Attack Surface Prioritization with Crash Dump Stack Traces

Author’s Name, Affiliation  Author’s Name, Affiliation
Author’s Name, Affiliation  Author’s Name, Affiliation  Author’s Name, Affiliation

Abstract
Resource limitations often preclude security professionals from reviewing, testing, and fortifying an entire code base. Identifying metrics that enable prioritization of security efforts would help practitioners discover security issues more efficiently. Risk-Based Attack Surface Approximation (RASA) makes use of crash dump stack trace from a targeted software system to provide an estimated attack surface. In this paper, we extend the RASA approach, to develop a series of metrics that could help identify how the attack surface changes and if areas of the attack surface have more dangerous vulnerabilities. The goal of this research is to aid software engineers in approximating the attack surface of software systems by developing metrics based on crash dump stack traces. In this paper, we present the RASA approach and three metrics based on crash dump stack traces: change, complexity, and boundary metrics. We parsed 24.5 million stack traces from Windows 8, 8.1, and 10 for inclusion in our study. With change metrics, we help security professionals identify code that has fallen off or been added to the attack surface for the target system. For example, 58.7% of code that was seen on crash dump stack traces changed from Windows 8.1 to Windows 10. With complexity metrics, we measure fan-in and fan-out measures from crash dump stack traces to determine whether certain vulnerabilities are more impactful than others. With boundary metrics, we determine where the boundary of the software system is, or where crash dump stack traces indicate entry and exit points to the system might be. We determined that only 4% of vulnerabilities fixed for Windows 8.1 appeared on the boundary of the system.

1. Introduction
For security teams, prioritizing code to review and/or test can improve a team’s ability to find and remove vulnerabilities. Some organizations may choose to prioritize security efforts based on their perception or knowledge of whether the vulnerability is on the attack surface of a system. Researchers have suggested methods for identifying the attack surface of a system in an efficient manner for practitioners [1-4]. However, identifying code on the attack surface of a system is only a partial solution. Beyond the attack surface, how should security professionals prioritize efforts?

Many organizations already collect empirical metrics for their software systems, such as crash dump stack traces. In this work, we propose that organizations use this existing dataset to prioritize security efforts. In this paper, we generate metrics derived from crash dump stack traces to help practitioners further prioritize their efforts. We hypothesize that:

Crash dump stack traces could be used to approximate how the attack surface of a system changes over time. Crash dump stack traces can be collected over the lifespan of software systems. By comparing traces from different versions of software systems, security professionals can observe how the profile of crashing code has changed, and use that information to update their understanding of critical security points in the system. Similar analysis can be performed between different time periods of development.

Crash dump stack traces can identify flawed data paths in a software system, and identify which code artifacts carry more risk of being exploited. By providing fan-in (number of incoming calls to code) and fan-out (number of outgoing calls from code) measurements based on crash dump stack traces to security professionals, we can express whether a specific code artifact is more or less exposed to flawed data paths. Because we know that any crash dump stack trace indicates some sort of flaw, an increase in fan-in and fan-out numbers based on crash dump stack traces indicates potentially riskier code.

Finally, code on entry and exit points of a software system is more likely to contain security vulnerabilities than the rest of the codebase. While some researchers have focused on identifying entry and exit points, or boundary, to secure [5], others have advocated for a defense-in-depth approach to security [6]. Crash dump stack traces can identify the boundary of a software system by identifying where the traces change from external code to system code.
To evaluate these hypotheses, we pose the following research questions:

**RQ1:** How effectively can stack traces to be used to approximate the attack surface of a system?

**RQ2:** How does the code that is seen on crash dump stack traces drop off, get added, and stay the same across versions and during development of software systems?

**RQ3:** How are security vulnerabilities correlated with code complexity, as measured by fan-in and fan-out metrics?

**RQ4:** How often are vulnerabilities seen on the entry and exit points of a software system compared to the rest of the codebase?

We analyzed 24.5 million stack traces from crash dumps from Windows 8, 8.1, and 10 from 2014 and 2015, and developed metrics in three categories: change metrics, complexity metrics, and boundary metrics. We correlated these metrics with security vulnerabilities found in the Windows codebase over the same time period. This correlation was used to make observations about the effectiveness of each metric.

We include the following as contributions in this paper:

- An exploration of the change in attack surface over time, as determined by crash dump stack traces.
- Results from a case study indicating the impact of vulnerabilities in a software system and the complexity of specific file.
- A database scheme to facilitate repeatable analysis of crash dump stack traces against vulnerability locations.

The rest of the paper is organized as follows: Section 2 discusses background and related work, Section 3 presents our methodology, Section X discusses the specific case study presented in this work, Section X presents our results, Section X contains discussion of these results, Section X discusses our lessons learned and challenges, Section X presents limitations and threats to validity, and Section X discusses future work, and Section X concludes.

### 2. Related Work

Vulnerabilities are a special case of software defects [7]. Vulnerabilities tend to be sparser than general software defects [8], as not all defects may allow an attacker to gain anything. In this section, we provide a brief overview of related work.

### 2.1. Attack Surface

Howard et al. [9] provided the seminal 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, such as the Windows operating system, may be for internal use only and cannot be reached or exploited by an attacker.

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 Analyzer⁴ 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. [1] 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. Later, Younis et al. [10] analyzed the relationship between the attack surface of Apache HTTP Server and the density of vulnerabilities in the system. Munaiah et al. [4] used call graphs to determine the proximity of security vulnerabilities to the attack surface of the software system, and found that vulnerabilities tended to cluster near areas of the call graph considered on the attack surface of the system. Younis et al. [11] used reachability analysis alongside entry points to determine how exploitable specific vulnerabilities were, combining the concepts above. Theisen et al. [3] developed Risk-Based Attack Surface Approximation (RASA), which uses crash dump stack traces to estimate the attack surface of a target system.

### 2.2. Exploiting Stack Traces

The use of crash reporting systems, including stack traces from the crashes, is becoming a standard industry

---

practice\textsuperscript{2} [12, 13]. Bug reports contain information to help engineers replicate and locate software defects. Liblit and Aiken [14] introduced a technique automatically reconstructing complete execution paths using stack traces and execution profiles. Later, Manevich et al. [15] added data flow analysis information on Liblit and Aiken’s approach. Other studies use stack traces to localize the exact fault location [12, 16, 17]. 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. [18] were among the first to take this approach. Other studies followed [13, 19]. Not all crash reports are precise enough to allow for this clustering. Guo et al. [20] used crash report information to predict which bugs will get fixed. Bettenburg et al. [21] assessed the quality of bug reports to suggest better and more accurate information helping developers to fix the bug.

Crash dump stack traces have been used as an empirical metric for localizing security vulnerabilities. Theisen et al. [3] used crash dump stack traces from Windows to generate an approximation of the attack surface of Windows, called a risk-based attack surface approximation. In that study, 48.4% of binaries appeared on at least one crash dump stack trace, while 94.8% of post-release vulnerabilities were fixed in that 48.4% subset of binaries. Their result allows security professionals to prioritize security efforts somewhat, but half of the binaries in a software system is not a practical reduction in review for many software teams.

2.3. Security Vulnerabilities

With respect to vulnerabilities, Huang et al. [22] used crash reports to generate new exploits while Holler et al. [23] used historic crashes reports to mutate corresponding input data to find incomplete fixes. Kim et al. [24] analyzed security bug reports to predict “top crashes”—those few crashes that account for the majority of crash reports—before new software releases.

Massacci et al. [25] provided suggestions on how to select good systems for studies about vulnerabilities. Meneely et al. [26] explored how Linus’s law affected the generation of security vulnerabilities, and found correlations between more authors of a piece of code and an increase in vulnerabilities in that code. Meneely et al. [27] later strengthen this evidence by confirming the result for additional sources.

A variety of research has been focused on using various properties of software to target vulnerable code. scandario and Text mining techniques to predict vulnerable components. RASA also uses text mining techniques, specifically to parse crash dump stack traces to gain insights. Smith et al. [29] used SQL hotspots as a heuristic for identifying vulnerabilities. Theisen et al. [3] used crash dump stack traces as a heuristic for localizing security vulnerabilities.

Based on the corpus of research in this area, we believe that using empirical software measurements and heuristics is a promising direction for research for prioritizing code with security vulnerabilities. Making economically informed decisions on where security vulnerabilities might be could save organizations critical security effort man-hours and resources while finding vulnerabilities before they effect end users.

3. Risk-Based Attack Surface Approximation

Theisen et al. [3] developed RASA, which uses crash dump stack traces to estimate the attack surface of a target system. Stack traces from crash dumps represent user activity that put the system into an unexpected state. As indicated by prior research outlined in section 2.1, researchers have focused on the edge of systems when computing the attack surface. Computing the entire attack surface of a target system is labor-intensive. By using empirical metrics for approximation, RASA is able to measure the entire attack surface of the target system.

To compute the RASA approach for a target system, a collection of stack traces from crash dumps are collected from the software system we are analyzing. These stack traces are chosen from a set period of time. For each individual stack trace pulled from a crash dump, we isolate the binary, file, or function on each line of each stack trace, and record what code artifact was seen and how many times it has been seen in a stack trace. Each of the code artifacts from stack traces should then be mapped to a code artifact in the system. For example, if the file foo.cpp appears in a stack trace, the matching foo.cpp in system should be identified. A software system may have multiple foo.cpp files, so a method for identifying which foo.cpp was in the crash is required. A list of code artifacts in a software system could come from toolsets provided by the company maintaining the system or pulled directly from source control, in the case of open source projects.

Next, RASA parses each individual stack trace in the dataset, and sequentially extracts the individual code artifacts that appear on each line of the trace. To tie stack trace appearances to the codebase, RASA gener-
Formalizing a database schema for the storage of stack traces from the target software system is the next step for an experimental setup, as proper indexes on these tables can then be created for faster queries and faster experimentation. The full database diagram for our collection of crash dump stack traces can be seen in Figure 2. The starting point for the database design is the “StackTraces” table, which contains all of the stack traces that are collected for these experiments. One individual record in the “StackTraces” table represents a single line of a single stack trace, with individual stack traces having a unique “stack_threat_id.” As an example, an individual stack trace with 22 lines in it would have 22 records in “StackTraces”, with a unique “stack_id” for each line and a single “stack_threat_id” for the 22 records. Crashes can contain multiple stack traces from different threads that are running at the time of the crash, so each stack trace record is mapped to a specific crash, organized in the “Crashes” table. Having separate tables for crashes and stack traces allows users to look for potential correlations with stack trace types that frequently appear together in the same crash. Each “Crashes” record is mapped to a specific version of Windows, itemized in the “Products” table. For both the “StackTraces” to “Crashes” mapping and the “Crashes” to “Products” mapping, a matching “Crashes” record or “Products” record must be assigned for a new “StackTraces” or “Crashes” record to be created. Because of this, we add each crash to the database first, then add the associated stack traces to the database.

In addition to mapping stack traces and crashes to specific products, we also map individual code artifacts at the binary, file, and function level to specific artifacts in versions found in the “Products” table. Each level of granularity of code artifacts has its own associated table: “Binaries”, “Files” and “Functions”, respectively. Each of these artifacts is given its own unique identifier and is mapped to a specific version of Windows in the “Products” table. Similar to the “Crashes” mapping for the “StackTraces” table, new stack traces placed into “StackTraces” must have a valid entry in the “Binaries”, “Files”, and “Functions” tables. This mapping is performed by using string parsing to determine what the correct name is for each line in the stack trace.

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

Each level of granularity has a frequency table, specified as “BinaryFrequency”, “FileFrequency”, and “FunctionFrequency.” Each table has a unique entry for each code artifact for each product. For each record, we then track three different values, used in our later metric generation step. “Edge_count” tracks how many unique incoming and outgoing edges a specific instance of a code artifact has in a list. For example, if “foo.cpp” has 5 unique ways to call into a file, then “edge_count” would have a value of 5 for the incoming entry in the list. “Crash_count” tracks how many times a code artifact appears in a unique crash. For our “foo.cpp” example, appearing in 8 different crashes would mean the “crash_count” value would be set to 8. Note that this is unique occurrences on crashes; if “foo.cpp” appears 6 times in a single crash, it would only contribute one additional occurrence of “foo.cpp” to the “crash_count” entry. Finally, “stackLine_count” is a count of the total number of times a code artifact appears in our datasets, with multiple occurrences in the same crash adding multiple entries. In our previous example of 6 occurrences of “foo.cpp” in a single crash, that would result in adding 6 to the “stackLine_count” entry for “foo.cpp.”

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

5. Attack Surface Metrics

While a measure of the code that could potentially contain vulnerabilities could be a useful metric for developers, we also explore additional types of metrics that could be gathered from crash dump stack traces.

We define three new metrics identified from crash dump stack traces that software developers could use to improve maintenance efforts in codebases: change, complexity, and boundary metrics. All of the metrics can be measured at any of three levels of granularity (binary, file and function), depending on need. Stack traces from crash dumps typically contain at least one of these levels of granularity.

5.1. Change Metric

Determining how software systems change over time is important for security professionals, as many security vulnerabilities are introduced as software changes. As new features are added by engineers to satisfy new requirements, vulnerabilities could be introduced alongside those new features, unbeknownst to the developers. While security reviewers could catch many potential vulnerabilities as these changes are flagged and inspected in source control systems, this process is still a very manual one for many organizations. For example, a common workflow for many organizations is to have developers flag changes they make as potentially security relevant, so security teams can then review the code. However, developers may not realize that a change they are making has security implications, as many developers have no background in security work. Therefore, introducing methods for automatic review of potential security issues would be desirable.

To that end, change metrics represent how much the crash data has changed between two approximations. Determining what binaries, files, and functions have changed from version to version will help security professionals focus efforts on code that has been newly introduced to the attack surface of a software system. Code that is newly appearing on the attack surface will typically have not been reviewed as stringently by the security team, if it was reviewed by the security team at all. Future iterations of this metric could include month to month changes as well, with the inclusion of dates on specific crashes.

We track the amount of code that appears in different versions of Windows to calculate our change metrics. Our three metrics are defined as follows:

First Only (FO), is defined as the code artifacts that only appear in the earlier (older version, previous time period) part of the comparison, and is expressed in equation 3:
Second Only (SO), is defined as the code artifacts that only appear in the later (newer version, next time period) part of the comparison, and is expressed in equation 4:

\[ SO = \text{code in second part of comparison} \]

In Both (IB), is defined as the code artifacts that appear in the both parts of the comparison, and is expressed in equation 5:

\[ IB = \text{code in both parts of comparison} \]

These metrics are generated using data from the appropriate time and version of the target software system. For example, if you wanted to compare the attack surface of a system in Windows 8 versus Windows 8.1, you would collect crash dump stack traces from those two versions, create RASA for both, and compare the results using the above metrics. A similar approach is used to compare different time periods for products under development, such as 2014 and 2015 for Windows 10.

5.2. Complexity Metric

Not only do stack traces contain the individual code artifacts that appeared in a crash, they also preserve the order in which that code was accessed leading up to the crash. This order can indicate the complexity of individual code artifacts that appear in failure cases of the software. We can determine, for specific code artifacts, how many different code entities immediately proceed that entity in crashes, and how many code entities immediately follow that entity in crashes. The count of how many different code entities immediately proceed a code entity in crashes is referred to as a “fan-in measure,” while the count of how many code entities immediately follow a code entity in crashes is referred to as a “fan-out measure.” Using these “fan-in” and “fan-out” measures as a measure of the complexity of specific code artifacts could be useful for software developers to identify points of failure in the software system. Additionally, these measures of complexity could be used to provide an estimation of the overall impact of specific security vulnerabilities. Code artifacts with higher “fan-in” and “fan-out” values may need to be prioritized for vulnerability fixes, as the greater span of possible flows through that code artifacts could indicate vulnerabilities with more severe consequences for the software system.

Therefore, the complexity metric is based on the fan-in and fan-out measures for a code entity. As an example, if binaries X, Y, and Z appear directly after binary A in stack traces, binary A therefore has three “outgoing” edges. The frequency in which a specific entity appears after another can also be measured.

We report the ratio of the number of times each permutation of complexity appears in the codebase, and how many of those complexity types have security vulnerabilities associated with them. As an example, if one complexity type has 100 occurrences, and there are 30 vulnerabilities in those occurrences, then we claim that complexity type has a 30% rate of vulnerability. These ratios are defined in two metrics, Shape Fanning (SF) and Vulnerability Fanning (VF).

For SF, we use the number of incoming and outgoing calls from a code artifact as inputs for the calculation of the metric. These inputs are defined as FanIn and FanOut, respectively, and are measured via the code at immediately precedes and follows an entity in crash dump stack traces.

For SF, we calculate individual values for each permutation of FanIn and FanOut values, as seen in equation 3:

\[ SF(X,Y) = \frac{\text{artifacts with } X \text{ FanIn and } Y \text{ FanOut}}{\text{total number of code artifacts}} \]

For VF, we calculate individual values for each permutation of FanIn and FanOut values, as seen in equation 4:

\[ VF(X,Y) = \frac{\text{vulns with } X \text{ FanIn and } Y \text{ FanOut}}{\text{total number of vulns.}} \]

5.3. Boundary Metric

Boundary metrics determine what system-specific entities appear first after outside entities, and which outside entities are seen directly before code entities in the software system under study. This boundary could be useful for determining which code entities to focus hardening and testing efforts on, while the outside list may indicate which third parties are causing the most issues for end users.

Determining what outside binaries are seen most frequently in crashes, and how frequently those binaries are associated with security fixes would be a useful metric for software engineers. As an example, if specific outside programs are causing the most issues for a system, then software teams can focus maintenance efforts on those areas. Additionally, the organization can reach out to the developers of those programs to help them make better use of the interfaces provided by the software.

Therefore, we measure the amount of vulnerabilities that appear on the boundary of a target system by analyzing crash dump stack traces. We identify the boundary from the stack traces by flagging which binaries are directly related to the target system versus which binaries are external to the system. Typically, the first few
binaries in the stack trace will be external, and the trace will eventually change over to system code. We identify the changeover to system binaries, and mark the first occurrence of system code as the boundary. We then calculate our boundary metric, Boundary Vulnerability Rate (BVR), via the following equation:

\[
BVR = \frac{\text{vulns. on the boundary of the system}}{\text{total number of vulns.}}
\]

From this metric, software developers can make data informed decisions on whether to focus maintenance efforts on the boundary of their system, where their code interacts directly with outside developers, or on other sections of the codebase.

6. Case Study Methodology
In this section, we present data collection methodology specific to the Windows study performed for this paper.

6.1. Stack Trace Collection and Parsing (RQ1)
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. an external 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 approximates 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.

If data about a level of granularity is missing from the stack trace, this data may be able to be extrapolated from the data that is present. For example, if a file and line number is provided in the stack trace, the binary can be determined by the name of the file and the function can be determined from the line number in the file. 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.

6.2. Data Sources
To improve on previous work by researchers [3], larger datasets were required for sufficient samples to work from. The dataset from the summer 2015 work consists of approximately 24.5 million stack traces from 2014 and 2015 from Windows 8, 8.1, and 10, illustrated by Table 1. These crashes represent approximately 500 million records in our database, with a single line in a crash dump represented by a single record. On average, each crash dump has 20 lines or records.
Table 1. Number of crash dump stack traces parsed in 2014 and 2015, by OS version.

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

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

6.3. Change Metric Generation (RQ2)

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

6.4. Complexity Metric Generation (RQ3)

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

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

For this study, we generate a pair of heatmaps displaying all of the permutations of incoming and outgoing calls, and the distribution of each of these permutations throughout the codebase. This type of visualization could be useful for identifying trends in code structure, and if certain structures are more or less likely to have security vulnerabilities.

6.5. Boundary Metric Generation (RQ4)

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

7. Results

In this section, we present the result of our investigation of each research question.

7.1. RQ1: Crash Dump Stack Trace Metric

Table 3 contains a summary of the results from running RASA on Windows 8, Windows 8.1, and Windows 10 on a set of crash dump stack traces from 2014 and 2015. We excluded Windows 10 in 2014 due to the small number of crashes available, as Windows 10 was still in development. The table contains the CC and VC metrics, which are the percentage of code in the target software system that appears in at least one stack trace and the percentage of files with security vulnerabilities that appear in at least one stack trace, respectively.

7.2. RQ2: Change Metric

Table 4 contains the FO, SO, and IB metrics for our three comparisons. For the Windows 8 to 8.1 and Windows 8.1 to 10 comparisons, we see a significant difference in the files that are covered for each version, with 39% of the code covered changing for 8 to 8.1 and 58.7% of the code covered changing for 8.1 to 10. While the VC metric for each version of Windows is
relatively similar, the significant change in code covered indicates that the attack surface of each version of Windows has changed drastically. Windows 10 – 2014 to 2015 represents a software system that is under development. As expected, we see a significant increase in attack surface size, as measured by RASA, from 2014 to 2015. The growth from 2014 to 2015 is expected, as features for Windows 10 were under active development.

7.3. RQ3: Complexity Metric

**RQ3:** How are security vulnerabilities correlated with fan-in and fan-out measures of code artifacts?

Figures 3 through 8 are heatmaps describing the distribution of the SF and VF metrics for Windows 8, 8.1, and 10. With these two heatmaps, we are looking for the differences in the distribution of files throughout the system compared to the distribution of vulnerabilities throughout the system, based on incoming and outgoing calls. This would be realized on the heatmap by differences in the distribution of the percentages on the chart.

Across all three versions of Windows, we very similar profiles of complexity types. All three versions have a cluster of files and vulnerabilities around “simple” files, represented by the darker colors in the top left corner of each of the heatmaps. We also see a “tail” effect in the bottom of the heatmap, with a cluster of complex files and associated vulnerabilities. Finally, each of the three heatmaps has a “valley” between simple and complex files. This indicates that files in Windows tend to be either highly complex or simple, with very little files exhibiting “moderate” complexity.

### Table 3. Percentage of files appearing on crash dump stack traces and vulnerabilities appearing on crash dump stack traces for operating system/year pairings.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Year</th>
<th>Code Coverage (CC)</th>
<th>Vulnerability Coverage (VC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 8</td>
<td>2014</td>
<td>11.0%</td>
<td>17.6%</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>11.9%</td>
<td>17.8%</td>
</tr>
<tr>
<td>Windows 8.1</td>
<td>2014</td>
<td>8.1%</td>
<td>16.3%</td>
</tr>
<tr>
<td></td>
<td>2015</td>
<td>10.0%</td>
<td>18.1%</td>
</tr>
<tr>
<td>Windows 10</td>
<td>2015</td>
<td>7.12%</td>
<td>12.1%</td>
</tr>
</tbody>
</table>

7.4. RQ4: Boundary Metric

**RQ4:** How often are vulnerabilities seen on the entry and exit points of a software system compared to the rest of the codebase?

For Windows 8.1 in 2015, we found that the BVD metric indicates that only 4% of the vulnerabilities found and fixed were on the entry and exit points of the system. This result suggests that focusing security hardening and testing efforts on the entry and exit points of Windows would result in most vulnerabilities going unfound.

Based on this result, we confirm the importance of “building security in,” as recommended by McGraw [6]. Treating security as a wrapper around a product to be handled by entry and exit points would seem to be a large mistake if applied to Windows, as most bugs are not ultimately fixed there. Considering security at every point of the development process and not something to be added later is more important than ever based on our result.


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

8. Conclusion

For RQ1 (How effectively can stack traces to be used to approximate the attack surface of a system?), we replicate the RASA approach for Windows 8, 8.1, and 10 at the file level, improving the granularity, and therefore applicability, of the approach. Further improvements in mapping specific files on crash dump stack traces to files in the system are needed to improve the VC metric.

For RQ2 (How does the code that is seen on crash dump stack traces drop off, get added, and stay the same across versions and during development of soft
Figure 3. A heatmap of the distribution of files in Windows 8, based on the number of incoming and outgoing calls from a file. Heat numbers are expressed as the percentage of files in Windows 8 with that distribution.

Figure 4. A heatmap of the distribution of vulnerabilities in Windows 8, based on the number of incoming and outgoing calls from a file. Heat numbers are expressed as the percentage of vulnerabilities in Windows 8 with that distribution.

Figure 5. A heatmap of the distribution of files in Windows 8.1, based on the number of incoming and outgoing calls from a file. Heat numbers are expressed as the percentage of files in Windows 8.1 with that distribution.

Figure 6. A heatmap of the distribution of vulnerabilities in Windows 8.1, based on the number of incoming and outgoing calls from a file. Heat numbers are expressed as the percentage of vulnerabilities in Windows 8.1 with that distribution.
ware systems?), we see that the attack surface of Windows significantly changes from version to version and year to year. Therefore, security teams should be constantly updating the code they consider riskiest in the system based on the changes made by developers during feature development and bug fixing iterations.

For RQ3 (How are security vulnerabilities correlated with fan-in and fan-out measures of code artifacts?), we did not observe a significant positive or negative correlation between complexity of files and vulnerabilities. However, we have made several observations based on our dataset. First, files in Windows tend to be either highly complex or simple, with very few files being moderately complex. Files in Windows could be assigned as either “complex” or “simple,” which would change the approach security professionals take in securing them.

Given no other metrics for severity, vulnerabilities that appear in “complex” files are more likely to have wide-reaching impact than vulnerabilities that appear in “simple” files. The complexity of files could be combined with the frequency with which a file crashes to form a severity metric independent of a qualitative analysis, which would allow the initial assignment of severity for a vulnerability to be automatic. While a human should assign the final severity to each vulnerability after inspection, an automatic severity measure based on frequency and severity could help them classify the vulnerability appropriately.

For RQ4 (How often are vulnerabilities seen on the entry and exit points of a software system compared to the rest of the codebase?), we see that vulnerabilities were rarely on the edge of the system as compared to the distribution of code throughout Windows 8.1. Focusing security efforts only on entry and exit points would be a mistake, based on this result. Security must be looked at holistically throughout the entire system as vulnerabilities do not seem to concentrate on the edges of the system.

Based on our results, we see that the metrics derived from crash dump stack traces by RASA have positive correlations with security vulnerabilities found in codebases. These metrics could be used by practitioners to guide their efforts in finding and fixing potential security vulnerabilities, and in refactoring and maintaining specific parts of their codebases that are potentially susceptible to vulnerability introduction. We see potential in these new stack trace crash dump metrics for new ways to observe and report on the stability and security properties of software systems.

9. Limitations
These metrics have only been evaluated for Windows 8, 8.1 and 10. This approach may not be generalized to other systems without similar studies in different do-
mains. 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.

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.

10. Future Work

Further exploration is needed of the use of stack traces from crash dumps as a potential source of new software security and reliability metrics. Stack traces from crash dumps are already collected by many companies, and reusing existing datasets that industry teams already maintain is one potential avenue for providing immediate value to practitioners.

One of the issues that should be is the issue of scale with the use of stack traces from crash dumps. Previous studies have focused on large amounts of data, with stack trace counts reaching into the millions for large software systems. How applicable are crash dump stack trace metrics to smaller software systems without that sense of scale? How many stack traces are required before these metrics are accurate and actionable for practitioners? Are millions of stack traces required, or can practitioners use ten thousand or one thousand stack traces to provide meaningful metrics to their teams?

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

Finally, building practical tools that plug into IDEs would be one way to get information from RASA in front of developers and security professionals. However, surveys and interviews to clarify the work environment of security professionals are needed to determine the best way to integrate new information into security workflows.

11. Acknowledgements

The researchers would like to thank team members at Microsoft and Microsoft Research for their invaluable work and feedback on the results found in this paper. We also thank all of the pre-submission readers and reviewers of this paper for their invaluable feedback.

12. References


