th Int. Symp. Foundations of Software Engineering*, pages 2–12, New York, NY, USA, 2008. ACM.

[40] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[41] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting Vulnerable Software Components via Text Mining. *Transactions on Software Engineering*, 40(10):993–1006, Oct 2014.

[42] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. M. Wing. Automated Generation and Analysis of Attack Graphs. *Symp. on Security and Privacy*, pages 273–284, Jan 2002.

[43] 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*, 37(6):772–787, Nov 2011.

[44] Y. Shin and L. Williams. An Empirical Model to Predict Security Vulnerabilities Using Code Complexity Metrics. In *Proc. 2nd Int. Symp. Empirical Software Engineering and Measurement*, pages 315–317, New York, NY, USA, 2008. ACM.

[45] Y. Shin and L. Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.

[46] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams. Approximating Attack Surfaces with Stack Traces. In *Proc. 37th Int. Conf. Software Engineering*, pages 199–208, Piscataway, NJ, USA, 2015. IEEE.

[47] J. J. Treinen and R. Thurimella. *Application of the PageRank Algorithm to Alarm Graphs*, pages 480–494. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[48] J. Walden and M. Doyle. SAVI: Static-Analysis Vulnerability Indicator. *IEEE Security & Privacy*, 10(3):32–39, May 2012.

[49] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[50] R. S. Wills. Google's PageRank: The Math Behind the Search Engine. *The Mathematical Intelligencer*, 28(4):6–11, 2008.

[51] A. A. Younis and Y. K. Malaiya. Using Software Structure to Predict Vulnerability Exploitation Potential. In *Proc. 8th Int. Conf. Software Security and Reliability-Companion*, pages 13–18. IEEE, Jun 2014.

[52] 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 *Proc. 15th Int. Symp. High-Assurance Systems Engineering*, pages 1–8. IEEE, Jan 2014.

[53] T. Zimmermann and N. Nagappan. Predicting Defects Using Network Analysis on Dependency Graphs. In *Proc. 30th Int. Conf. Software Engineering*, pages 531–540, New York, NY, USA, 2008. ACM.

[54] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *Proc. 3rd Int. Conf. Software Testing, Verification and Validation*, pages 421–428, Apr 2010.

APPENDIX

A. PARAMETER TUNING

In this section, we describe the methodology used to tune the parameters (i.e. personalization vector, damping factor, and edge weights vector) for the risky walk metric. The objective is to explore parameter values that, when used to compute the risky walk metric, enables a clear delineation of functions that were fixed for a historical post-release vulnerability (i.e. historically vulnerable functions/files) from those that were not (i.e. neutral functions/files). Although the label "historically vulnerable" may seem similar to the term "vulnerable" introduced in Section 4.3, there is a key difference in usage. To understand the difference, consider a sequence of chronologically ordered FFmpeg releases with version numbers 1.0.0, 1.1.0, 1.2.0, and 1.2.1. All functions/files fixed for a vulnerability in 1.0.0 and 1.1.0 are considered *historically vulnerable* in 1.1.0, whereas, all functions/files fixed for a vulnerability in 1.2.1 are considered *vulnerable* in 1.2.0. Furthermore, the set of vulnerable functions/files and the set of historically vulnerable functions/files do not intersect in any given release.

Our overall approach for parameter tuning is a brute-force exploration of parameter values along an exponential scale. Collectively, we examined the following seven variables:

> Damping factor (α), personalization of entry points (P_{entry}), personalization of exit points (P_{exit}), weight of call edges (W_{call}), weight of return edges (W_{return}), additive weight for edges terminating at dangerous points ($Aw_{dangerous}$), and additive weight for edges terminating at historically vulnerable functions/files ($Aw_{vulnerable}$)

For the personalization vector, the attack surface metaphor argument says that an attacker is likely to start the reconnaissance for an attack at either an entry point or an exit point (e.g. via fuzz testing techniques). We capture this behavior by having the personalization vector contain *higher probability* for entry and exit points than non-entry and non-exit points. For the non-entry and non-exit points, the personalization vector contains probability drawn from a uniform distribution.

For the damping factor, tailoring the input to the system is the only way for an attacker to affect the flow of control through the call graph. We could conceive situations where explorations would end quickly or after a long time, so in our parameter tuning we considered a wide range of values.

For assigning weights to edges, we considered four types of edges: calls, returns, edges to dangerous points, and edges to historically vulnerable functions/files. We assigned all call and return edges a base weight. A function/file that makes dangerous system calls or was historically vulnerable could be potential targets for an attacker. We capture this behavior by increasing the weight of edges terminating at such functions/files. Weights for edges become probabilities by summing them per node and dividing each by the total for that node.

We defined a set of candidate values for each of the seven variables and constructed a collection of 7-tuple permutations of the candidate values. The range of candidate values for each of the seven variables were: (a) α from 0.1 to 0.9, (b) P_{entry} and P_{exit} from 1 to 1,000,000, (c) W_{call} and W_{return} from 10 to 10,000, and (d) $Aw_{dangerous}$ and $Aw_{vulnerable}$ from 10 to 1,000. While the candidate values for α were on a linear scale (with an interval of 0.1), the candidate values for the remaining variables were on an exponential scale. Although the candidate values for the personalization variables (P_{entry} and P_{exit}) are not probabilities, they are transformed into probabilities before being used in the algorithm.

The total number of permutations in the collection was 63,504. We used an iterative approach to evaluate each permutation to obtain a set of values for the PageRank parameters, which when used in the computation of the risky walk metric, results in the metric being statistically significantly associated (p-value ≤ 0.05) with historically vulnerable functions and have the largest effect size evaluation. We used the non-parametric Mann-Whitney-Wilcoxon (MWW) test to assess the association and Cohen's d effect size statistic [6] to assess the effect size. The process was repeated in all 16 releases of FFmpeg and 7 releases of Wireshark. The permutation that resulted in risky walk having the largest average value of Cohen's d when aggregated across releases in each subject was chosen to compose the PageRank parameters. The highest ranking value of the parameters in FFmpeg and Wireshark is presented in Table 2.

Table 2: Highest ranking value of the variables that compose the PageRank parameters in FFmpeg and Wireshark

Variable	Subject	
	FFmpeg	**Wireshark**
α	0.9	0.9
P_{entry}	1	10,000
P_{exit}	1	10,000
W_{call}	10	100
W_{return}	10	10
$Aw_{dangerous}$	10	10
$Aw_{vulnerable}$	1,000	1,000

To avoid over-fitting the parameters, we also ran a sensitivity analysis of our prediction question (RQ2, in Section 5.2) by using the average of the parameter values of the top 100 highest ranking permutations ordered by the average Cohen's d (aggregated across releases). The final precision and recall was within 2.88% of those obtained from the model with the averaged weights.