Improving Real-World Vulnerability Characterization with Vulnerable Slices

Solmaz Salimi
s.salimi@sharif.edu
Sharif University of Technology
Tehran, Iran

Maryam Ebrahimzadeh
maryam.ebr@student.sharif.edu
Sharif University of Technology
Tehran, Iran

Mehdi Kharrazi
kharrazi@sharif.edu
Sharif University of Technology
Tehran, Iran

ABSTRACT

Vulnerability detection is an important challenge in the security community. Many different techniques have been proposed, ranging from symbolic execution to fuzzing in order to help in identifying vulnerabilities. Even though there has been considerable improvement in these approaches, they perform poorly on a large scale code basis. There has also been an alternate approach, where software metrics are calculated on the overall code structure with the hope of predicting code segments more likely to be vulnerable. The logic has been that more complex code with respect to the software metrics, will be more likely to contain vulnerabilities.

In this paper, we conduct an empirical study with a large dataset of vulnerable codes to discuss if we can change the way we measure metrics to improve vulnerability characterization. More specifically, we introduce vulnerable slices as vulnerable code units to measure the software metrics and then use these new measured metrics to characterize vulnerable codes. The result shows that vulnerable slices significantly increase the accuracy of vulnerability characterization. Further, we utilize vulnerable slices to analyze the dataset of known vulnerabilities, particularly to observe how by using vulnerable slices the size and complexity changes in real-world vulnerabilities.

CCS CONCEPTS

• **Security and privacy** → *Software and application security*; *Vulnerability scanners.*

KEYWORDS

Static Analysis, Program Slicing, Vulnerability Characterization, Vulnerability Prediction.

ACM Reference Format:
Solmaz Salimi, Maryam Ebrahimzadeh, and Mehdi Kharrazi. 2020. Improving Real-World Vulnerability Characterization with Vulnerable Slices. In *Proceedings of the 16th ACM International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '20), November 8–9, 2020, Virtual, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3416508.3417120

1 INTRODUCTION

Vulnerability detection has been an important challenge ever since programs have become an important part of our daily lives. Although much more attention has been given to the challenge over the past decade, where the challenge has been exasperated given the huge volume of code, which is only expanding, and the complexity of the vulnerabilities. There has been a multitude of techniques proposed to search for vulnerabilities within code, these range from black-box random fuzzing [37] to white-box symbolic execution [11, 44] and path-sensitive software analysis [14].

There are two fundamental issues with all vulnerability detection approaches: (1) proposed techniques are not scalable enough and therefore incapable of handling large code basis, regardless as to whether these methods are designed to find vulnerability statistically or dynamically, and (2) proposed techniques suffer from false positives as they are unable to understand or model vulnerable behavior, which is impossible for all the static approaches [28] and a good portion of dynamic approaches are incapable of producing or reproducing vulnerable behavior [45], e.g., black-box fuzzing can only generate crashes rather than the vulnerable behavior.

Nevertheless and even with the above noted limitations, there has been a surge in the number of vulnerabilities detected in real-world software applications over the past two decades. This large trove of data has resulted in a new research direction where known-vulnerability characteristics are leveraged to either prioritize sections of the program which are more prone to vulnerabilities, e.g., vulnerability detection based on similarity to previous known codes [19, 23, 26, 29–31, 40] or alternatively exclude large parts of the code which is unlikely to contain vulnerabilities, therefore minimizing the effect of both the scalability and false positive issues.

In fact the same approach has been used traditionally to expedite the process of bug detection in software [13, 20, 22, 24, 33, 42, 49]. The logic has been constituted by observing that most previously detected bugs exist in complex parts of programs, therefore complex code is more likely to be buggy [34]. Similarly and following on the same logic, many studies have been looking for the correlation between different code metrics, and the probability of finding software vulnerabilities. Seeking to find the most correlated software metrics with vulnerabilities, has resulted in several studies [7, 8, 15, 18, 35, 38, 39, 43, 50, 51].

In fact, a major problem with applying metrics directly on source code is the fact that even though contiguous program statements are related in the overall functionality provided, they may not all be related to a vulnerability in the code. For example, a code segment may include a call to the infamous *strcpy* function which may result in a buffer overflow, but the statements before and after could be concerned with how to parse the input. Now, any metric applied to the complete code would see many statements, some of which are only related to the vulnerability. In other words, there are too many unrelated statements that act as noise to any metric calculated on the code.

We hypothesize that if one could remove unrelated statements to vulnerability, then the metrics would be more precise in identifying vulnerable code. Therefore, we propose using program slicing [48], to decompose a given code to independent slices and instead of measuring metrics for a part of code with pre-defined boundary, i.e., function or file, we measure the metrics for slices obtained from the code. In other words, we measure metrics for vulnerable slices instead of vulnerable functions or vulnerable files.

Furthermore, there is a second benefit associated with slicing the code. As code is decomposed into a set of slices, each slice would be less complex in terms of the number nodes and edges in its control flow graph as well as the number of variables among other things. This minimized complexity would be beneficial to vulnerability detection tools as they will concentrate on a smaller piece of code with a smaller number of variables.

In order to investigate the above-noted issues, we gather a comprehensive dataset by aggregating a number of resources including, NVD [5] and Red Hat vulnerability databases [6], GitHub [4] metadata collected with GitHub API, and other available open-source projects metadata and their security tracker [2], and utilize this massive metadata to select target projects and extract program slices related to known vulnerabilities, which we call vulnerable slices, and slices that are not related with any known vulnerabilities, non-vulnerable slices. We then measure software metrics for all slices and their corresponding functions to answer the following research questions:

RQ1: Could program slices better characterize vulnerabilities in programs? In other words, can program slicing preserves all source code features that are related to vulnerability and yet makes vulnerable and non-vulnerable slices more distinguishable than their corresponding vulnerable and non-vulnerable functions?

RQ2: How much does slicing help with providing less complex code? This is an important issue that affects the performance and effectiveness of vulnerability detection tools.

More specifically, we make the following contributions:

- We leverage a program slicing based approach to extract (non-consecutive) code units that contain vulnerabilities with utilizing public known CVEs and public open source projects
- We conduct a comprehensive empirical study of extracted vulnerabilities, investigate more than 6000 code commits related to vulnerabilities with CVEs to build a dataset of vulnerable functions and corresponding vulnerable slices, and calculate software metrics over them to measure their correlation with vulnerabilities.
- We show that vulnerable slices outperform vulnerable functions in characterizing vulnerabilities.
- Finally, we statistically measure a complexity and size change rate between vulnerable function and its corresponding vulnerable slices for all known vulnerabilities in our dataset.

In what follows, we first introduce our high-level approach in Section 2.In Section 3 we explain our data collection methodology, dataset construction, and software metrics that we used in our empirical study. In Section 4 we analyze the results to answer RQ1 and RQ2. In Section 5, we present the literature and related work. Finally, in Section 6 we discuss some aspects of our approach and previous approaches and we conclude.

2 APPROACH

Software boundaries are conceptual lines that divide parts of the program that we want to study or relevant parts from other parts of the program that are considered as irrelevant. Large programs may have multiple boundaries and defining boundaries means deciding what part of programs are relevant to the specific program properties, which is an inherently undecidable problem. Due to this fact, programs have pre-defined boundaries indicated by developers. The most important pre-defined boundaries traditionally used in the software are files and functions.

Functions are defined by developers and for specific functionality, they inherently isolate parts of the programs that are relevant to that particular functionality. Furthermore, using a function as a boundary to pinpoint a vulnerability is used in previous studies repeatedly [12, 18, 26] and it is obvious that they have a well-defined input/output that may be enough for a vulnerability to trigger and eliminate lots of irrelevant statements comparing to file-level boundaries, although function-level boundaries still have irrelevant data. Consider the fact that even though statements in a function are executed sequentially and depending on the order in which they are located in the function, two statements next two each other may not necessarily affect each other.

This important observation could affect the way we calculate metrics on the code, especially when considering vulnerable code, as some of the statements in the code have no effect on the vulnerability, but are in the same region as the vulnerable code. For example, consider the code snippet illustrated in Figure 1 obtained from the *GD Graphics Library* [3] directly used in PHP source code, which includes a heap-based buffer overflow vulnerability identified by CVE-2019-6977 [1]. In the same Figure, statements that are related to the noted vulnerability are highlighted. As clearly shown in this motivational example, only a fraction of the statements are required for the vulnerability to be exploited and a large portion of the code in this function would be redundant, where vulnerable slice obtained by backward program slicing the code from the vulnerability triggering statement to the top of the function.

In what follows, at first, we review how slices are obtained from code, and then present our approach in investigating the effect of slicing in improving the accuracy of metrics calculated on the code units as opposed to when the metrics are calculated on functions.

2.1 Program Slicing

Program slicing [48] is one of the most practicable techniques to extract code units of programs that are related to a particular property. This technique was originally introduced to isolate parts of the program that are related to a bug in order to minimize the debug process for developers.

```
1  BGD_DECLARE(int) gdImageColorMatch (gdImagePtr im1, ↩
       gdImagePtr im2)
2  {
3    unsigned long *buf; /* stores our calculations
4    unsigned long *bp;  /* buf ptr */
5    int color, rgb;
6    int x,y;
7    int count;
8
9    if (!im1->trueColor) {
10     return -1; /* im1 must be True Color */
11   }
12   if (im2->trueColor) {
13     return -2; /* im2 must be indexed */
14   }
15   if ((im1->sx != im2->sx) || (im1->sy != im2->sy)) {
16     return -3; /* the images are meant to be the same ↩
       dimensions */
17   }
18   if (im2->colorsTotal < 1) {
19     return -4; /* At least 1 color must be allocated */
20   }
21
22   buf = (unsigned long *)gdMalloc(sizeof(unsigned long) * 5 * gdMaxColors);
23   memset (buf, 0, sizeof(unsigned long) * 5 * gdMaxColors );
24   for (x=0; x < im1->sx; x++) {
25     for( y=0; y<im1->sy; y++ ) {
26       color = im2->pixels[y][x];
27       rgb = im1->tpixels[y][x];
28       bp = buf + (color * 5);
29       (*(bp++))++;
30       *(bp++) += gdTrueColorGetRed(rgb);
31       *(bp++) += gdTrueColorGetGreen(rgb);
32       *(bp++) += gdTrueColorGetBlue(rgb);
33       *(bp++) += gdTrueColorGetAlpha(rgb);
34     }
35   }
36   bp = buf;
37   for (color=0; color < im2->colorsTotal; color++) {
38     count = *(bp++);
39     if( count > 0 ) {
40       im2->red[color]   = *(bp++) / count;
41       im2->green[color] = *(bp++) / count;
42       im2->blue[color]  = *(bp++) / count;
43       im2->alpha[color] = *(bp++) / count;
44     } else {
45       bp += 4;
46     }
47   }
48   gdFree(buf);
49   return 0;
50 }
```

Figure 1: Vulnerable function coupled with the vulnerability identified by CVE-2019-6977, a heap-based buffer overflow from GD Graphics Library. By using vulnerable slice instead of a vulnerable function, we can emit parts of code that are irrelevant to the vulnerability.

Because of the various applications of program slicing, various algorithms exist that compute program slices [46], while one of the most common program slicing approaches, is the original static program slicing approach [48], which extracts slice, S, as an *executable* and *reduced* part of the program, P, where S replicates a particular behavior of P. This algorithm needs a slicing criterion, which indicates a program point, and a set of program variables, and by starting from the program point and moving to the top of the program (i.e. backward slicing), it removes irrelevant part of the program with respect to variable set until it reaches the beginning of the program.

By using the original program slicing technique, to generate a slice for any given function and statement of interest within this function, at the first step, we extract function source code from the project and build its control flow graph (CFG), then we transform its CFG, such that every basic block corresponds to one program statement. Then we generate program dependence graph (PDG) from this CFG. Using so-called PDG is one of the established program slicing methods. PDG is originally introduced in [27] as a program representation, but due to an explicit representation of dependencies between statements in the program that PDG provides, utilizing it for program slicing is quite suitable. PDG is a directed graph that has nodes from a CFG and two kinds of edges: control dependence edges and data dependence edges. Given a PDG of the program and a node of interest in this graph, we need to recursively find all the nodes that the node of interest have data or control dependence on them. As PDG nodes are equal to CFG nodes (where each node corresponds to a single program statement), if the slicing criterion is equal to the vulnerability point, the resulting slice is vulnerable and if the criterion is selected independently of the vulnerability, the resulting slice is non-vulnerable.

In this work, we gather slices for all nodes of functions and mark them as non-vulnerable slices. When it comes to vulnerable function, we pinpoint a slicing criterion, i.e., any statement in the program and a set of all *used* variables within this statement which result in the vulnerability, and by leveraging backward program slicing, we obtain a vulnerable slice, which includes all the related statements of the function that are necessary to reach that program point. This would be separate from non-vulnerable slices obtained from the vulnerable function. In our motivation example in Figure 1, line *23* is the vulnerability point, i.e., where vulnerability is triggered. If we use this statement as a slice criterion and use backward slicing to obtain its related program slice, highlighted statements correspond to the slice. It should be noted that this function will also results in 38 non-vulnerable slices, given there are 38 statements that can be used as program points for slicing criteria in this program.

2.2 Software Metrics

In this study, we reuse most of the software metrics introduced in [18], due to the fact that they are already proved to be correlated with vulnerabilities. we categorize software metrics based on the way we measure them, as if they are computed by utilizing the structures, we call them structure-level, or they are obtained over code, we call statement-level metrics. For example, cyclomatic metric is computed over the program CFG, regardless of code text, while some other metrics like the number of variable declared in the program are computed by checking source code of a target program. Table 1 summarizes metrics and their categories we use in our empirical study.

To measure the metrics, we utilize the CFG of the program and transformed it where each of its block represent only one program statement instead of a basic blocks. After this CFG transformation, we measure each category of metrics as follows:

Table 1: Software metrics used in our empirical study. Structure-Based metrics are obtained by analyzing a program structure, while Statement-Based metrics are computed directly by using the program statement.

Category	ID	Metric
Structure-Based	M1	Cyclomatic complexity
	M2	# loops
	M3	# nested loops
	M4	Maximum nesting level of loops
	M5	# nested control structures
	M6	Maximum nesting level of control structures
	M7	# if structures without else
	M8	CFG Size
Statement-Based	M9	# variables as parameters for callee function
	M10	# pointer arithmetic
	M11	# variables involved in pointer arithmetic
	M12	Max pointer arithmetic a variable is involved in
	M13	# variables involved in control predicates
	M14	# variables defined

- **Structure-Based:** These metrics can be computed directly from the structure of CFG. We compute the cyclomatic complexity metric (M1) by counting the number of nodes (N) and edges (E) of graph and M1 is calculated by using $E - N + 2$ formula. Metrics (M2-M4) are computed by using CFG loops and metrics (M5-M7) are computed by counting nodes that have more than one successors. Finally, CFG size metric (M8) is equal to the N.
- **Statement-Based:** These metrics can be computed by processing the text of each CFG node statement. During CFG generation, we store text and each node's special metadata for each statement, where the most important data is statement's kind, i.e., a call, declaration, binary operation, etc. By processing code unit's AST, we aggregate all used variables for each statement and compute M9. Then we use these mentioned variable lists and text of each statement to extract pointers and other metrics related to pointers (M10-M12). For M13 we use CFG to find control structure nodes, then we find variables involved in these nodes. We extract all variables which are involved in each corresponding CFG node and aggregate them in M14.

2.3 Evaluation Methodology

Figure 2 illustrates the flow of our empirical study, where we gather a large dataset of vulnerabilities, leveraging the available corpus of real-world vulnerabilities in public. We collect open-source programs in the *C* language from GitHub, and collect metadata about known vulnerabilities to distinguish vulnerable and non-vulnerable functions. We discuss the details on the the dataset is collected in Section 3.

The dataset is employed to generate a large number of vulnerable and non-vulnerable slices by utilizing the PDG-based backward slicing approach we described previously. Afterwards, the software metrics noted in Section 2.2 are applied to both the vulnerable and non-vulnerable slices, as well as vulnerable and non-vulnerable functions. Lastly, we leverage machine learning techniques with the obtained dataset to classify vulnerable and non-vulnerable slices, as well as vulnerable and non-vulnerable functions separately. More details on the evaluation can be found in Section 4.2. By comparing the obtained accuracy with slices vs. function as well as how the value of the metrics changed between slices and function we could evaluate **RQ1** and **RQ2**, as noted in Section 1 with confidence.

3 DATA COLLECTION METHODOLOGY

In this section, we introduce our method to explore real-world applications to construct a dataset of the vulnerable and non-vulnerable samples of functions and slices. As our study is based on software metrics, we need to have access to program source codes on large scale, therefore we leverage the large number of projects hosted on GitHub to collect our dataset. Our main source of information for vulnerabilities is obtained from the National Vulnerability Database (NVD) [5], a database of publicly disclosed vulnerabilities where each known vulnerability is identified with a unique id known as CVE. We complement this database by Red Hat CVE Database [6]. In the following, we describe the process of collecting repositories from GitHub, searching and extracting vulnerabilities with CVE id.

3.1 Vulnerability Metadata Collection

We crawled the public vulnerability database from NVD and Red Hat. We leveraged NVD API and crawled its database on March 2020, and obtained 141710 CVE identifiers. Further, we crawled Red Hat CVE database on the same date, which resulted in 21620 identifiers. Finally, our integrated database of CVE identifiers comprises 163330 unique CVE ids. This metadata is required, because even though the CVE id structure is known (i.e. CVE-year-0∼1000000), in some cases the vulnerability is assigned but never confirmed, so we need to inquire public vulnerability metadata to assure the CVE identifier is related to a confirmed vulnerability.

3.2 Selecting Project Repositories

GitHub provides a rich API to explore projects with various metadata, including the main language of projects, number of commits, number of stars and forks (which show the popularity of the project), and number of commits and collaborators (which show how active the projects is). We developed a crawler by utilizing this API to gather enough metadata from GitHub, more specifically we crawled Github and obtained 3409172 entries for *C* and *C++* language projects, then we selected the top projects by the number of stars and number of forks and download 23,650 projects. We then extract all commits for each project to count the number of its commits with a reference to one or more confirmed CVE identifiers in their *commit message*. Repositories are then ranked with respect to the number of CVE-related commits and the top 10 projects are

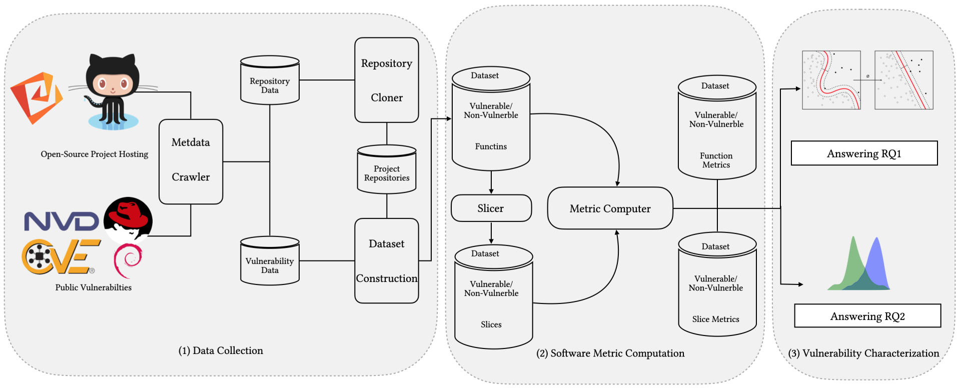


Figure 2: Our empirical study flow. In this flow, at first we collect data by using the public vulnerability database and public open-source projects and construct our dataset. In the second step, we process the data we collect and compute software metrics. Finally, we use the computed metrics as a list of features to analyze vulnerabilities and their characteristics over these software metrics. Particularly, we classify both vulnerable/non-vulnerable functions and slices and compare their models to answer RQ1. Finally, we estimate complexity rate change to answer RQ2.

Table 2: List of selected projects for our empirical study and their metadata

Project	#Commits	#CVE	#Stars	#Forks	SLoC (C)
samba	122150	1924	310	255	1427309
mptcp	799300	758	518	224	11701125
xen	41080	757	140	121	410014
postgres	49240	740	140	121	801782
libvirt	37082	534	422	311	458152
openssl	26101	500	8871	4050	384653
linux	916276	378	66686	24180	14368375
FFmpeg	97817	349	13058	5070	987113
libav	45202	332	658	299	518083
google kmsan	916433	385	175	31	14368466
Total	3050681	6657	-	-	45425072

selected. The GitHub metadata and project source files were downloaded in March 2020. Table 2 summarizes the top 10 projects we selected for our empirical study, the data for these selected project was update in April 2020.

3.3 Identifying CVE Related Functions/Slices

In order to identify the program version which includes the vulnerability, we look for commit messages which point to a vulnerability patch, including a confirmed CVE identifier. Hence we avoid employing buggy code instead of vulnerable code. Figure 3 illustrates the process of vulnerability code sampling. Given a project, we first extract all commit metadata, then search them to find any of confirmed CVE identifiers. At the next step, for each commit message with a CVE id, which we know as code that patches that particular vulnerability, we find its parent commit, and change the version of project to the parent commit, this version contains the vulnerability. We also store the commit *diff* file for patch commit which further we utilize to identify the root cause of vulnerability in source code.

For each stored vulnerable and non-vulnerable function, we employ the slicing process as illustrated in Section 2.1. If the function is vulnerable, we extract a vulnerable slice from that by processing the *diff* file and store the rest of slices as non-vulnerable. For non-vulnerable functions, we extract and store all of its slices as non-vulnerable slices. In the end, we have a dataset composed of a collection of functions and slices, where each item is either tagged as vulnerable or non-vulnerable. Table 3 represents our final dataset sample counts. It is worth mentioning that the dataset contains more non-vulnerable samples, but during analysis, we used a balanced portion of entries. Furthermore, as the table indicates, the number of vulnerable slices is less than the number of vulnerable functions; the main reason is that in some cases, mostly when the patch only changes the function *prototype*, corresponding extracted slices are empty and marked as invalid.

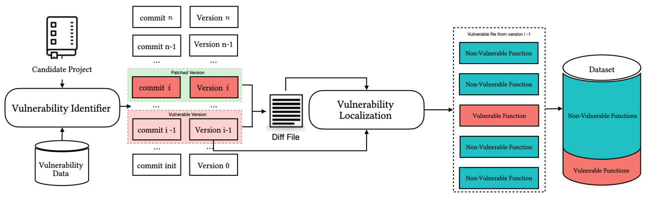


Figure 3: Code sampling process. To construct our dataset, for any given (git) project, we extract all its commits and assume each commit corresponds to a specific version. We explore the commit message to find any sample of CVE id from known vulnerability database. Any version, i, updated with commit message that contains a CVE identifier, is considered as the patch for the vulnerability with mentioned CVE id, therefore version $i-1$ contains the vulnerability. We rollback the project to version $i-1$, localize the vulnerability and find the file and the function that are changed in version i and are related to the vulnerability. Then we store the vulnerable function and all other functions from the file as non-vulnerable in our dataset.

Table 3: Dataset information.

Tag	**#Samples**
Non-vulnerable Function	54252
Non-Vulnerable Slice	108090
Vulnerable Function	3715
Vulnerable Slice	1403
Total	167460

4 ANALYSIS

In this section, we evaluate the two research questions put forth in Section 1. In Section 4.2 we present our approach to build a model with support-vector machines for both slice and function datasets. By comparing these two models we can decide if vulnerable slices are more suitable than vulnerable functions in characterizing vulnerabilities based on the software metrics discussed earlier and answer **RQ1**. Further, in Section 4.3 we leverage our dataset to measure the complexity change rate between a vulnerable function and its corresponding vulnerable slice to answer **RQ2**. But first, we provide some implementation details at 4.1.

4.1 Implementation Detail

We developed a metadata crawler that efficiently utilize GitHub and Red Hat APIs to gather project’s metadata, which we developed in Python 3.7. For function CFG generation, we leveraged Clang.8.0 [16] and its C++ API, and developed an accurate CFG extractor. Our CFG nodes contains all metadata that Clangs’s provided for a given program statement, particularly we store each program’s kind, e.g., deceleration, loop, etc. Finally, that we developed an efficient program slicer which accepts a CFG, change its basic blocks to blocks with single statement and generates data and control dependence graphs, and finally combines these two graphs to build PDG. CFG and PDG have common nodes.

For slice generation phase, our slicer accepts a program’s statement, finds its corresponding PDG node as slicing criterion, explores the PDG to find nodes that criterion node has dependence on them. We should mention that both slicer and CFG generator are implemented to work on stand-alone files and functions, while tools like Frama-C [17] cannot be applied in stand-alone manner.

4.2 Vulnerability Classification

We have introduced the set of software metrics we want to use to classify our dataset in Section 2.2. This set of metrics have been previously shown [18] to be correlated with vulnerabilities. As we pointed in our motivation and specified in **RQ1**, we want to investigate our data to see if this model improves when it is built with vulnerable slices instead of vulnerable functions. Table 4 presents the average and variance for each metric, measured for four categories of data available in our dataset.

Accordingly, we aim to build two models, one with vulnerable and non-vulnerable function samples and one with vulnerable and non-vulnerable slice samples and compare these two models to see which one is preferable. Given the size of our dataset and various previously proved correlated metrics we reuse as learning features, we leverage support-vector machines [10] (SVM) for classification. SVM inherently is proved to be promising if the set of supervised data has enough features, meaning that it is very effective in high dimensional spaces and by tuning its various configurable parameters it is possible to improve the model efficiently.

We setup SVM with the radial kernel. To tune kernel parameters we utilize grid search and finally, we leverage 10-fold cross-validation, i.e., using the dataset for both training and testing, to evaluate our model. We train our models with our dataset which we presented in Table 3, and we limit sample size to vulnerable slices to get a balanced sized dataset. Table 5 presents the evaluation of function-based and slice-based models. Further, Figure 4 illustrates ROC score for 10-fold cross validation for both models. As evident from the results, slice based metrics do outperform function based metrics considerably.

4.3 Complexity Change Rate

In **RQ2** we brought up the importance of other aspects of vulnerability characterization that are especially efficient for assessing the process of vulnerability detection. Mainly we want to analyze real-world vulnerabilities, and compare corresponding slice and function for the specific vulnerability, and measure how the size and complexity change. For instance, we know that slices are generally smaller than their corresponding functions, but how size difference is changing for the known and real-world vulnerabilities? If the size is not changing so much, using vulnerable slices is not so beneficial. Also, if the size of slices is not changing as expected, should we consider complexity changes? For instance, if the vulnerable slice size is almost equal to the vulnerable function size, but their complexity difference is high, it is still worth to use slices for vulnerability identification tools.

As its has been previously studied, the average slice contains just under one third of the program [9]. So our expectation is that we should observe the same size change for vulnerable slice and its corresponding function. Further, as we have the assumption that

Table 4: Metrics' average and variance over dataset.

	NVS[*]		**VS**[◇]		**NVF**[+]		**VF**[†]	
Mteric	Avg.	Variance	Avg.	Variance	Avg.	Variance	Avg.	Variance
M1	2.52	3.96	2.46	3.07	4.39	7.97	6.11	22.73
M2	0.80	2.16	0.62	1.41	0.25	0.58	0.40	1.11
M3	0.61	3.87	0.65	4.07	0.10	0.26	0.13	0.33
M4	0.25	0.44	0.20	0.32	0.00	0.00	0.07	0.06
M5	3.51	110.54	2.72	85.11	10.56	555.12	26.07	2800.49
M6	1.14	2.20	1.23	2.24	3.01	5.08	4.27	13.35
M7	1.18	1.69	1.39	2.37	0.03	0.03	0.08	0.14
M8	8.42	27.39	10.29	24.95	13.04	56.26	18.88	184.64
M9	0.49	1.05	0.55	0.98	0.18	0.24	0.27	0.46
M10	0.77	2.22	0.81	1.76	0.92	2.20	1.15	2.84
M11	0.44	0.93	0.43	0.75	0.18	0.24	0.25	0.44
M12	0.11	0.10	0.14	0.12	0.09	0.08	0.09	0.08
M13	0.44	1.12	0.51	0.85	0.72	0.98	0.75	1.00
M14	3.07	5.17	2.88	4.25	5.03	10.79	6.33	18.04

[*] Non-Vulnerable Slice, [◇] Vulnerable Slice , [+]Non-Vulnerable Function , [†] Vulnerable Function

Table 5: Vulnerability models evaluation with 10-fold cross-validation.

Model	**Accuracy**	**Precision**	**Recall**	**F-1**
Function-Based	62%	60%	63%	61%
Slice-Based	81%	75%	76%	75%

size cannot represent the complexity of code properly, as such we used 14 metrics in our empirical study. Therefore, we also want to measure the complexity difference between slice and vulnerable function.

To conduct this statistical analysis, we assume that for any given vulnerable function, we obtain $s_v \in S_f$ as its vulnerable slice where $S_f = \{s_1, s_2, ..., s_n\}$ are the set of slices for f. We measure the complexity change rate and size change rate as follows:

$$\textit{complexity change}(f, s_v) = \frac{\sum_{i=1}^{14} \left(\frac{M_i(f) - M_i(s_v)}{M_i(f)} \right)}{14}$$

$$\textit{CFG size change}(f, s_v) = \frac{M_8(f) - M8_i(s_v)}{M_8(f)}$$

Where each M_i is a metric that we described in Table 1, and $i = 14$ is the number of metrics. M_8 in the size change equation is the particular metric related to *CFG size* where it highly correlated with code size. Figure 5 illustrates the distribution of the difference for known vulnerable slices vs. vulnerable functions.

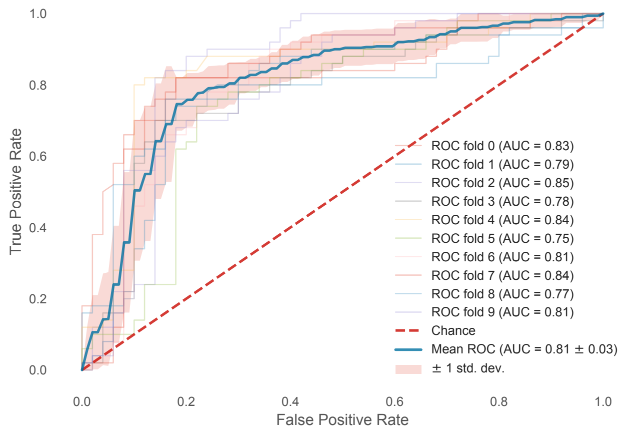
The average rate of complexity change measured for all vulnerability samples in our dataset is 0.43. This means, that overall, slices are simpler that functions. Also, the CFG size is smaller.

5 RELATED WORK

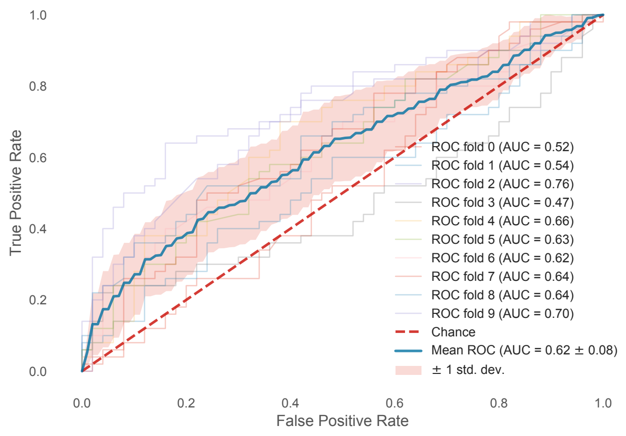
There are two aspects that exist in previous work, but we use them in different manner. At first we discuss these two aspects and then we review the previous work.

- **Bugs versus vulnerabilities:** As we have mentioned in Section 3, for this study we focused on vulnerabilities rather than bugs. There have been previous works that construct their dataset based on bugs, and to do so they do not use CVE identifiers. Therefore, they use some other keywords that is coupled with software bugs, .e.g., they search commit messages for "fix" keyword. This approach will result to gather more data, yet they might contain lots of noisy data. For the given example, developers use the "fix" keyword in their commits to announce a functionality fix, or "bug" might refer to a functionality bug rather than security bugs. This particular intuition encourages us to focus on CVE identifiers instead of other keywords.

- **Slice-based metrics versus software metrics for slices:** Some previous works, including [7] focuses on improving vulnerability characterization by using slice-based metrics



(a) ROC for Slice-Based Model



(b) ROC for Function-Based Model

Figure 4: Receiver Operating Characteristic (ROC) metric to evaluate classifier output quality using 10-fold cross-validation.

for file-level or function-level. The most reputable slice-based metrics are known as cohesion and coupling introduced in [41]. The main goal of adding slice-based metrics is to improve the function-based or file-based model for characterizing vulnerabilities. We differ fundamentally from these works since we introduced vulnerable slices and then train a new model based on vulnerable and non-vulnerable slices rather than adding slice-based metrics as features to function-based model.

One of the main applications of characterizing software bugs and vulnerabilities based on software metrics and with different granularity levels (i.e., project, file, function, etc) is to make software

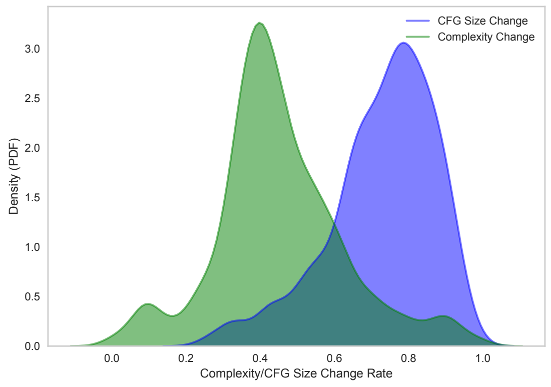


Figure 5: The distribution of known vulnerability based on *complexity change* rate

systems more secure. The characteristics of buggy and vulnerable codes can be utilized in two distinct ways: (1) to examine off-the-shelf software to measure the probability of finding vulnerable and buggy codes and fixing them in software before they become critical, and (2) to measure the trustworthiness [36] and security level of available software systems [21] in order to benchmark them, while the second way is out of the scope of this of our empirical study, we focus on exploring the previous researches on the first category, where the aim is to make characterization more precise to find more vulnerabilities.

In [7] five new slice-based metrics in file-level granularity and five other baseline metrics in function-level are introduced to build a predictive model over a dataset of buggy files of code. The result shows, as expected, that adding these new slice-based features can improve the accuracy of buggy file detection. Although the slice-based metrics are introduced in this work, the granularity and boundary of buggy codes are still constant, they model their dataset on files and functions again. While we have utilized the slice as a unit that characterizes the vulnerability, in [7], the unit that characterizes the vulnerability is still function.

In [50] a method is proposed to train vulnerability prediction models with several project-level metrics including popularity metrics and developer metrics. In [15] a similar supervised model is trained by leveraging complexity, coupling, and cohesion (CCC) file-level metrics, and then the model is utilized to predict vulnerability existence in software systems. The same coupling metrics with file-level granularity are used in [39].

In [43], a text mining approach is proposed to learn a model by using token frequency metrics for each file, to predict vulnerabilities, while later in [47], text mining approaches and metrics are both examined to assess their effectiveness in vulnerability prediction. Finally, in [51] both metrics and text mining approaches are combined to train a more comprehensive model to predict vulnerable codes. In [12] 36 supervised and 12 unsupervised methods are assessed in file-level granularity using effort-aware performance measures.

There are a number of works [13, 20, 24, 33, 49] focusing on defect prediction, they focus on comparing unsupervised and supervised models in just-in-time [25] granularity level that considers changes in metrics in each code churn. Tracy Hall et al. [22] review 36 fault prediction studies and present a boxplot diagram for each level of granularity: class, file, module, combination, or other (e.g., plug-ins, binaries), and their diagram does not suggest a clear correlation between granularity and performance.

In [18] 4 software complexity metrics and 11 vulnerability-related metrics are calculated with function-level granularity and a score is measured for each function. Then by ranking metrics more potential vulnerable functions are discovered. In [32] software metrics for the complexity of both files and functions is calculated in order to find how vulnerabilities are distributed. In [8], a set of 27 software metrics is used in function-level granularity to compute the correlation between them and the number of vulnerabilities using some coefficient like Spearman's Rank. In [35] different project-level, file-level, and function-level metrics are defined and a genetic algorithm is applied to find the best set of metrics, which were more correlated with number of vulnerabilities.

6 CONCLUSION

In conclusion, in this work we introduced vulnerable slices as base code units to characterize vulnerabilities. We brought up two research questions which by answering them we demonstrated how useful are vulnerable slices. To answer if the vulnerable slices are effective to improve the vulnerability characterization model, we leveraged SVM classification to train two models for function-based and slice-based samples. Then we evaluate them, and showed slice-based model can improve the model accuracy at least 10%.

The other research question was brought up to investigate the usefulness of vulnerable slices to assess the process of vulnerability detection. To answer this question, we leveraged our dataset to investigate the amount of difference between vulnerable slice and its corresponding vulnerable function with respect to both complexity metrics and size. Which we measured that the average vulnerable slices contain just under one third of the vulnerable function, meaning it changes almost 72% in size and we showed that with respect to complexity metrics vulnerable slices are 45% less complex to vulnerable functions.

ACKNOWLEDGMENTS

This work was partially supported by the SUT Research Grant G970816.

REFERENCES

[1] 2020. *CVE-2019-6977.* Retrieved April, 2020 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-6977

[2] 2020. *Debian Security Tracker.* Retrieved March, 2020 from https://salsa.debian.org/security-tracker-team/security-tracker

[3] 2020. *GD Graphics Library.* Retrieved April, 2020 from https://libgd.github.io

[4] 2020. *GitHub.* Retrieved March, 2020 from https://github.com

[5] 2020. *NVD Database.* Retrieved March, 2020 from https://www.cvedetails.com/browse-by-date.php

[6] 2020. *Red Hat CVE Database.* Retrieved March, 2020 from https://access.redhat.com/security/security-updates/#/cve

[7] Basma S. Alqadi and Jonathan I. Maletic. 2020. Slice-Based Cognitive Complexity Metrics for Defect Prediction. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). IEEE, 411–422.

[8] Henrique Alves, Baldoino Fonseca, and Nuno Antunes. 2016. Software Metrics and Security Vulnerabilities: Dataset and Exploratory Study. In *12th European Dependable Computing Conference, EDCC 2016, Gothenburg, Sweden, September 5-9, 2016*. IEEE Computer Society, 37–44.

[9] David W. Binkley, Nicolas Gold, and Mark Harman. 2007. An empirical study of static program slice size. *ACM Trans. Softw. Eng. Methodol.* 16, 2 (2007), 8.

[10] Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. 1992. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory, COLT 1992, Pittsburgh, PA, USA, July 27-29, 1992*, David Haussler (Ed.). ACM, 144–152.

[11] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224.

[12] Xiang Chen, Yingquan Zhao, Zhanqi Cui, Guozhu Meng, Yang Liu, and Zan Wang. 2020. Large-Scale Empirical Studies on Effort-Aware Security Vulnerability Prediction Methods. *IEEE Trans. Reliability* 69, 1 (2020), 70–87.

[13] Xiang Chen, Yingquan Zhao, Qiuping Wang, and Zhidan Yuan. 2018. MULTI: Multi-objective effort-aware just-in-time software defect prediction. *Inf. Softw. Technol.* 93 (2018), 1–13. https://doi.org/10.1016/j.infsof.2017.08.004

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, Rajiv Gupta and Todd C. Mowry (Eds.). ACM, 265–278.

[15] Istehad Chowdhury and Mohammad Zulkernine. 2011. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *J. Syst. Archit.* 57, 3 (2011), 294–313.

[16] Clang. 2020. *Clang.* Retrieved March, 2020 from https://clang.llvm.org

[17] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-C - A Software Analysis Perspective. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7504)*, George Eleftherakis, Mike Hinchey, and Mike Holcombe (Eds.). Springer, 233–247.

[18] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 60–71.

[19] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. 2016. Scalable Graph-based Bug Search for Firmware Images. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. 480–491.

[20] Wei Fu and Tim Menzies. 2017. Revisiting Unsupervised Learning for Defect Prediction. *CoRR* abs/1703.00132 (2017).

[21] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 18–21.

[22] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. [n.d.]. A Systematic Literature Review on Fault Prediction Performance in Software Engineering. *IEEE Trans. Software Eng.* 38, 6 ([n. d.]), 1276–1304.

[23] Yikun Hu, Yuanyuan Zhang, Juanru Li, and Dawu Gu. 2017. Binary code clone detection across architectures and compiling configurations. In *Proceedings of the 25th International Conference on Program Comprehension, ICPC 2017, Buenos Aires, Argentina, May 22-23, 2017*. 88–98.

[24] Qiao Huang, Xin Xia, and David Lo. 2017. Supervised vs unsupervised models: A holistic look at effort-aware just-in-time defect prediction. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, 159–170.

[25] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A Large-Scale Empirical Study of Just-in-Time Quality Assurance. *IEEE Trans. Software Eng.* 39, 6 (2013), 757–773.

[26] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*. 595–614.

[27] David J. Kuck, Robert H. Kuhn, David A. Padua, Bruce Leasure, and Michael Wolfe. 1981. Dependence Graphs and Compiler Optimizations. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages, Williamsburg, Virginia, USA, January 1981*, John White, Richard J. Lipton, and Patricia C. Goldberg (Eds.). ACM Press, 207–218.

[28] William Landi. 1992. Undecidability of Static Analysis. *LOPLAS* 1, 4 (1992), 323–337.

[29] Hongzhe Li, Hyuckmin Kwon, Jonghoon Kwon, and Heejo Lee. 2016. CLORIFI: software vulnerability discovery using code clone verification. *Concurrency and*

Computation: Practice and Experience 28, 6 (2016), 1900–1917.

[30] Hongzhe Li, Jaesang Oh, Hakjoo Oh, and Heejo Lee. 2016. Automated Source Code Instrumentation for Verifying Potential Vulnerabilities. In *ICT Systems Security and Privacy Protection - 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30 - June 1, 2016, Proceedings*. 211–226.

[31] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*. 201–213.

[32] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A Large-Scale Empirical Study on Vulnerability Distribution within Projects and the Lessons Learned. In *2020 IEEE/ACM 42 st International Conference on Software Engineering (ICSE)*.

[33] Jinping Liu, Yuming Zhou, Yibiao Yang, Hongmin Lu, and Baowen Xu. 2017. Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.). IEEE Computer Society, 11–19.

[34] Gary McGraw. 2006. Software Security: Building Security In. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*. IEEE Computer Society, 6.

[35] Nadia Patricia Da Silva Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. 2017. Software Metrics as Indicators of Security Vulnerabilities. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*. IEEE Computer Society, 216–227.

[36] Nadia Patricia Da Silva Medeiros, Naghmeh Ivaki, Pedro Costa, and Marco Vieira. 2018. An Approach for Trustworthiness Benchmarking Using Software Metrics. In *23rd IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2018, Taipei, Taiwan, December 4-7, 2018*. IEEE, 84–93.

[37] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* 33, 12 (1990), 32–44.

[38] Patrick Morrison, Rahul Pandita, Xusheng Xiao, Ram Chillarege, and Laurie Williams. 2018. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering* 23, 3 (2018), 1383–1421.

[39] Sara Moshtari and Ashkan Sami. 2016. Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction. (2016), 1415–1421.

[40] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The Attack of the Clones: A Study of the Impact of Shared Code on Vulnerability Patching. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*. 692–708.

[41] Linda M. Ott and Jeffrey J. Thuss. 1993. Slice based metrics for estimating cohesion. In *Proceedings of the First International Software Metrics Symposium, METRICS 1993, May 21-22, 1993, Balimore, Maryland, USA*. IEEE Computer Society, 71–81.

[42] Danijel Radjenovic, Marjan Hericko, Richard Torkar, and Ales Zivkovic. 2013. Software fault prediction metrics: A systematic literature review. *Inf. Softw. Technol.* 55, 8 (2013), 1397–1418.

[43] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. *IEEE Trans. Software Eng.* 40, 10 (2014), 993–1006.

[44] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 317–331.

[45] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Krügel, and Giovanni Vigna. 2016. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. 138–157.

[46] Frank Tip. 1995. A survey of program slicing techniques. *J. Prog. Lang.* 3, 3 (1995). http://compscinet.dcs.kcl.ac.uk/JP/jp030301.abs.html

[47] James Walden, Jeff Stuckman, and Riccardo Scandariato. 2014. Predicting Vulnerable Components: Software Metrics vs Text Mining. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*. IEEE Computer Society, 23–33.

[48] Mark Weiser. 1984. Program slicing. *IEEE Transactions on software engineering* 4 (1984), 352–357.

[49] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. 2016. Effort-aware just-in-time defect prediction: simple unsupervised models could be better than supervised models. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 157–168.

[50] Mengyuan Zhang, Xavier de Carné de Carnavalet, Lingyu Wang, and Ahmed Ragab. 2019. Large-Scale Empirical Study of Important Features Indicative of Discovered Vulnerabilities to Assess Application Security. *IEEE Trans. Information Forensics and Security* 14, 9 (2019), 2315–2330.

[51] Yun Zhang, David Lo, Xin Xia, Bowen Xu, Jianling Sun, and Shanping Li. 2015. Combining Software Metrics and Text Features for Vulnerable File Prediction. In *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia, December 9-12, 2015*. IEEE Computer Society, 40–49.