Approximating Attack Surfaces with Stack Traces

Christopher Theisen[†], Kim Herzig[‡], Patrick Morrison[†], Brendan Murphy[‡], Laurie Williams[†]
crtheise@ncsu.edu, kimh@microsoft.com, pjmorris@ncsu.edu, bmurphy@microsoft.com, williams@csc.ncsu.edu

[†]Department of Computer Science, NCSU, Raleigh, North Carolina
[‡]Microsoft Research, Cambridge, UK

***Abstract* – Security testing and reviewing efforts are a necessity for software projects, but are time-consuming and expensive to apply. Identifying vulnerable code supports decision-making during all phases of software development. An approach for identifying vulnerable code is to identify its *attack surface*, the sum of all paths for untrusted data into and out of a system. Identifying the code that lies on the attack surface requires expertise and significant manual effort. This paper proposes an automated technique to empirically approximate attack surfaces through the analysis of stack traces. We hypothesize that stack traces from user-initiated crashes have several desirable attributes for measuring attack surfaces. The goal of this research is *to aid software engineers in prioritizing security efforts by approximating the attack surface of a system via stack trace analysis.* In a trial on Windows 8, the attack surface approximation selected 48.4% of the binaries and contained 94.6% of known vulnerabilities. Compared with vulnerability prediction models (VPMs) run on the entire codebase, VPMs run on the attack surface approximation improved recall from .07 to .1 for binaries and from .02 to .05 for source files. Precision remained at .5 for binaries, while improving from .5 to .69 for source files.**

***Index Terms*—stack traces, security, vulnerability, models, testing, reliability, attack surface.**

I. Introduction

Howard et al. [17] introduced the concept of an attack surface, describing entry points to a system that might be vulnerable along three dimensions: targets and enablers, channels and protocols, and access rights. Later, Manadhata and Wing [18] formalized the notion of attack surface, including methods, channels, untrusted data, and a direct and indirect entry and exit point framework that identifies methods through which untrusted data passes. Security professionals can focus their efforts on code on the attack surface because it contains vulnerabilities that are reachable, and therefore exploitable, by malicious users. Code not on the attack surface may contain latent vulnerabilities, but these are unreachable by malicious users. With this prioritization, the security professional could find vulnerabilities more efficiently.

As valuable as the concept of attack surface is, we still lack a practical means of identifying the parts of the system that are contained on the attack surface. Manadhata and Wing proposed metrics to measure attack surface size, and empirically measured direct entry points by building a call graph, but they left identifying indirect entry and exit points for future work. Other approaches to defining the attack surface have been done at a configuration level without considering code, and other code using API calls have required significant manual work [18].

We propose *attack surface approximation*, an automated approach to identifying parts of the system that are contained on the attack surface through stack trace analysis. We parse stack traces, adding all code found in these traces onto the attack surface approximation. By definition, code that appears in stack traces caused by user activity is on the attack surface because it appears in a code path reached by users.

We hypothesize that stack traces from user-initiated crashes have three desirable attributes for measuring attack surfaces: (a) they represent user activity that puts the system under stress; (b) they include both direct and indirect entry points; and (c) they provide automatically generated control and data flow graphs. We seek to assess the degree to which these attributes of stack traces support the identification of attack surfaces. We call our approach attack surface *approximation* because code entities will only be added to the attack surface when a crash has occurred. As such, the attack surface approximation will evolve over time. We assess our approach by analyzing the percentage of actual reported vulnerabilities in the code and whether they occur in our approximated attack surface.

The goal of this research is *to aid software engineers in prioritizing security efforts by approximating the attack surface of a system via stack trace analysis*.

We explore the following questions as a part of this paper:

RQ1: How effectively can stack traces to be used to approximate the attack surface of a system?
RQ2: Can the performance of vulnerability prediction be improved by limiting the prediction space to the approximated attack surface?

We build an attack surface approximation for the Windows operating system based on stack traces. This system is a completely automated process, requiring no human input to what is considered on the attack surface and what is not, unlike previous approaches requiring human input. To assess our approximation, we compared the set of known security vulnerabilities from earlier versions of Windows 8 against this attack surface.

The contributions of this paper include:

- A practical, automated attack surface approximation based upon analysis of stack traces
- An evaluation of attack surface approximation performance for vulnerability prediction in Windows 8
- Visualizations of the attack surface approximation for use during security reviews to find security-related relationships between code

The rest of the paper is organized as follows: Section II discusses background and related work, Section III discusses the stack trace data sources, Section IV presents our methodology, Section V discusses vulnerability prediction model (VPM) construction for our evaluation, Section VI presents our results and a discussion of these results, Section VII discusses our lessons learned and challenges, Section VIII presents limitations and threats to validity, and Section IX discusses future work.

II. Background and Related Work

Vulnerabilities can be seen as a special case of software defects [1]. Vulnerabilities tend to be sparser than general software defects [41], as not all defects may allow an attacker to gain anything. In this section, we provide a brief overview of related work.

A. Attack Surface

As mentioned previously, Howard et al. [17] provided a definition of attack surface using three dimensions: targets and enablers, channels and protocols, and access rights. Not all areas of a system may be directly or indirectly exposed to the outside. Some parts of a complex system, e.g. Windows OS, may be for internal use only and cannot be reached or exploited by an attacker. For example, installation routines are left in the system after initialization, but they are never accessed again and are unlikely to have security implications for the system.

Knowing the attack surface of a piece of software supports decision-making during all phases of software development. To date, approaches to empirical measurement of attack surfaces have relied on manual effort or on alternative definitions of 'attack surface'. Tools like Microsoft's Attack Surface Analyzer[1] determine where potential input vectors exist on a system. However, this tool currently focuses on delivered systems that are code-static; it detects *configuration* changes, not *code* changes.

Manadhata et al. [43] describe how an attack surface might be approximated by looking at API entry points. However, this approach does not cover all exposed code, as the authors mention. Specifically, internal flow of data through a system could not be identified. While the external points of a system are a useful place to start, they do not encompass the entirety of exposed code in the system. These intermediate points within the system could also contain security vulnerabilities that the reviewer should be aware of. Further, their approach to measuring attack surfaces required expert judgment and manual effort.

B. Exploiting Crash Reports

The use of crash reporting systems, including stack traces from the crashes, is becoming a standard industry practice[2] [24][26]. Bug reports contain information to help engineers replicate and locate software defects. Liblit and Aiken [20] introduced a technique automatically reconstructing complete execution paths using stack traces and execution profiles. Later, Manevich et al. [21] added data flow analysis information on Liblit and Aiken's approach. Other studies use stack traces to localize the exact fault location [22][23][24]. Lately, an increasing number of empirical studies use bug reports and crash reports to cluster bug reports according to their similarity and diversity, e.g. Podgurski et al. [25] were among the first to take this approach. Other studies followed [26][27]. Not all crash reports are precise enough to allow for this clustering. Guo et al. [28] used crash report information to predict which bugs will get fixed. Bettenburg et al. [29] assessed the quality of bug reports to suggest better and more accurate information helping developers to fix the bug.

With respect to vulnerabilities, Huang et al. [30] used crash reports to generate new exploits while Holler et al. [31] used historic crashes reports to mutate corresponding input data to find incomplete fixes. Kim et al. [32] analyzed security bug reports to predict "top crashes"—those few crashes that account for the majority of crash reports—before new software releases.

However, we are not aware of any study parsing stack traces to identify and predict vulnerable source files or to measure vulnerability exposure of software artifacts.

C. Defect Prediction Models

The goal of *defect prediction models* (DPMs) is to identify code that is most likely to contain defects. For example, Nagappan and Ball [3] showed that code churn metrics can be used to predict defect density, the number of defects per line of code. Later, Zimmermann et al. [4] used code dependency information and network metrics to classify defect prone code. Other studies used change-related [1], developer-related [5], organizational [6], process [7], change dependency [8], and test [9] metrics to build DPMs. Hall et al. [10] presented a systematic literature review of DPMs and showed that model-building methodology impacts prediction accuracy. DPMs are the basis for Vulnerability Prediction Models.

D. Vulnerability Prediction Models

Similar to DPMs, Vulnerability Prediction Models (VPMs) predict code with the highest chance of containing vulnerabilities. Studying VPMs for Microsoft Windows, Zimmermann et al. concluded that vulnerabilities are not as

[1] http://www.microsoft.com/en-us/download/details.aspx?id=24487

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

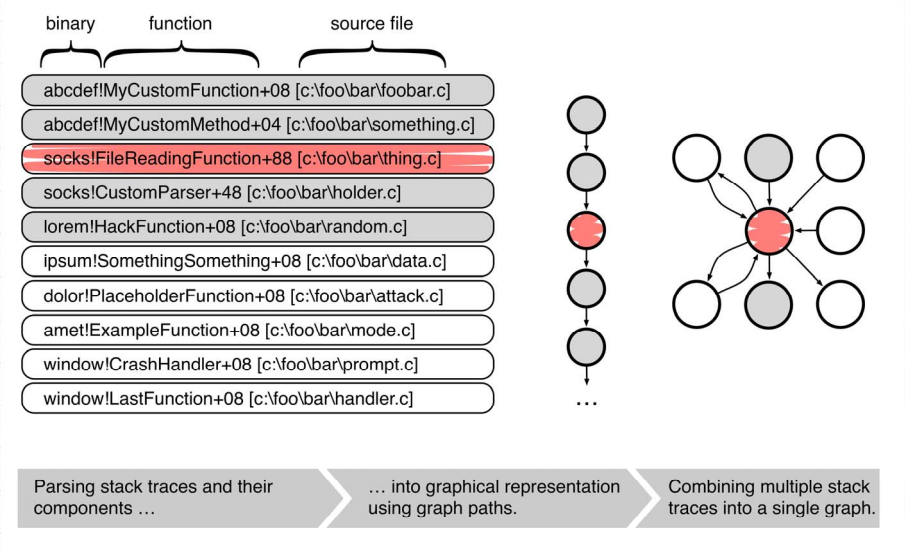

Figure 1 - Example of an anonymous stack trace. From the stack trace, we can identify binary, function, and sometimes source file information. Each of these stack traces can be transformed into a graphical representation. Each stack trace contributes one particular path (see middle) to the overall attack surface graph (see right hand side).

simple to predict as defects [11]. A study on Mozilla's Firefox web browser showed that, on Firefox, fault prediction model and the vulnerability prediction model provided similar prediction performance [12]. In general, many VPMs presented in literature are based on the same basic principles as DPMs, with adjustments to account for the relative rarity of security vulnerabilities. Transferring the concept of DPMs to VPMs, many studies use complexity, code churn, or static-alert measurements to predict vulnerabilities [13][14][15]. Published VPMs [16] have had challenges achieving precision and recall rates that DPMs have been able to achieve, presumably due to the relative rarity of vulnerabilities. As a result, VPMs may not be considered for practical use. Neuhaus et al. [16] used historical data on imports and functions. VPMs are used in this work to evaluate whether attack surface approximation improves the state of the art in vulnerability prediction.

III. Data Sources

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

A. Fuzzing Crashes

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

B. User Mode Crashes

Windows users generate user mode crashes. User mode crashes are field/customer crashes that are not due to hardware failures. Crashes in applications running without administrator rights triggers collection of crash information that may be sent to Microsoft. The system responsible for collecting the data, *Microsoft Error Reporting* [19], is executed under the same user privileges as the user that ran the crashed application. However, running in user mode limits the stack trace generator's ability to access and resolve which resources were involved in the crash. Thus, user mode crashes may only identify code areas that are accessible by the user in that mode.

C. Kernel Mode Crashes

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

D. Known Vulnerabilities

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

IV. Automated Approximation Methodology (RQ1)

Stack traces identify code that was loaded in memory at the time of the crash. Typically used for debugging, traces define the logical path between an external input and the system crash. Through the parsing of crashes, we can build a set of artifacts that users have had influence over. While bugs can appear in any artifact, the subset of code appearing in these stack traces has important security implications. Malicious users can only exploit vulnerabilities that they can access. If a bad check on incoming data could result in a buffer overflow attack in a specific function, but no outside user can ever pass input to that bad check, then that vulnerability has a lower priority unless code or configuration changes cause it to be exposed. By de-prioritizing code unlikely to be on the execution path, we can limit the scope of what security professionals have to review by having them focus on files that outside users can pass input to.

To identify artifacts that may be reached and thus exploited from users, we use stack traces of existing crashes to generate our data set. These stack traces identify entities that were executed just before the crash took place. Figure 1 shows an anonymized example stack trace describing a crash of Windows. Each line of the stack trace identifies code artifacts on the memory stack prior to the crash occurring. The order of lines and thus the order in which these artifacts occur represent the time sequence of events. The initial entry point is the top line item on a stack trace, and should in most cases be the first artifact touched by outside user input. Consecutive lines in a stack trace identify artifacts that called each other; if a function in foo.cpp calls a function in bar.cpp, we say that there is a direct neighbor relationship from foo.cpp to bar.cpp.

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

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

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

When doing mapping from stack traces to actual entities within the system, sometimes mappings are unable to be made. Two examples of this are when errors occur storing the stack trace, such as when the system is under duress, and mismatched names between the report to crash handlers and data about the system. When a mapping is unable to be made, we label that entity as "unknown," and do not place that entity on the attack surface.

For this work, we specifically remove hardware related crashes as errors resulting from such hardware failures do not indicate a potential input vector for potential attackers. The identification of hardware crashes is done by an automated stack trace classification system within Microsoft. Code that is inaccessible by user activity cannot be manipulated by an attacker, and therefore is not to be on the attack surface. The assumption carries forward when discussing our results below.

The ultimate output used by the development and security teams is a classification of whether an entity is on or off the attack surface. This classification can be used for prioritizing defect fixing and validation and verification efforts.

V. Vulnerability Prediction Model Construction Methodology (RQ2)

In addition to the classification scheme discussed in Section IV, we limit the prediction space to the approximated attack surface for Vulnerability Prediction Models (VPMs) to see if we get better prediction accuracy measurement when compared to VPMs that do not use attack surface information.

We replicated VPMs for Windows, as published by Zimmermann at al. [11]. The experiments conducted by Zimmerman et al. were conducted on datasets collected for Windows Vista, a product of significant size. Choosing the same product as the original study enables a comparison with the original study, giving insight in to how vulnerability prediction metrics in a codebase change over time.

The VPM described by Zimmermann et al. [11] is based on static code metrics and pre- and post-release vulnerabilities. As an analogue to the original study, we gathered these measurements using the CODEMINE process [33]. Microsoft developed CODEMINE to allow the company to monitor the development attributes of its products during development and after product release. The CODEMINE process provides a central repository of development and vulnerability metrics that were used within this research study.

A. Code Metrics

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

- *Churn metrics* [3]. Churn measures are relative to a time period; the period for all presented calculations is between the start and Release to Manufacturing (RTM) date of the project.
- *Complexity metrics* [38]. More complicated code is more likely to exhibit errors.
- *Dependency metrics* [13]. The degree to which a piece of code is depended upon, or depends upon other code, influences its impact on software vulnerabilities.
- *Legacy metrics.* Metrics of particular interest to Microsoft. The importance of security in the development of software at Microsoft began receiving increased attention after the Bill Gates' 2002 Trustworthy Computing Memo [34], with significant investments made in security training, tools, and process [2]. Code written after these processes were put in place has had a higher, more process-driven, level of attention to security applied in its design, construction and testing. These metrics are evidence for the theory that code written before the security reset may be more likely to contain vulnerabilities.
- *Size metrics.* Larger source files are more difficult to mentally manage, and, therefore, are more prone to defects and vulnerabilities.
- *Pre-Release vulnerabilities*. For VPMs predicting post-release vulnerabilities, we used pre-release vulnerabilities to model usual suspects.

Table 2, in the appendix, identifies all metrics used in the study and provides a description of each metric. Where noted, average, maximum and total values were taken for several of the metrics. Depending on the metric, data was available at either the source file level or at the function level. In cases where function level data was present, amounts were aggregated up to the file level via averages, totals and maximums. Binary-level data was obtained by aggregating source-file level data up to the binary in which each source file is used. This study uses additional metrics that were not available at the time of the original study by Zimmermann at al. [11]. The table identifies which metrics are common between the two studies and which are unique to this study.

All size, churn, complexity, and dependency metrics were measured as of each releases' Release to Manufacturing date.

B. Pre and Post-Release Vulnerabilities

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

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

C. Prediction Models

We ran our prediction models at both the binary and source level granularity. For both levels of granularity, we build classification models to identify code entities that had at least one vulnerability. For each level of granularity, we split the overall data collection into a training dataset that contains 2/3 of all data points. The remaining data is used for testing purposes. To split the data, we used stratified sampling—the ratio of code entities associated with vulnerabilities from the original dataset is preserved for both subsets. We repeatedly sampled the original dataset 100 times (100-cross-fold-validation). In total, we generated 200 independent training and testing sets: two levels of granularity and 100-cross folds each (similar to [4][11]).

We remove all code entities that are not part of the attack surface of Windows 8.1 from both training and testing sets before sampling takes place. Thus, classification models referring to attack surfaces contain less data points.

We conducted the experiments using the R statistical software [35] (Version 3.10). Instead of using the original feature vectors provided by the raw metric values, we applied R's prcomp [39] procedure to our data to produce principal components. Principal Component Analysis (PCA) [36] reduces redundancy in our matrix of metrics and observations by maximizing the variance of linearly independent variables. Deciding how many of these variables to use in model building typically takes one of two forms; either a limit on the number of terms in the model is set, or some total amount of variance to be accounted for by the model is set. We selected principal components that accounted for 95% of variance.

In pursuit of high prediction performance, we used Max Kuhn's R package caret [37] to build VPMs based on the components selected by PCA. We used a *Random Forest [40]* machine learning technique to generate our prediction. Random Forest is a variant of decision trees that can be represented as a binomial tree, and is popularly used for classification tasks. We chose Random Forest because this data set is highly unbalanced (many entities have no vulnerability, very few have some) and Random Forest is particularly good in handling unbalanced datasets.

Table 1 - DESCRIPTIVE STATISTICS FOR ATTACK SURFACE AT BINARY LEVEL, BROKEN DOWN BY TYPE OF STACK TRACE CATEGORY THAT IDENTIFIED THE BINARY.

	User mode	Kernel mode	Fuzzing	Kernel and user mode
%binaries	40.2%	7.1%	0.9%	48.4%
%vulnerabilities	66.7%	40.6%	14.9%	94.6%

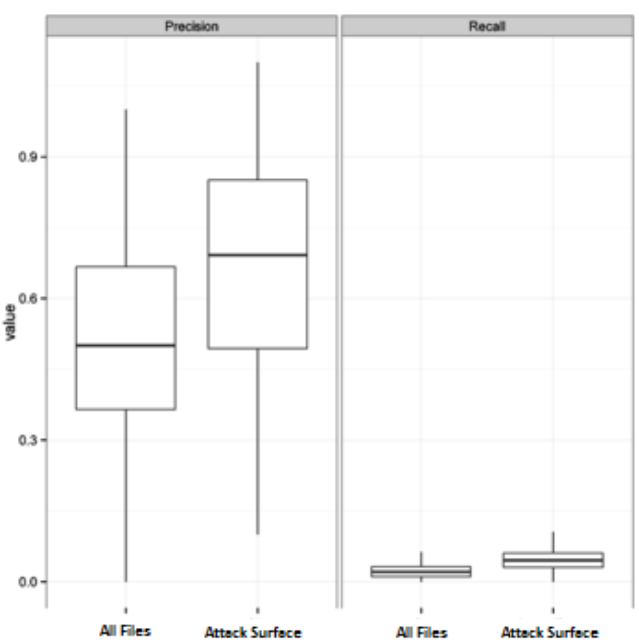

Figure 2 - Precision and recall of file level comparison of VPMs on all files and files on the attack surface approximation.

VI. RESULTS AND DISCUSSION

In this section, we present the data collected to address the research questions. After presenting the results as they related to our two research questions, we discuss how these results might benefit practitioners.

A. Attack Surface Approximation (RQ1)

Table 1 contains a summary of our Windows 8 attack surface approximation from our different data sources. We split our data sources into three separate columns, indicating user mode stack traces, kernel mode stack traces, and fuzz testing stack traces. The first row indicates the percentage of binaries seen in stack traces from that data source. The second row is the percentage of all shipped binaries that is included in row 1. For our purposes, shipped binaries are any binary that is included in a commercial release of the Windows product. For this example, the 32 binaries seen in fuzzing stack traces account for 0.9% of all shipped binaries. There can be overlap between multiple sources of stack traces. The %vulnerabilities row contains the percentage of all known vulnerabilities seen in that specific subset of binaries. Similar to the shipped binaries line, there can also be overlap on this row, which shows why

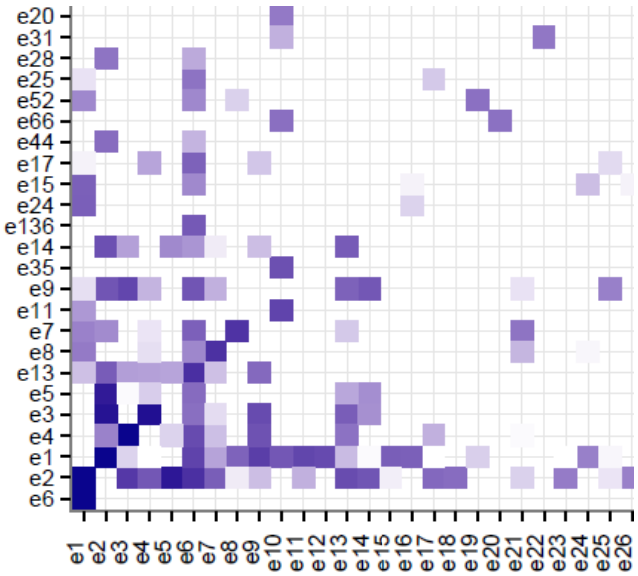

Figure 3 - Heatmap visualizing relationships between files on the attack surface and the frequency of these relationships.

the percentages add up to over 100%. The table shows that 48.4% of binaries contain 94.6% of the known vulnerabilities within Windows 8 when considering kernel and user mode crashes. Fuzzing crashes were left out of our combined column because it offered no improvement to our approximation. User and kernel mode covered all known vulnerabilities in the fuzzing dataset.

Based on this result, we suggest that this approach can reduce the amount of time spent inspecting code that currently cannot be exploited directly by malicious users. This time savings is critical when faced with incoming delivery deadlines and time crunches.

B. Vulnerability Prediction Models (RQ2)

We explore how limiting the prediction space to the approximated attack surface affects the accuracy of VPMs. As discussed earlier, VPMs predict where vulnerabilities might appear in a particular codebase based on software engineering metrics, such as code churn, code complexity, code dependencies, and pre-release vulnerabilities, among others. In Figure 2, we display the precision and recall of our VPM runs when considering the entire codebase and only code on our attack surface approximation. Compared with vulnerability prediction models (VPMs) run on the entire codebase, VPMs run on the attack surface approximation improved recall from .07 to .1 for binaries and from .02 to .05 for source files. Precision remained at .5 for binaries, while improving from .5 to .69 for source files. While these figures are low due to the scarcity of vulnerabilities, this represents improvement over the state of the art and is an open research area.

The initial results of applying the attack surface concept to VPMs are encouraging. We have demonstrated a statistical improvement in the accuracy of VPMs when the set of artifacts is narrowed down at both the binary and file level, and continual improvement based on better targeting of test data is

possible. By continually improving the accuracy of these VPMs, we hope to allay developer concerns about them and hope to see additional use of them to detect vulnerabilities.

C. Discussion

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

> "*Attack surface approximation would be useful to help target our review efforts.*"

This buy in on the concept of an attack surface being sound is an important step as we look to take the next steps in demonstrating the value of such tools.

One of the benefits of this approach is the high degree of automation possible for this process. The only software-specific element is the parsing of the individual stack traces. Everything else in this process is portable and could be applied to any other software project in development, though the accuracy and reliability of these results on different systems have not been tested as a part of this paper. By construction of our automated stack trace parser, we can provide an approximation of the attack surface of a system with some accuracy.

VII. Lessons Learned and Challenges

One of the difficulties with parsing stack traces is the fact that the problem reduces to string parsing to pull out code entities. While string parsing in itself is not a difficult task, stack traces from different parts of a system may be in different formats, and some stack traces contain different information. For example, during this research we found some traces only contained the file or function that was in the call stack. We then built a mapping tool to look up which binary these artifacts belonged to. In other systems with less mature software engineering tools and data, mapping missing code may not be as straightforward, though smaller systems may have more consistent stack traces.

We found that our fuzzing set of stack traces overlapped with the user and kernel datasets, finding only a few additional vulnerabilities. While there remained some overlap between user and kernel mode stack traces, together these sources are able to account for the majority of known vulnerabilities. Our results indicate that the attack surface approximation may be computed using only user and kernel crashes.

As this research progressed, we received a high amount of feedback related to the visualizations of this data and how it could be used to strengthen the security review process. In particular, graph visualizations of this data was of interest to security professionals within Microsoft. During this process, we devised a series of different graph formats for the data, including a graph representation of the entire system with single node entities, a centralized node example showing all of the neighboring relationships around that specific artifact, and the use of color to show the source of specific artifacts in stack traces. When these visualizations were shown to security professionals within Microsoft, they were received positively as a way to understand what the security-relevant relationships were between code entities.

Heatmaps are another possible way to demonstrate the relationships between code. A relationship means that one part of the codebase directly follows another in a stack trace, typically indicating a direct call from one function to another. An example heatmap is shown in Figure 3. In this example, files appear on the X and Y axis of the graph. A file on the Y-Axis of the chart has the files on the X-Axis of the chart directly follow it in a stack trace at least once if a purple box appears at that location. As the color of the box gets darker, that sequence is seen more often in the set of stack traces. This visualization can show practitioners at a glance which files have more appearances in the stack trace data and the variety in which different files appear around a specific file. For example, e6 is immediately followed by e1 in many stack traces, but no other files ever follow e6 in this dataset. One possibility is that this is crash handling code or a standard path taken during a crash. In contrast, e2 also appears frequently in stack traces, but a wide variety of files may immediately follow it in a trace. This visualization could help developers understand the context that a function or file operates in, without overwhelming the user with a graph of the entire system.

Graphs of these relationships may be useful in several ways. By displaying the graph of security-relevant relationships to security reviewers while they work, they can focus on known possible malicious entry points during review rather than checking all of the input edges into a function or file. An example of a graph is shown in Figure 1. By presenting this data to reviewers, they can make more informed decisions about the context in which a file or function is operating in. The data is similar to the heatmap data discussed above, presented in a different visual format.

These visualizations, in the view of the security professionals that the researchers spoke to, may have an impact on security reviewer productivity in the same way call graph displays can assist software developers.

VIII. Limitations and Threats to Validity

In this preliminary exploration of the research topic, we have only explored one specific software system. This approach may not be generalized to other systems without similar studies in different domains. In the absence of an oracle for the complete attack surface, we cannot assess the completeness of our approximation. Our determination of accuracy currently is based only on known vulnerabilities, which may introduce a bias towards code previously seen to be vulnerable. While this may be a good assumption, further exploration is needed. The set of artifacts set as part of the attack surface is an approximation, and we do not claim to capture all possible vulnerable nodes.

We have done our VPM analysis on different levels of abstraction of code; binary level and file level. In this paper, we present results reflecting both of these levels. Each of these levels of abstraction has different strengths and shortcomings. For example, while binary level allows us to make more accurate claims about where vulnerabilities might be, we are

painting with a broad brush. A single binary could contain hundreds of files that need to be looked at, which limits their usefulness. Running future analysis on file level and continuing to improve at that abstraction level is one way to mitigate this. By the reasoning above, our attack surface approximation only being on binary level is a limitation in the granularity of our approximation.

Finally, due to code or configuration changes, code that is not on the attack surface may be moved on to the attack surface. However, prioritization of code on the attack surface, using our method or other attack surface identification methods, can be used to reduce security risk.

IX. Future Work

We explore several different avenues for the continuation of this work. A further exploration of statistical data on frequency of appearance, and number of times a specific sequence of artifacts appear in user generated stack traces from crashes is one such avenue. By assigning weights to specific artifacts based on the frequency in which they appear in stack traces and in relationships with other artifacts, we may be able to improve VPM performance in detecting vulnerabilities if there is a relationship between these metrics and how often vulnerabilities appear. We hypothesize based on a cursory look at the data that this sort of exposure metric may be related to the discovery of security vulnerabilities.

We plan to explore the expansion of the graph visualization and parsing of the stack traces, as this area seems to hold the most promise. In particular, discovering particular flow patterns within the security graph representation of the system is of particular interest to the security professionals we spoke to as a part of this research. Empirical studies to show improvement in the efficiency of security reviews when using these visualizations is one avenue this research branch could take.

As mentioned in the limitations, these results cannot be generalized to all systems. Doing a similar analysis on other software systems, such as web applications, could give us insight into how generalizable this approach could be. Identifying how many stack traces are required to build a reasonably accurate approximation of the attack surface would be useful when trying to apply this technique to smaller software systems.

All current results can be replicated at a finer level of granularity. The attack surface approximation could be done at both the file and function level, and the VPMs could be run with a function level attack surface as well. To do this, we will need to resolve unmapped file and function data for some of the entities on our attack surface approximation.

X. Acknowledgements

This work was completed as part of a summer internship at Microsoft Research Cambridge UK in the summer of 2014. The researchers would like to thank the Windows Security Teams for their feedback, as well as the intern support staff at MSR Cambridge for their work. Funding for Laurie Williams for part of the advising of this proposal was provided by the National Security Agency. We also thank the Realsearch group at NCSU for their invaluable feedback.

XI. References

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

[2] M. Howard and S. Lipner, The Security Development Lifecycle, Microsoft Press, 2006.

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

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

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

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

[7] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[8] K. Herzig, S. Just, A. Rau and A. Zeller, "Predicting Defects Using Change Genealogies," in *Proceedings of the 2013 IEEE 24nd International Symposium on Software Reliability Engineering*, 2013.

[9] K. Herzig, "Using Pre-Release Test Failures to Build Early Post-Release Defect Prediction Models," in *Accepted at The 25th IEEE International Symposium on Software Reliability Engineering (ISSRE)*, Neaples, 2014.

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

[11] T. Zimmermann, N. Nagappan and L. Williams, "Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, 2010.

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

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

[14] Y. Shin, A. Meneely, L. Williams and J. Osborne, "Evaluating Complexity, Code Churn, and Developer

Activity Metrics as Indicators of Software Vulnerabilities," *Software Engineering, IEEE Transactions on,* vol. 37, pp. 772--787, 2011.

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

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

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

[18] P. Manadhata and J. Wing, "An Attack Surface Metric," *Software Engineering, IEEE Transactions on,* vol. 37, no. 3, pp. 371-386, 2011.

[19] "Description of the Dr. Watson for Windows," Microsoft Corporation, [Online]. Available: http://support.microsoft.com/kb/308538/en-us.

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

[21] R. Manevich, M. Sridharan, S. Adams, M. Das and Z. Yang, "PSE: Explaining Program Failures via Postmortem Static Analysis," in *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, Newport Beach, CA, USA, 2004.

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

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

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

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

[26] Y. Dang, R. Wu, H. Zhang, D. Zhang and P. Nobel, "ReBucket: A Method for Clustering Duplicate Crash Reports Based on Call Stack Similarity," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

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

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

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

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

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

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

[33] J. Czerwonka, N. Nagappan, W. Schulte and B. Murphy, "CODEMINE: Building a Software Development Data Analytics Platform at Microsoft," *Software, IEEE,* vol. 30, no. 4, pp. 64--71, 2013.

[34] "Four Grand Challenges in Trustworthy Computing," Computing Research Association, Warrenton, VA, 2003.

[35] R. D. C. Team, "R: A Language and Environment for Statistical Computing," 2010.

[36] K. Pearson, "LIII. On lines and planes of closest fit to systems of points in space," *Philosophical Magazine Series 6,* vol. 2, pp. 559-572, 1901.

[37] M. Kuhn, "caret: Classification and Regression Training," 2011.

[38] Emam, K.E., Melo, W., and Machado, J.C. The prediction of faulty classes using object-oriented design metrics. *J. Syst. Softw.*, 56 (feb 2001), 63–75.

[39] W.N. Venables and B.D. Ripley, Modern Applied Statistics with S. Fourth Edition. Springer, 2002

[40] L. Breiman, "Random Forests", Machine Learning 1, vol. 6., pp. 5-32, 2001.

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

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

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

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

XII. APPENDIX

Table 2 - **METRICS DEFINITIONS**

Metric	Category	Definition
Added LOC	Churn	Lines of code added during the development cycle.
ChurnedLOC	Churn	Lines of code added, deleted or altered during the development cycle.
DeletedLOC	Churn	Lines of code deleted during the development cycle.
Editors	Churn	Count of unique users who have made changes to each file
NumberOfEdits	Churn	Count of distinct commits made to each file
RelativeChurn	Churn	Relative code measure, ratio of Churned LOC to LOC Project End
Complexity (Avg, Sum, Max)	Complexity	McCabe complexity measure, number of linearly-independent paths through each function, rolled up to source-file level
Arcs (Avg, Sum, Max)	Dependency	Number of transfer of control points between basic blocks defined in the file
FanIn (Avg, Sum, Max)	Dependency	Number of calls to functions within each source file
FanOut (Avg, Sum, Max)	Dependency	Number of calls by functions within each source file to other functions
FanOutExternal (Avg, Sum, Max)	Dependency	Number of calls by functions within each source file to other functions outside the source file
Incoming Cross Binary	Dependency	Number of calls from outside a source file's binary to functions within the source file
Incoming Dependencies (Avg, Sum, Max)	Dependency	Number of function call, import, export, RPC, COM and Registry access dependencies on the source file
Outgoing Dependencies (Avg, Sum, Max)	Dependency	Number of function call, import, export, RPC, COM and Registry access dependencies by the source file
AddedLOCSinceReset	Legacy	Lines of code added since the security reset date
AgeinWeeks	Legacy	Number of weeks since recorded file creation date of source file
ChurnedLOCSinceReset	Legacy	Lines of code added, deleted or altered between the security reset date and RTM
ChurnSinceReset	Legacy	Lines of code deleted or altered between the security reset date and RTM.
DeletedLOCSinceReset	Legacy	Lines of code deleted between the security reset date and RTM
LegacyLOCPct	Legacy	Relative code measure, percentage of LOC written prior to reset
LOC Pre-Reset	Legacy	Total lines of code in the file on the security reset date
NumberOfEditsSinceReset	Legacy	Count of distinct commits made to each file since the reset
Arguments (Avg, Sum, Max)	Size	Number of function arguments defined within the file
Blocks (Avg, Sum, Max)	Size	Number of basic blocks contained within the file (a block is a single, contiguous set of instructions with one entry, one exit and no branches)
Functions	Size	Number of functions defined within the file
LOC Project End	Size	Total lines of code in the file at Release-To-Manufacturing (RTM)
LOC Project Start	Size	Total lines of code in the file at the start of the development cycle
Locals (Avg, Sum, Max)	Size	Number of local variables defined within the file
Paths (Avg, Sum, Max)	Size	Number of paths
VulnCount_PreRelease	Defects	Number of vulnerabilities fixed prior software release
VulnCount_SecurityReview	Defects	Number of security related changes prior software release