Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms

Dongyu Meng
dmeng@ucsb.edu
UC Santa Barbara

Michele Guerriero
michele.guerriero@polimi.it
Politecnico di Milano

Aravind Machiry
amachiry@purdue.edu
Purdue University

Hojjat Aghakhani
hojjat@ucsb.edu
UC Santa Barbara

Priyanka Bose
priyankabose@ucsb.edu
UC Santa Barbara

Andrea Continella
a.continella@utwente.nl
University of Twente

Christopher Kruegel
chris@cs.ucsb.edu
UC Santa Barbara

Giovanni Vigna
vigna@cs.ucsb.edu
UC Santa Barbara

ABSTRACT

Software is continually increasing in size and complexity, and therefore, vulnerability discovery would benefit from techniques that identify *potentially vulnerable* regions within large code bases, as this allows for easing vulnerability detection by reducing the search space. Previous work has explored the use of conventional code-quality and complexity metrics in highlighting suspicious sections of (source) code. Recently, researchers also proposed to reduce the vulnerability search space by studying code properties with neural networks. However, previous work generally failed in leveraging the rich metadata that is available for long-running, large code repositories.

In this paper, we present an approach, named Bran, to reduce the vulnerability search space by combining conventional code metrics with fine-grained repository metadata. Bran locates code sections that are more likely to contain vulnerabilities in large code bases, potentially improving the efficiency of both manual and automatic code audits. In our experiments on four large code bases, Bran successfully highlights potentially vulnerable functions, outperforming several baselines, including state-of-art vulnerability prediction tools. We also assess Bran's effectiveness in assisting automated testing tools. We use Bran to guide syzkaller, a known kernel fuzzer, in fuzzing a recent version of the Linux kernel. The guided fuzzer identifies 26 bugs (10 are zero-day flaws), including arbitrary writes and reads.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Static Analysis; Vulnerabilities; Machine Learning

ASIA CCS '21, June 7–11, 2021, Hong Kong, Hong Kong.
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8287-8/21/06.
https://doi.org/10.1145/3433210.3453115

ACM Reference Format:
Dongyu Meng, Michele Guerriero, Aravind Machiry, Hojjat Aghakhani, Priyanka Bose, Andrea Continella, Christopher Kruegel, and Giovanni Vigna. 2021. Bran: Reduce Vulnerability Search Space in Large Open Source Repositories by Learning Bug Symptoms. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21), June 7–11, 2021, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3433210.3453115

1 INTRODUCTION

Software is continually growing in both size and complexity. However, along with new products and features, such growth often comes with new security vulnerabilities, which, when abused by attackers, can drastically affect people's lives. For instance, a high severity vulnerability in Apache Struts, an open-source web development framework used by an estimated 65 percent of Fortune 100 companies, led to a data breach at Equifax that affected 147 million Americans [30].

Automated Vulnerability Discovery. Security researchers constantly propose new approaches to detect security vulnerabilities in source code before they are exploited by malicious attackers. Nevertheless, discovering security vulnerabilities takes time, given the size and complexity of modern software. Furthermore, once a vulnerability is discovered, more time is needed for the vendors to develop and release a patch, and for the patch to be deployed on the end systems. During this whole time, software is exposed to attacks. Thus, it is critical to promptly identify the vulnerable portions of code in large code bases.

However, most of the existing source code analysis tools are insufficient when directly applied to large code bases. In fact, static analysis approaches often need to make a trade-off between soundness and completeness when dealing with complex analysis targets [36, 44]. As a consequence, these approaches either produce too many false positives, making them impractical, or are not generic enough, preventing them from detecting more classes of bugs [38].

To address the aforementioned problems, researchers proposed techniques to guide vulnerability detection tools toward code areas that are likely to be vulnerable, reducing the amount of code to be analyzed. Previous work tried to identify potentially vulnerable code by leveraging software complexity metrics [46], or metrics related to software evolution and maintenance [54, 62]. Others tried to look at syntactic or semantic characteristics of source code that relate to security vulnerabilities [34, 50]. Recently, with the advancement of deep learning techniques, researchers have proposed several methods to automatically learn and extract bug-related features from the code [15, 24, 58]. Most of these methods focus solely on the code itself to come up with suspicious code sections, ignoring the crucial fact that any large, long-running software project is the collective effort of a large number of different engineers over a non-trivial time span. A new comer to a project is more likely to make mistakes than a core developer of the project; a code snippet being modified by a large group of engineers back-and-forth may incur inconsistency; a large commit spanning many components of a project often indicates high complexity, which may lead to a bug. In other words, the *human factors* buried in repository metadata can serve as an indicator for software quality. Luckily, most (if not all) open-source projects use version control systems (VCS), like Git, and have their code repositories publicly available. This gives us rich repository metadata for analysis. By observing the whole history of the studied repository, in part and as a whole, we are able to capture key factors that influence the code quality of the project, based on which we will be able to predict parts of code that are more likely to contain bugs.

Our Approach. We propose an approach to characterize potentially vulnerable code by examining *bug symptoms*, which is a collection of both code metrics directly observable from the code and the repository metadata that indirectly reflects code quality. Different from previous work on vulnerability detection, which models and identifies the characteristics of vulnerabilities, our approach focuses on the conditions in which vulnerabilities are born, and on the different types of symptoms that lead to their occurrence. Our key observation is that vulnerabilities arise from errors that are more likely to occur when either the code or the software development process present certain patterns. While the idea of leveraging code metrics and metadata is not novel, and it has been explored by several previous works, existing tools fail to fully leverage the power of such data, and they often lead to undesirable results.

To capture effective patterns and improve existing techniques, we devise four categories of features to describe functions in large code bases. The four categories focus on code complexity, use of security- or quality-related primitives, code commit history, and developer characteristics. We show that, using these features, we are able to train a machine learning model that effectively highlights bug-prone code sections.

We implemented our approach in Bran, a system that is able to learn different types of bug symptoms for large code bases, and that effectively predicts potentially vulnerable functions. Given a code base, its history, and the list of past, known vulnerabilities (i.e., CVEs), Bran leverages machine learning techniques to generalize the symptoms that characterize previously discovered bugs, providing a ranking of the functions that are more at risk for undiscovered vulnerabilities. In another word, Bran looks at the past and the present of a code base to predict its future.

Bran is not meant to identify the root cause of vulnerabilities, nor it is focused on specific classes of vulnerabilities. Instead, Bran is designed to assist security experts and automatic vulnerability detection tools in complex vulnerability auditing tasks by prioritizing high-risk code. In this way, it reduces the search space, making vulnerability discovery more practical.

We evaluated Bran on four real-world, large code bases (Linux, Wireshark, FFmpeg, and OpenSSL) showing that it is able to predict which functions are likely to be vulnerable. Finally, we used Bran as a pre-step for fuzzing a recent version of the Linux kernel. We provided syzkaller [13] with a list of 43,473 functions that Bran reported as likely to be vulnerable, and discovered 26 bugs, out of which 16 have been independently discovered by Google syzbot [19], and 10 are zero-days, which would have not been discovered without Bran's guidance.

Contributions. In summary, we make the following contributions:

- We propose an approach that extends previous works and characterizes potentially vulnerable code by modeling *bug symptoms*. Our key observation is that vulnerabilities are more likely to appear in code sections with bug symptoms. These symptoms are identifiable when one inspects software from the perspectives of both the code itself and its development process.
- Leveraging our approach, we design and develop a system, Bran, that is able to analyze and predict potentially vulnerable code in large code bases.
- We evaluate Bran on four large code bases, the Linux Kernel, OpenSSL, FFmpeg, and Wireshark. Results show that Bran can successfully predict unseen vulnerable functions. Our evaluation highlights potentially vulnerable functions in a recent version of the Linux Kernel, among which we identify 26 bugs (10 zero days).

In the spirit of open science, we make the code developed for this work publicly available at https://github.com/ucsb-seclab/bran.

2 APPROACH

To highlight the sections of code that are more likely to contain vulnerabilities, Bran takes two inputs: a database of CVEs, and an open-source code base managed with a version control system. Bran ingests the input code base and extracts two sets of functions: the set V of vulnerable functions across the whole history of the code base, identified using the CVE database, and the set C of all the functions in the current snapshot of the input code base. By examining CVE records, Bran builds a ground truth dataset of vulnerable functions (Section 3.1). However, we are not sure about whether any function in C is vulnerable or not. A function in C could be either safe or vulnerable. Fortunately, for any real-world code base, non-vulnerable functions outnumber vulnerable functions in C. This is especially true in the case of large, widely used code bases with a long history. In light of this observation, Bran can assume that a randomly sampled set of functions $NV \subset C$ is non-vulnerable.

Bran extracts code metric and repository metadata features from both function sets. Using V as positive examples and NV as negative examples, Bran trains a regression model that takes in a function feature vector and generates a vulnerability score. During vulnerability search, users may first use Bran to highlight the parts of code that are more likely to be buggy. Then, they may employ more computationally expensive tools, such as fuzzers, targeting only the highlighted code sections for more fine-grained vulnerability information. Users are able to tune the threshold of the vulnerability score based on their computational budget. With a suitable threshold, users get the chance to capture the majority of vulnerabilities while testing only a small fraction of the repository.

2.1 Code Metric Features

Intuitively, large, complex functions are more likely to have unforeseen corner cases; code that shows known bad practices (e.g., no sanity checks on input parameters, many casting operations) is more prone to bugs such as memory corruption issues.

Bran quantifies these intuitions using *code metric features* as summarized in Table 1 under category *Complexity* and category *Function Properties*. Table 1 provides a brief intuition of why these features can help capture bug symptoms. While each feature has its own limitations, and there exist corner cases where the application of a specific feature is not helpful, we argue that, on a large scale, their combination can result in accurate predictions, as demonstrated in our experiments.

Complexity. Complex functions, i.e., functions performing many operations through non-trivial control flows, are difficult to debug and maintain, and thus are more likely to contain bugs. To some extent, the complexity of a function is also related to the complexity of the module it belongs to. Functions that live in complex modules have a greater chance of encountering errors, which might introduce vulnerabilities [11, 47, 62, 63].

Function Properties. Certain function characteristics might indicate a higher probability of the presence of bugs. Functions may contain examples of bad coding practices (e.g., many casts (**F6**), or the lack of comments (**F9**) [6]), or they may present a large attack surface (e.g., a long parameter list (**F7**), or the lack of sanity checks (**F10**)).

2.2 Repository Metadata Features

While code metric features describe quality- and security-related properties of the code directly, we can also leverage the rich repository metadata information to derive useful signals for vulnerability prediction. We devise a set of repository metadata features, as summarized in Table 1 under category *History* and category *Developers & Reputation*, where we also provide a brief rationale behind the selection of each feature.

History. The history of a function also provides hints about the likelihood of the function containing bugs. A function or module that is frequently patched usually plays a critical role in the project. Frequent modification also makes code more prone to flaws (**F11**, **F12**) [49, 62]. In version control systems, such as Git, a commit is the basic unit for content update. Commits with radical changes (**F13**, **F14**) usually imply major code refactors, which may introduce new bugs or violate established conventions in the code. Commits that involve too many parts of different components (**F13**, **F14**) indicate a tightly coupled design in the project. With more moving parts, tightly coupled components are usually more bug-prone than independent ones.

We note here that bugs related to a hard-to-maintain commit are not necessarily introduced by the commit itself. It could be the commit before or after this commit that actually implants the bug. Therefore, predicting vulnerabilities using single commits as the basic unit [54] is not enough. We try to capsule bug symptoms like this by using functions as Bran's basic operation unit. A function carries consistent semantics throughout its lifetime in the development procedure. More importantly, most of the time, it honestly reflects the design changes in the code. By aggregating the history of a function, Bran is able to capture the correlation among multiple commits and program components.

Developers & Reputation. Traits of a developer may reflect the quality of their code. For instance, well-known or experienced developers generally write more reliable code. The trustworthiness of developers can be estimated through various indicators, such as their average number of contributions (**F21**, **F22**), and their popularity in the community (**F19**). Sometimes, collaboration among developers may also come into play. For example, having a larger number of developers contributing to the same function is against the separation of concerns and responsibilities principle, which, in turn, is an evidence of poor project management (**F17**, **F18**). In addition to the programmer information fetched from Github, we also estimate developers' expertise in terms of their contributions towards the repository being studied (**F24**), as some experienced engineers do not keep an active public profile. Feature (**F24**) also reflects the level of knowledge (familiarity) a developer has for the project.

We note here that the features used in this paper are not complete, and do not fit all scenarios.

2.3 Dataset

The dataset quality is essential to the performance of data-driven vulnerability prediction techniques. With inappropriate datasets, even advanced machine learning algorithms will be ineffective. For instance, small datasets may provide too little information for models to learn proper features automatically, and incorrectly labeled datasets give wrong feedback to models while training. All these defects in datasets may lead to poor performance on real code bases.

Past work on vulnerability-type agnostic, machine learning-based vulnerability prediction techniques relied on flawed datasets. Among them, Chernis and Verma [10] used only 100 vulnerable functions during training; VulSniper [15] used only two types of vulnerabilities in the SARD project [8]; Harer et al. [24] and Russell et al. [58] collected dataset from sources including GitHub [17] public repositories and Debian Linux distribution [12]. The size of their dataset is approximately 12 million functions. However, they labeled the dataset with static analyzers, like Cppcheck [2] and Flawfinder [3], which are known to have high false positive or high false negative rates (i.e., incorrect labeling).

In this paper, we make use of a dataset that comprises four large, real-world projects: the Linux kernel, OpenSSL, FFmpeg, and Wireshark. For each project, we collected the current code repository, its history, and its corresponding CVE reports. We labeled vulnerable functions based on CVE reports, and trained Bran to predict future vulnerabilities based on history knowledge. In this way, our dataset is able to provide Bran better knowledge about vulnerabilities with respect to previous work.

Table 1: Features computed by Bran, grouped by category.

	Feature	Description & Rationale
Complexity	(**F1**) Function lines of code (LOC)	Long functions are more difficult to debug and maintain, thus are more prone to bugs.
	(**F2**) Cyclomatic complexity	Complex functions are more prone to vulnerabilities. We measure the McCabe's complexity [40].
	(**F3**) Module LOC	LOC of the module that the function belongs to. Using large modules is not a good practice.
	(**F4**) Module complexity	Sum of the McCabe's cyclomatic complexity of all the functions in the function's module.
	(**F5**) Number of co-located functions	Number of functions in the function's module. Using large modules is not a good practice.
Function Properties	(**F6**) Number of casts	Casting is known to be a source of bugs.
	(**F7**) Number of input parameters	More function input parameters imply a larger attack surface.
	(**F8**) Number of local variables	Declaring many local variables is a sign that the function performs many tasks. Complex functions are more prone to vulnerabilities.
	(**F9**) Number of lines of comment	The lack of comments is a bad practice.
	(**F10**) Number of input sanity checks	Sanity checks on input parameters reduce the function's attack surface.
History	(**F11**) Number of past changes	Frequently patched functions may be poorly written and maintained, or play critical roles.
	(**F12**) Number of module's past changes	Frequently modified modules may contain critical tasks or be poorly written and maintained, leaving functions in a more error-prone environment.
	(**F13**) Size of second-largest commit addition	Commits with more lines of addition usually indicate more radical changes in the code. As the commit with the largest number of addition is mostly the first commit, we take the second largest commit addition number.
	(**F14**) Number of largest commit deletion	Commits with more lines of deletion usually indicate more radical change in the code.
	(**F15**) Max commit module span	A commit that spans across more modules tends to be more complex and therefore more bug-prone.
	(**F16**) Max commit function span	A commit that touches more functions tend to be more complex and therefore more bug-prone.
Developers & Reputation	(**F17**) Number of function's contributors	A reason for having a modular software is separation of concerns *and* of responsibilities. Too many developers contributing to the same function may cause issues.
	(**F18**) Number module's contributors	Too many developers contributing to a module may indicate bad team management.
	(**F19**) Avg. number of contributor followers	Well-known developers in the open-source community are generally more reliable. Open-source hosting services like GitHub allow developers to *follow* other developers.
	(**F20**) Avg. number of contributor stars	Open-source developers can *star* projects in public repositories to express appreciation. We consider the average number of stars received by all the contributors of a given function. Each contributor's stars are counted as the sum of the stars of all the public repositories they own.
	(**F21**) Avg. number of contributor forks	We consider the average number of forks of all the contributors of a given function, counted as the sum of the forks of all public repositories that they own.
	(**F22**) Avg. number of contributor repositories	Developers who frequently contribute to the open-source community are generally more experienced. We consider the average number of public contributions across all of the function's contributors.
	(**F23**) Avg. number of contributor watchers	Watching a repository means receiving notifications about activities of the repository. The number of watchers reflects the popularity of a project, which can be an estimation of the trustworthiness of its developers. Here, we consider the average number of watchers of all of the function's contributors.
	(**F24**) Min. number of contributor's contributions	The number of contributions within the project of the least experienced developer among all developers contributing to this function. Experience is measured as the number of commits a developer has made for the project.

3 IMPLEMENTATION

In this section, we discuss relevant aspects of Bran's implementation. We first outline the process used to extract vulnerable functions out of a code base, and then detail the implementation of Bran's machine learning model.

3.1 Inputs Processing

Unlike other automated vulnerability detection systems (e.g., a static analyzer), which may introduce a large amount of false positives into the dataset, the CVE database is a reliable source for vulnerability-related information. Therefore, as mentioned in Section 2.3, Bran relies on the CVE reports of the input code bases to create a ground truth dataset of vulnerable functions. In particular, Bran is able to enact an automated ground truth generation process for any software product *P* that: 1) is open-source; 2) uses git for version control; 3) is hosted on GitHub. Note that the same data preparation logistics can be adapted to other version control systems and hosting services as well.

To extract vulnerable functions, Bran performs the following nine steps:

1. Fetch a local copy of *P*'s code base with `git clone`.
2. Download lists of CVEs associated with *P* from the CVE public database [53].
3. For each CVE associated with *P*, refer to the National Vulnerability Database (NVD) for the git patch that fixed the vulnerability. The patch information (SHA-1 hash of a git commit) is not always available, as there might be missing data or simply the CVE might not have a patch yet.
4. If a given CVE has an associated fixing commit `c`, run `git checkout c` to retrieve the corresponding revision.
5. Extract all files modified by `c` and rename the patched files. For instance, a file with name `sample_file.c` gets copied to `sample_file_fixed.c`.
6. Run `git checkout c^` to retrieve the revision right before `c` was applied.
7. Extract and copy the non-patched version of all the files modified by `c` (e.g., `sample_file.c` is copied to `sample_file_unfixed.c`).
8. For each (**_unfixed.c*, **_fixed.c*) pair, identify functions that are modified by `c`.
9. Extract the body of each function identified from the corresponding unfixed file, i.e., the version of the function right before `c` was applied.

Following the steps above, we generate a dataset of vulnerable functions. Furthermore, we can configure the above procedure to extract vulnerable code considering only CVEs within a given time span (for example from 2012 to 2014) or considering only specific submodules of the code base.

It is worth mentioning that VUDDY [27] proposed a similar vulnerable code extraction process. The main difference between the two approaches lies in the way they identify commits associated to CVEs. While we rely on the National Vulnerability Database (NVD), allowing Bran to retrieve patches for CVEs precisely, VUDDY looks for commit messages containing the "CVE-20" string. It then filters out irrelevant commits (i.e., commits that contain "CVE-20" but did not fix any CVE) using a number of heuristics (e.g., excluding merging commits and reverting commits). VUDDY's heuristics-based approach may cause the resulting dataset to be inaccurate and incomplete. Indeed, irrelevant commits may be included in the dataset while relevant commits may be ignored.

In addition to the NVD, Bran allows for the integration of other sources that provide a mapping between CVEs and git commits. For example, in the case of the Linux kernel, we augment the NVD with the mapping provided by the `nluedtke/linux_kernel_cves` GitHub repository [52], which allows us to retrieve 168 more commits than those retrievable by solely relying on the NVD.

3.2 Learning Bug Symptoms

We compute code metric features and repository metadata features with a set of external tools and services. More specifically, we use Joern [72] to extract features (**F5**), (**F6**), (**F7**), (**F8**) and (**F10**). We feed Joern with the source code of all the functions for which context properties need to be extracted. Joern treats the set of functions as an entire program and produces the Program Dependency Graph (PDG). We then query the PDG to extract context properties. Through multiple `git` operations, we compute features (**F11**), (**F12**), (**F17**) and (**F18**). Features (**F13**-**F16**) and feature (**F24**) are extracted with the help of `git log`. Features (**F19**), (**F20**), (**F21**), (**F22**) and (**F23**) are computed by querying code bases through GitHub's public APIs [18]. These features are fairly straightforward counts of events or basic properties. Features (**F1**), (**F3**) and (**F9**) are collected directly from raw code files. Features (**F2**) and (**F4**) are computed with `pmccabe` [56], which provides a complexity score for functions and modules.

Code base context properties extraction is implemented as a parallel and distributed data pipeline on top of Apache Flink [1]. This allows us to efficiently process large code bases. We use random forest as the classification model for context properties representation, as this gives us a score which lies between 0 and 1, rather than a binary output. Although we also experimented with others classifiers (e.g., a neural network composed of a sequence of dense layers), we did not experience significant performance improvements. Using scikit-learn, Bran trains a random forest classifier over the vector representation of input functions. In particular, Bran uses the extremely randomized trees training algorithm. The number of trained tree is 1000.

4 EVALUATION

In this section, we evaluate Bran and demonstrate that, by studying both code metric and repository metadata features, Bran is able to effectively reduce the search space of vulnerability search. We organize this section around the following research questions:

RQ1: *Is* Bran *effective in predicting previously unseen vulnerabilities?* We evaluated the vulnerability prediction ability of Bran comparing to previous work. We considered four large real-world code bases: the Linux kernel, OpenSSL, FFmpeg, and Wireshark. Results indicate that Bran is effective in identifying previously unseen (i.e., not in the training set) vulnerabilities (Section 4.2). We compared our system to existing tools, demonstrating that it outperforms state-of-We made it clear, mostly in Section 1 and 7, that the idea of leveraging code metrics and meta the-art approaches (Section 4.3).

RQ2: *How does the non-vulnerable to vulnerable functions ratio in the training set affect machine learning-based methods?* The severe imbalance between vulnerable and non-vulnerable functions poses a practical challenge for machine learning-based vulnerability detection methods [14]. In search of a proper ratio for training set, we evaluated Bran on different non-vulnerable function to vulnerable function ratios. Results suggest that a severe imbalance in the training set does indeed harm performance. However, we show that effective machine learning models can still be trained considering only part of the data, overcoming the imbalance issue (Section 4.4).

RQ3: *Are learned bug symptoms unique to the training code base?*

Otherwise, are they generalizable to other code bases? Although the choice of a feature set is generically applicable to different code bases, whether the trained model on one code repository is effective for another repository is not clear. We evaluated models trained using one code repository and applied them to a different repository. Results show that bug symptom knowledge is generally not repository agnostic. (Section 4.5).

RQ4: *Is* Bran *effective in assisting large scale software testing?* We show one of the possible applications of Bran in the context of guided fuzzing (Section 4.6). We configured a fuzzer to prioritize functions marked as highly-vulnerable by Bran. In this way, Bran restricts the exploration space of the fuzzer. Experimental results show that the fuzzer guided by Bran significantly outperforms the unguided fuzzer.

4.1 Datasets

We evaluated Bran against four large, real-world projects: the Linux kernel, OpenSSL, FFmpeg, and Wireshark. For each of these projects, we considered the current code repository, its history, and the corresponding CVE reports.

To evaluate the performance of Bran, we have the following setup. Given the snapshot of the code base at time T in its history, we collect a *training set* for model training, and an *evaluation set* for performance evaluation. The *training set* is composed of the vulnerable functions that appeared in CVE reports before time T (as vulnerable examples), and a set of functions sampled from the code base snapshot at time T (as non-vulnerable examples). The data collection procedure is described at the beginning of Section 2.3. We tried to keep a balance between the vulnerable and non-vulnerable classes in the training set by controlling how many non-vulnerable functions we sample (Section 4.4). The *evaluation set* is used to evaluate the performance of machine learning models in finding vulnerabilities that were not known at time T. More specifically, to create the ground truth dataset of vulnerable functions, we look at the CVEs that have been reported *after* time T. The primary objective of our evaluation is to study how many of such functions are correctly predicted by Bran. The non-vulnerable functions in the evaluation set are all the functions belonging to the code base snapshot at time T, with the exclusion of those functions sampled to build up the training set and the functions that were included in the vulnerable set. Note that not all functions belonging to this set are guaranteed to be non-vulnerable. Some of them might actually be vulnerable, and the vulnerability has not yet been discovered.

To answer our research questions, we gathered different *training sets* and *evaluation sets* under two settings:

Setting I. T = end of 2015: In this setting, we use knowledge up to the end of year 2015 to predict vulnerabilities that were discovered in 2016 or later. By setting T back to 2015, we were able to accumulate enough CVEs to build a vulnerability ground truth for the evaluation set. We answer questions **RQ1**-**RQ4** with experiments under this setting.

Setting II. T = end of 2018: In this setting, we used knowledge up to the end of year 2018 to predict vulnerabilities that were not known at that point in time. In this setting, we do not have a sufficiently large ground truth of vulnerable functions in the evaluation set, as only few vulnerabilities have been reported since the end of 2018. Therefore, the only way to confirm Bran's predictions is to check them with testing tools ourselves. We use this setting to answer question **RQ4**.

The size of each dataset generated in the two settings is reported in Table 2. For each code base studied, we balanced the non-vulnerable-to-vulnerable function ratio to 6 for the training set (including validation) (see Section 4.4). In this paper, we use the output of the classifier as a vulnerability score for the input function. Higher scores indicate a better chance to find a vulnerability in the input function.

4.2 Prediction Capabilities

In this section, we use **Setting I** to answer **RQ1** about Bran's ability to predict vulnerable functions. Under this setting, we are interested in how many of the functions that are found to be vulnerable *after* 2015 are correctly predicted by models trained with the knowledge available *up to* 2015. Formally, let VP be the set of functions that are predicted as vulnerable by Bran and LV be the set of vulnerable functions in the evaluation set; we calculate the *recall* of Bran on the set LV with

$$recall = \frac{|VP \cap LV|}{|LV|}$$

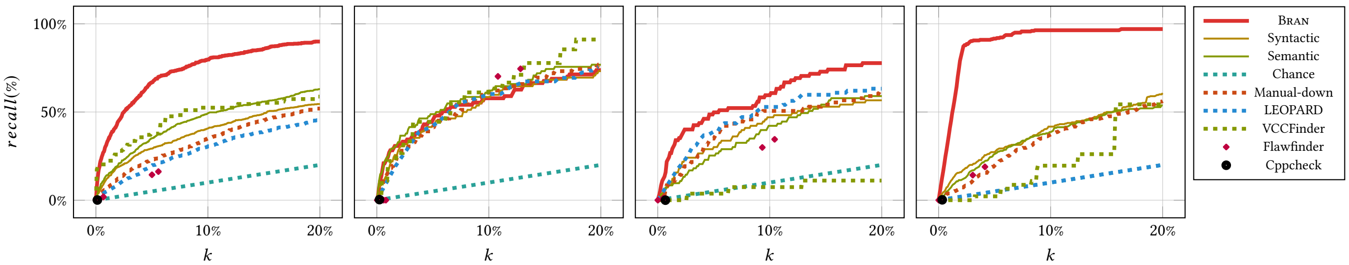
as the primary metric for effectiveness. Note that, since some bugs in the code base remain unidentified, *recall* in this experiment is an underestimation of Bran's performance.

A key aspect of computing *recall* is how we define VP, i.e., based on what criteria Bran judges a function to be vulnerable. Bran assigns a vulnerability score (a probability of being vulnerable) to each function. Therefore, naturally, we can set a vulnerability score threshold γ for each code base, and label functions with a score higher than γ as vulnerable. How to pick a suitable γ, however, is not trivial. In the scenario of large-scale automated vulnerability discovery, the number of functions that one is able to inspect depends on the available computing resources. Therefore, the objective of Bran is to maximize *recall* given the maximum size of VP as imposed by the available computing budget. We evaluate *recall* for different values of γ to simulate various computing budgets. More specifically, we ranked all functions according to their vulnerability score and considered the highest $k\%$ as the VP set. We set k to $[0, 20]$ and $[0, 1]$ and show the results of all sub-models together with Bran combined model in Figure 1 and Figure 2, respectively.

As shown in Figure 1, with the help of Bran, by inspecting only the top 20% functions in a code base, security experts are able to cover more than 70% of the vulnerabilities in all the four considered

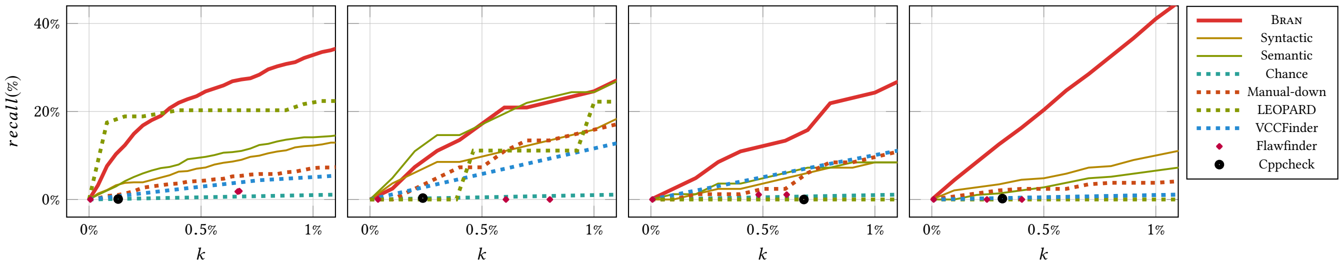
Table 2: Number of functions in each dataset.

			Dataset			
T	Set	Class	Linux	OpenSSL	FFmpeg	Wireshark
2015	Training	Vuln	1,473	91	167	127
		Non-Vuln	8,838	546	1,002	762
	Evaluation	Vuln	927	128	107	388
		Non-Vuln	257,290	4,965	14,081	24,419
2018	Training	Vuln	2,400	219	274	515
		Non-Vuln	14,400	1,314	1,644	3,090
	Evaluation	Both	271,624	6,243	16,377	25,367



(a) Linux (b) OpenSSL (c) FFmpeg (d) Wireshark

Figure 1: Models trained using data from a single code base and then used to rank functions in the same code base: recall on predicting vulnerable functions over top k% of ranked functions. $k \in [0, 20]$.



(a) Linux (b) OpenSSL (c) FFmpeg (d) Wireshark

Figure 2: Models trained using data from a single code base and then used to rank functions in the same code base: recall on predicting vulnerable functions over top k% of ranked functions. $k \in [0, 1]$.

code bases. For more time-consuming human inspections, users can conveniently set the threshold of Bran to a higher value and only invest in high-confidence suspicious functions. As Figure 2 shows, by inspecting only 1% of the code base, Bran achieved more than 20% recall of vulnerabilities on all four code bases tested.

We conclude that Bran is able to effectively distinguish vulnerable functions from non-vulnerable ones.

Precision Analysis. Bran is not designed to pinpoint bugs as a stand alone tool. Instead, it assists other vulnerability discovery tools in reducing their search space (as we will demonstrate in Section 4.6). While classification problems are generally evaluated with both recall and precision, avoiding a high false positive rate in our case is particularly challenging but not rewarding. We therefore choose not use precision as a key metric of the effectiveness of Bran.

4.3 Comparison with Existing Tools

To further assess the performance of Bran, we compared it against existing tools for source code vulnerability prediction.

Code metric based approaches. We compared Bran against two code metric baselines, namely the manual-down approach and LEOPARD [14]. The manual-down approach ranks functions by their source lines of code in descending order and labels longer functions to be more prone to bugs. Simple as it is, past work has shown that manual-down approach achieves similar or even better results compared to state-of-art vulnerability prediction models [14, 29, 83]. LEOPARD is a recent tool that leverages both complexity and vulnerability metrics in code defect detection. It first categorizes target functions into a set of bins based on the complexity metrics, then ranks functions in each bin by their vulnerability metrics (e.g., number of pointer operations), and labels the top n functions in each bin as vulnerable.

We implemented the manual-down approach ourselves, and the authors of LEOPARD kindly tested LEOPARD on our dataset. Manual-down processed all functions successfully; LEOPARD failed to process the Wireshark dataset according to the LEOPARD authors. The results of both baselines are shown in Figure 1 and Figure 2. In Figure 1 where the top k% ($k \in (0, 20]$) functions were considered vulnerable, Bran's outperformed metrics baselines in the Linux, FFmpeg, and Wireshark datasets and got comparable results in the OpenSSL dataset. In the high-confidence zone, where only the top 1% functions were inspected (Figure 2), Bran achieved twice the recall of any of the tools on three of the four code bases. Similar to the baseline techniques, Bran also used code metric features for classification. However, in addition to the code metrics employed by both manual-down and LEOPARD, Bran further takes the repository metadata features into consideration. With richer context features, Bran is able to predict vulnerabilities more effectively.

Static analyzers. We ran two popular static source code analyzers, namely Flawfinder [3] and Cppcheck [2], on our **Setting I** dataset as baseline comparisons. Both Flawfinder and Cppcheck examine C/C++ source code for possible bugs and undefined behaviors by matching a built-in database of known problems. Nevertheless, Bran and such static analyzers are designed differently. Bran assigns a continuous score to each analyzed function, effectively producing a ranking. It is then up to the users to set a threshold score according to their time and computational budgets to further analyze only the top k% of the functions. On the other hand, static analyzers make binary decisions for matches, such that a function is either suspicious (i.e., leads to warnings or errors being raised) or safe. This still leads to only k% of the input functions being further analyzed by security experts, with k being the ratio between the number of reported suspicious functions and the total number of analyzed functions. In both the cases, given k% of the input functions, we use recall, i.e. how many of these suspicious functions are actually vulnerable, as the performance measure. For Cppcheck, we only have one single value for k, which reflects the number of reported suspicious functions. Flawfinder allows users to choose from 5 *risk levels*, which, in turn, gives 5 different values of k in the test.

Results for Cppcheck and Flawfinder are shown in Figure 1 and Figure 2. Bran outperformed both baselines by a large margin in all code bases except the OpenSSL dataset. For low-budget inspection assistance, where only the top 1% functions are considered interesting, both analyzers gave near-zero recall, while Bran was able to provide a recall of more than 30%. One interesting observation is that Cppcheck exhibited only near-zero recall for all code bases. This might be due to the fact that while Cppcheck strives to achieve a low false positive rate, it inevitably suffers from low true positive rates [2]. We note here that we are aware that it seems unfair to compare Flawfinder and Cppcheck to Bran in terms of recall as they are designed to match vulnerabilities in a much more fine-grained way. However, these are the ones among others in this category that are close to Bran's design goal. We believe that the comparison adds interesting perspectives to the discussion.

We conclude that current static analyzers only match known patterns and are sub-optimal in the setting of vulnerability prediction compared to Bran.

VCCFinder We compared Bran with VCCFinder [54], a similar work that also leverages code-metrics and repository meta information for bug finding. We replicated VCCFinder on top of PyDriller [65], trained and tested it against the same repositories used for Bran. VCCFinder uses commits as the basic unit for analysis. It aims to find vulnerability-contributing commits (VCC) and does not have the notion of a function being vulnerable. However, in order to have a fair comparison with Bran, we tag all the functions that are modified as a part of a commit marked by VCCFinder, as vulnerable. The recall is calculated based on function numbers instead of commit numbers. As opposed to Bran, VCCFinder is designed to be trained on a mixture of repositories. We trained VCCFinder on the mixture all four repositories and tested it on each repository individually.

Figure 1 and Figure 2 showcase the results for VCCFinder and Bran with respect to performance. Note that out of the four repositories, Bran outperformed VCCFinder in three and got comparable results in one of them i.e., OpenSSL. Although Bran and VCCFinder both examine code-metrics and repository metadata as features, they approach similar abstractions from different perspectives. For instance, to describe the complexity of a basic programming unit, VCCFinder focuses more on the patch and does not fully utilize the information about the location of the patch. However, Bran captures function and module complexity naturally but may have trouble modeling commit information in a straightforward way. It can be an interesting open problem to analyze which way of modeling is more suitable for the task of bug finding. However, as we observe in the experiments shown above, Bran clearly has an advantage.

VCCFinder outperformed other baselines and tools tested on the Linux kernel and OpenSSL repository, but failed to achieve comparable results on FFmpeg and Wireshark. One possible explanation of VCCFinder's mixed performance can be attributed to the fact that it is designed to be trained on a composition of different repositories(both smaller and larger), and is likely to over-fit to larger code bases. This may results in both positive or negative effects to a different code base depending on how similar it is to the dominating code bases used during the training phase.

Automatic feature learning. With the recent advancement of deep learning, researchers also propose to locate suspicious code sections by studying code properties with neural networks. VulSniper [15] and Russell et al. [24, 58] leveraged neural networks for type-agnostic bug finding, which is similarly to Bran. We contacted both groups for help in reproducing their results on our dataset but unfortunately we were turned down.

To compare Bran to automatic feature learning methods, we designed and implemented two neural network models based on previous work, and compared them to Bran. The first model extracts syntax pattern [34] of functions with tree-shaped convolutional neural network [48] by abstract syntax tree vectorization. We refer to this model as the *syntax model* for the remainder of this paper. The second model captures semantically relevant vectorial representation of functions. More specifically, it leverages the word2vec [45] approach from the natural language processing (NLP) community and embeds the functions as a paragraph [58]. We refer to this model as the *semantic model* for the remainder of this paper. The word *semantic* here is borrowed from NLP. It does not refer to *program semantics* as defined in the programming language community. Both models are trained on the same dataset as Bran as binary classifiers.

In the syntax model, the convolution layer of the syntactic model deployed 128 kernels (feature detectors). The classification subnetwork was a fully connected network with one hidden layer of width 20. The model was trained for 30 epochs with batch size 16. The learning rate was set to 0.001. In the semantics models, the one-dimensional convolution layer deployed 16 kernels, each of the size of 3 words, and used a ReLU activation function. The LSTM layer was initialized using 50 memory units. The final dense layer used a sigmoid activation function. The model was trained for 5 epochs with batch size of 50. In our experiments, different configurations did not show significant performance improvements.

For the high-budget scenario shown in Figure 1, Bran achieved similar results in the OpenSSL dataset comparing to both the syntax and the semantic model. For the other three datasets, Bran greatly outperformed both models. In the low-budget case, as shown in Figure 2, Bran beat the syntax model in all four datasets. The semantic model displayed comparable result as Bran on the OpenSSL dataset, but only got half the recall on the other three datasets.

As a side note, both neural-network-based models generally outperformed the two code-metric-based methods.

4.4 Training Set Balance

Known vulnerable functions are rare in real-world projects, compared to the total number of functions in a code base. This imbalance between vulnerable and non-vulnerable code hinders the

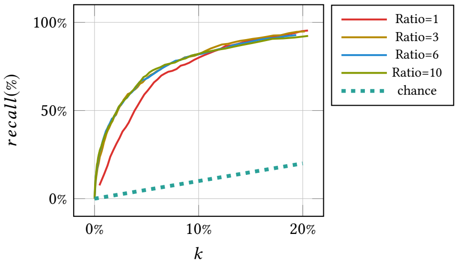


Figure 3: Model performance with various non-vulnerable function to vulnerable function ratio in the training set: recall on predicting vulnerable functions among the top k% of ranked functions.

adaptation of machine-learning-based methods for vulnerability detection [14]. The reason is that an imbalance in training data often results in poor classification accuracy.

In practice, fortunately, we do not have to stick to the actual ratio between non-vulnerable function and vulnerable function (denoted as R) in the code base when it comes to training. For Bran, we tuned R in the training set by sampling only a part of non-vulnerable functions. Empirically, we tested several R values to search for the best R setup for training.

Figure 3 shows how data balance in the training set R influences vulnerability detection performance for Bran. We found that models trained with R between 3 and 6 generally outperform other models by a notable margin. Much higher or too low R values have negative effects on the classification performance. The results also indicate that machine learning based methods are in fact practical in use cases where vulnerabilities are very sparse.

4.5 Bug Symptoms Knowledge Transfer

An interesting question to consider when modeling bug symptoms is whether such symptoms are code-base-dependent. If bug symptoms are shared across code bases, it is possible to use models trained on one code base to predict vulnerabilities in other ones. In other words, if bug symptoms are *transferable* across code bases, the resulting machine learning models would be generic, drastically widening their scope of applicability.

To assess how well bug symptoms generalize, we repeated the experiment discussed in Section 4.2 (again under **Setting I**). However, this time, instead of using models trained on one code base to predict future vulnerabilities in the same code base, we used models trained on a different code base. For example, we used the model learned from Linux using context properties to predict vulnerable functions on OpenSSL, FFmpeg, and Wireshark. We did this for all 12 transfer pairs (training code base -> prediction code base). The results are shown in Figure 4. Each subplot also reports, as a baseline, the performance of the *native* model, whose training code base is the same as its prediction code base.

From Figure 4 we notice that, for half of the target repositories, native models delivered the best performance, while for the other half of the target repositories, one (and only one) of the transferred models outperformed the native model. Considering all transfer pairs tested, most of the time transferred models had poor performance. Sometimes the transferred model was even worse than chance (e.g., Linux -> Wireshark). This indicates that code metrics and repository metadata features like traits of developers are highly correlated to the code base being analyzed.

To conclude, bug symptoms learned from one code base have a certain level of ability to be transferred to a different one. However, in most cases, using the symptoms learned from the same repository derives better results.

4.6 Bran-guided Fuzzing

In this section, we want to understand the effectiveness of Bran in helping automated testing tools to find new bugs in very large and well-tested code bases. Thus, we present a real-world scenario where Bran can be directly applied. Specifically, we use syzkaller [13], a widely used Linux kernel fuzzer, in two configurations:

syzkaller: The recommended configuration as suggested in the official documentation [70].

syzkaller$_{\mathbf{bp}}$: syzkaller guided toward fuzzing functions marked with high confidence as vulnerable by Bran.

To configure *syzkaller*$_{bp}$, we first obtained all the functions that Bran marked as vulnerable with more than 80% confidence. These is our initial set of functions (I). We want the fuzzer to generate inputs that have a high likelihood of exercising such functions. An input from user space can reach a function f in I either directly (if f is a `syscall` [7]) or through one of its callers in the call-graph. To create the static call-graph, we first compile the Linux kernel with LLVM [32] to get bitcode files of all the source files. Second, we use a custom script that takes all the bitcode files and creates a call-graph by mimicking the linking process ignoring indirect calls.

Given the call-graph [59] of the Linux kernel and the initial functions (I), we obtained all the additional functions (A) that can reach, in the call-graph, a function in I.

Any input that reaches a function in A has the potential to reach a function in I. Thus, we added the functions in A to the list of the functions used to guide syzkaller. Doing so, we obtained a list of $T = I \cup A \approx 100K$ functions—compared to the $\approx 500K$ functions in the Linux Kernel. However, the actual number of functions that get compiled in the kernel is much less and depends on the corresponding kernel configuration [81]. This reduces T, the list of functions that need to be fuzzed, to 43,473 functions. Finally, we modified the `kcov` [69] pass in `gcc` [20] to instrument only the functions T, so that syzkaller prioritizes the inputs that have a high likelihood of exercising the target functions.

We ran our customized *syzkaller*$_{bp}$ configuration for 48 hours. On the other hand, to reduce any non-deterministic effect and to perform a stronger comparison, we ran the default *syzkaller* configuration for twice as long, i.e., 96 hours. Note that, we also compared our findings to Google syzbot [19], which performs continuous analyses. Table 3 shows the total number of crashes we found using the two fuzzer configurations on the latest version of the Linux kernel (April 2019). In total, the fuzzer guided by Bran (*syzkaller*$_{bp}$) found, in half the time, 26 bugs compared to only 2 bugs found by the default configuration (*syzkaller*). This shows that Bran is effective in guiding fuzzers to find *more* bugs *quickly*. Indeed, 18 (out of 26) functions where we found crashes were actually *predicted by Bran to be vulnerable with 100% confidence*. Among such 26 bugs, we found that 16 of them have been independently

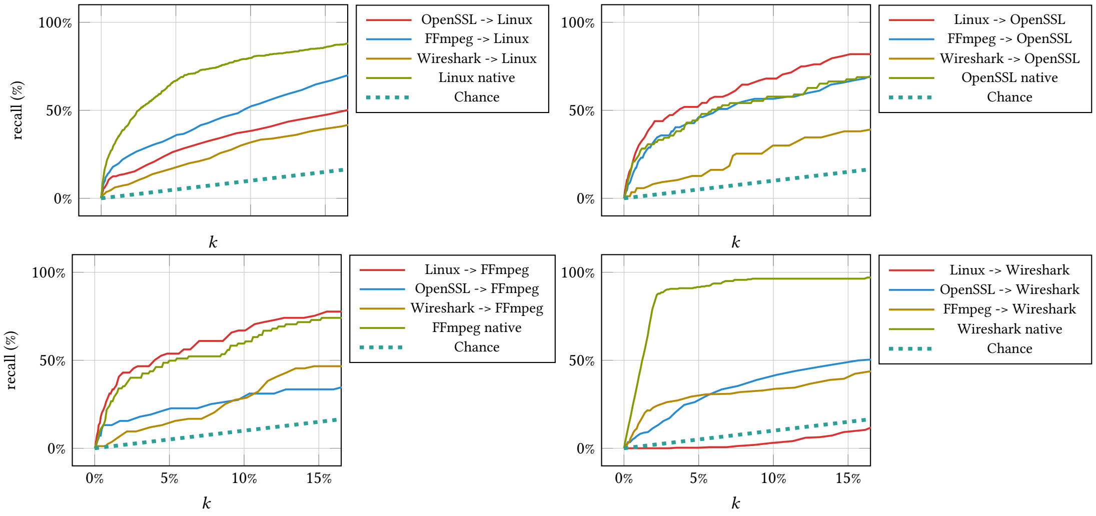


Figure 4: Models trained using data from one source code base and used on a different target code base (source -> target): recall on predicting vulnerable functions among the top k% of ranked functions.

discovered by Google syzbot [19], and 10 are previously unknown bugs. Note that, at the time of our evaluation, only 13 bugs were known, while 3 more bugs have been discovered by Google syzbot later on. We manually confirmed the discovered bugs and reported all the unknown bugs to the Linux Kernel core developers.

5 DISCUSSION AND LIMITATION

Ground Truth Dataset. Despite our efforts to provide Bran with appropriate ground truth during the training phase, the collected datasets might still bear some inaccuracies, which might introduce a bias in Bran's predictions, and in turn affect its effectiveness. In particular, when extracting vulnerable functions from CVE reports, we label all functions modified in the patching commit as vulnerable. However, in some cases a patch modifies multiple functions, but only some of them are directly related to the actual root of the vulnerability. In future work, we may refine our procedure to precisely identify vulnerable functions by leveraging other available sources of information, e.g., CVE descriptions and commit messages [80].

Similarly, to label non-vulnerable functions, we rely on the assumption that, given a snapshot of a large code base, the vast majority of the functions can be assumed to be safe. Thus, a random sample of the functions in a code base can be used as a ground truth for non-vulnerable functions. Inevitably, such a ground truth might contain functions that are indeed vulnerable but are not discovered yet, introducing inaccuracies. This could also be affected by the fact that not all the developers and security researchers apply for CVEs.

Table 3: Results of guided fuzzing (Section 4.6).

Type of crash	Number of crashes	
	syzkaller (96 hrs)	**syzkaller$_{\mathbf{bp}}$** (48 hrs)
Arbitrary memory write	0	16
Arbitrary memory read	2	5
Null-pointer-deref	0	1
Use-after-free	0	4
Total	**2**	**26**

Distinguish Types of Bugs. By design, Bran is only able to detect potentially vulnerable functions, without providing information about the type of vulnerability. Future work could address such limitation by extending Bran to distinguish different classes of bugs. However, this would require a labeled training dataset.

Precision and Recall. Bran incurs a relatively high false positive rate compared to stand-alone bug finding tools, such as Flawfinder and Cppcheck. However, as a search space reduction tool, Bran is not designed to achieve high precision. When being compared to tools like LEOPARD with similar design objectives, Bran delivered both higher precision and higher recall.

Identification of Functions. Bran uses function as the basic unit for repository metadata analysis, and relies on the pair of file path and function name as the unit identifier. This abstraction introduces inaccuracy when the function or the file is refactored to a different name. We believe that the accuracy of the analysis can be further improved by tracking renames and moves. For the purpose of this paper, however, this simplified abstraction already works well enough.

Cold Start Problem. Bran may suffer from the cold start problem where new functionalities in the project may have fewer metadata items upon which to base the analysis, compared to the long-running ones, causing Bran to have a slight bias towards code with longer history. While we admit that Bran is likely to have an systemic bias favoring older functionalities, we point out that this bias is reasonable as there is simply more chance of making mistakes for functionalities with longer history. Also, as most of the features that Bran uses (all of the code metric features and many of the repository metadata features - e.g., author reputation) are in fact unbiased against new functions, Bran should be able to handle newly added functionalities reasonably in practice.

Assembly Code. Source code written in C/C++ may contain hand-written assembly code. However, the current implementation of Bran does not support assembly. Providing such support might require adapting some of the features that Bran uses.

Feature Selection & Future Work. We acknowledge that there exist scenarios in which some of our features might not be effective (e.g., functions with less parameters might indicate that the function uses globals, which can lead to insecure concurrent accesses). However, we argue that, on a large scale, the combination of our features can lead to effective predictions—as validated in our experiments. Nonetheless, to alleviate the manual feature engineering process and explore new metrics that can capture bug symptoms, we plan to leverage deep learning techniques in future work. This, however, comes with the cost of producing models that are difficult to explain.

6 RELATED WORK

Pattern-based. Pattern-based approaches make use of already-known vulnerability patterns to find potential vulnerable code. Some of the popular tools are Flawfinder [3], PScan [4], RATS [5], and ITS4 [68]. Though these tools are very efficient and practical, they incur very high false positive and false negative rates, because they often fail to identify complex vulnerability patterns. Since then, several efforts have been made to build more advanced static analysis techniques for pattern-based vulnerability detection [22, 31, 37, 64, 67, 71, 73, 75, 77] . However, these techniques rely on previous knowledge of vulnerability patterns, and thereby fail to discover new ones. In addition, there are approaches [41, 66, 76] that aim to find vulnerabilities automatically. However, they target only specific types of vulnerabilities, e.g., taint-style vulnerabilities [41, 76] or missing-check vulnerabilities [66]. On the contrary, our approach is agnostic to specific types of vulnerability.

Code Similarity-based. These approaches [25, 28, 33, 55, 60] start by dividing the entire program into code fragments, which are then converted into abstract representation, e.g., tokens [25, 28, 60], trees [26, 55], or graphs [33, 55]. Lastly, those abstracted code fragments are used to infer similarities between them. The main advantage of this kind of approach is that it can find similar occurrences of vulnerabilities from a single instance. However, these approaches heavily rely on the fact that two semantically similar code fragments are not significantly different from a syntax perspective, which could be false. Instead of finding vulnerable patterns, which cannot be modeled exhaustively, our approach focuses on the conditions that lead to vulnerabilities.

ML & Metric-based. Prior work [9, 10, 21, 23, 24, 34, 35, 39, 57, 58, 74] attempted to apply various machine learning techniques for vulnerability prediction at the source-code level. In addition, other work [11, 16, 46, 50, 51, 61, 63, 78, 79, 82] combined machine learning with software quality metrics, code churns, or token frequency metrics to identify vulnerable code. A recent paper [14] leverages complexity and vulnerability metrics to identify potentially vulnerable functions. Bran is different from these works, as Bran considers repository metadata features in addition to code metrics features, which, as we showed in our experiments, significantly enhance prediction capabilities.

Repository-based. There is a line of research investigating the correlation between repository metadata and software defects [42, 43, 50, 54]. Among them, VCCFinder [54] is the most similar work compared to Bran. VCCFinder tries to locate vulnerability-contributing commits (VCC) incrementally by learning from code metrics and repository meta-information within the commit, using SVM classifiers. Unlike VCCFinder, Bran operates on functions as the basic analysis unit. By doing this, Bran is able to correlate different commits instead of focusing only on one commit at a time. Second, Bran introduces new additional classes of features (e.g., *Developers & Reputation*, Table 1) that have not been explored by VCCFinder and that showed to be effective in identifying potentially vulnerable code.

7 CONCLUSIONS

In this paper, we discussed the need for methods and tools aimed at assisting real-world security testing of large code bases by predicting potentially vulnerable portions of code. To do so, we presented an approach that analyzes code by leveraging both code metric features and repository metadata. We extended existing techniques by introducing new features and we implemented our approach in Bran, a tool that is able to highlight potentially vulnerable functions in large code bases. Bran relies on CVE reports and machine learning techniques to train classifiers that are able to identify bug symptoms, which capture patterns from both the code and the software development process perspectives. By evaluating Bran against four large code bases, we showed that it is able to effectively predict unseen vulnerable functions. Experimental results also show that although learned bug symptoms have a certain level of ability to transfer from one code base to another, it is best practice to use learned bug symptoms from the same project. Finally, we leveraged Bran's results to guide a Linux kernel fuzzer, discovering 10 zero-day bugs that would have not been discovered without Bran's guidance.

REFERENCES

[1] Apache flink project. https://flink.apache.org/. (Used version: 1.6.0).

[2] Cppcheck - a tool for static c/c++ code analysis.

[3] flawfinder - lexically find potential security flaws ("hits") in source code.

[4] Pscan, http://deployingradius.com/pscan/, 2017.

[5] Rats, https://code.google.com/archive/p/rough-auditing-tool-for-security/, 2017.

[6] and and. Improving vulnerability prediction accuracy with secure coding standard violation measures. In *2016 International Conference on Big Data and Smart Computing (BigComp)*, pages 115–122, Jan 2016.

[7] Maurice J Bach et al. *The design of the UNIX operating system*, volume 5. Prentice-Hall Englewood Cliffs, NJ, 1986.

[8] Paul E. Black. SARD. https://samate.nist.gov/SARD/.

[9] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36:7346–7354, 05 2009.

[10] Boris Chernis and Rakesh Verma. Machine learning methods for software vulnerability detection. In *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics*, IWSPA '18, pages 31–39, New York, NY, USA, 2018. ACM.

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

[12] Debian. Debian - the universal operating system. https://www.debian.org/.

[13] David Drysdale. Coverage-guided kernel fuzzing with syzkaller. *Linux Weekly News*, 2:33, 2016.

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

[15] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In *IJCAI*, 2019.

[16] Wei Fu and Tim Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 72–83, New York, NY, USA, 2017. ACM.

[17] GitHub. GitHub. https://github.com.

[18] GitHub. Github's public apis. https://developer.github.com/v3/. (Accessed: 2019-01-15).

[19] Google. syzbot. https://syzkaller.appspot.com/.

[20] Arthur Griffith. *GCC: the complete reference*. McGraw-Hill, Inc., 2002.

[21] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct 2005.

[22] Brian Hackett, Manuvir Das, Daniel Wang, and Zhe Yang. Modular checking for buffer overflows in the large. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 232–241, New York, NY, USA, 2006. ACM.

[23] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, 38(6):1276–1304, Nov 2012.

[24] Jacob A. Harer, Louis Y. Kim, Rebecca L. Russell, Onur Ozdemir, Leonard R. Kosta, Akshay Rangamani, Lei H. Hamilton, Gabriel I. Centeno, Jonathan R. Key, Paul M. Ellingwood, Marc W. McConley, Jeffrey M. Opper, Sang Peter Chin, and Tomo Lazovich. Automated software vulnerability detection with machine learning. *CoRR*, abs/1803.04497, 2018.

[25] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[26] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105, May 2007.

[27] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.

[28] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.

[29] A Güneş Koru, Khaled El Emam, Dongsong Zhang, Hongfang Liu, and Divya Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473, 2008.

[30] Brian Krebs. Experts Urge Rapid Patching of Struts Bug, 2018. https://krebsonsecurity.com/2018/08/experts-urge-rapid-patching-of-struts-bug/.

[31] J. R. Larus, T. Ball, , R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May 2004.

[32] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[33] J. Li and M. D. Ernst. Cbcd: Cloned buggy code detector. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 310–320, June 2012.

[34] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. Sysevr: A framework for using deep learning to detect software vulnerabilities. *CoRR*, abs/1807.06756, 2018.

[35] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *CoRR*, abs/1801.01681, 2018.

[36] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, January 2015.

[37] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[38] Aravind Machiry, Chad Spensky, Jake Corina, Nick Stephens, Christopher Kruegel, and Giovanni Vigna. {DR}.{CHECKER}: A soundy analysis for linux kernel drivers. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1007–1024, 2017.

[39] Ruchika Malhotra. A systematic review of machine learning techniques for software fault prediction. *Appl. Soft Comput.*, 27(C):504–518, February 2015.

[40] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976.

[41] Ibéria Medeiros, Nuno Neves, and Miguel Correia. Dekant: A static analysis tool that learns to detect web application vulnerabilities. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 1–11. ACM, 2016.

[42] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez Tejeda, Matthew Mokary, and Brian Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 65–74. IEEE, 2013.

[43] Andrew Meneely, Alberto C Rodriguez Tejeda, Brian Spates, Shannon Trudeau, Danielle Neuberger, Katherine Whitlock, Christopher Ketant, and Kayla Davis. An empirical investigation of socio-technical code review metrics and security vulnerabilities. In *Proceedings of the 6th International Workshop on Social Software Engineering*, pages 37–44, 2014.

[44] Bertrand Meyer. Soundness and Completeness: With Precision, 2019. https://cacm.acm.org/blogs/blog-cacm/236068-soundness-and-completeness-with-precision/fulltext.

[45] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[46] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013:8–17, 05 2013.

[47] Sara Moshtari, Ashkan Sami, and Mahdi Azimi. Using complexity metrics to improve software security. *Computer Fraud & Security*, 2013(5):8–17, 2013.

[48] Lili Mou, Ge Li, Yuxuan Liu, Hao Peng, Zhi Jin, Yan Xu, and Lu Zhang. Building program vector representations for deep learning. *CoRR*, abs/1409.3358, 2014.

[49] J. C. Munson and S. G. Elbaum. Code churn: a measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31, Nov 1998.

[50] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 529–540, New York, NY, USA, 2007. ACM.

[51] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, MetriSec '10, pages 3:1–3:8, New York, NY, USA, 2010. ACM.

[52] nluedtke. Linux kernel cves. https://github.com/nluedtke/linux_kernel_cves. (Accessed: 2019-01-15).

[53] Serkan Ozkan. Cve details: The ultimate security vulnerability datasource. https://www.cvedetails.com. (Accessed: 2019-01-15).

[54] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 426–437, 2015.

[55] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 447–456, New York, NY, USA, 2010. ACM.

[56] pmccabe. Mccabe-style function complexity and line counting. https://people.debian.org/~bame/pmccabe/pmccabe.1. (Used version: 2.6).

[57] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics. *Inf. Softw. Technol.*, 55(8):1397–1418, August 2013.

[58] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. Automated vulnerability detection in source code using deep representation learning. *CoRR*, abs/1807.04320, 2018.

[59] Barbara G Ryder. Constructing the call graph of a program. *IEEE Transactions on Software Engineering*, (3):216–226, 1979.

[60] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1157–1168, New York, NY, USA, 2016. ACM.

[61] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, Oct 2014.

[62] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.*, 37(6):772–787, November 2011.

[63] Yonghee Shin and Laurie Williams. An empirical model to predict security vulnerabilities using code complexity metrics. In *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '08, pages 315–317, New York, NY, USA, 2008. ACM.

[64] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. Rolecast: Finding missing security checks when you do not know what checks are. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 1069–1084, New York, NY, USA, 2011. ACM.

[65] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 908–911, New York, New York, USA, 2018. ACM Press.

[66] Lin Tan, Xiaolan Zhang, Xiao Ma, Weiwei Xiong, and Yuanyuan Zhou. Autoises: Automatically inferring security specifications and detecting violations. In *Proceedings of the 17th Conference on Security Symposium*, SS'08, pages 379–394, Berkeley, CA, USA, 2008. USENIX Association.

[67] J. Vanegue and S. K. Lahiri. Towards practical reactive security audit using extended static checkers. In *2013 IEEE Symposium on Security and Privacy*, pages 33–47, May 2013.

[68] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. Its4: a static vulnerability scanner for c and c++ code. In *Proceedings 16th Annual Computer Security Applications Conference (ACSAC'00)*, pages 257–267, Dec 2000.

[69] Dmitry Vyukov. gcc: add fuzzing coverage, 2016. https://codereview.appspot.com/280140043.

[70] Dmitry Vyukov. Recommended kernel configs, 2019. https://github.com/google/syzkaller/blob/master/docs/linux/kernel_configs.md.

[71] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu. Accurate and scalable cross-architecture cross-os binary code search with emulation. *IEEE Transactions on Software Engineering*, pages 1–1, 2018.

[72] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[73] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[74] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[75] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM.

[76] Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, SP '15, pages 797–812, Washington, DC, USA, 2015. IEEE Computer Society.

[77] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. ACM.

[78] M. Yan, Y. Fang, D. Lo, X. Xia, and X. Zhang. File-level defect prediction: Unsupervised vs. supervised models. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 344–353, Nov 2017.

[79] Yibiao Yang, Yuming Zhou, Jinping Liu, Yangyang Zhao, Hongmin Lu, Lei Xu, Baowen Xu, and Hareton Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 157–168, New York, NY, USA, 2016. ACM.

[80] Wei You, Peiyuan Zong, Kai Chen, XiaoFeng Wang, Xiaojing Liao, Pan Bian, and Bin Liang. Semfuzz: Semantics-based automatic generation of proof-of-concept exploits. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2139–2154. ACM, 2017.

[81] Christoph Zengler and Wolfgang Küchlin. Encoding the linux kernel configuration in propositional logic. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010) Workshop on Configuration*, volume 2010, pages 51–56, 2010.

[82] Feng Zhang, Quan Zheng, Ying Zou, and Ahmed E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 309–320, New York, NY, USA, 2016. ACM.

[83] Yuming Zhou, Yibiao Yang, Hongmin Lu, Lin Chen, Yanhui Li, Yangyang Zhao, Junyan Qian, and Baowen Xu. How far we have progressed in the journey? an examination of cross-project defect prediction. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(1):1, 2018.