Improving Vulnerability Inspection Efficiency Using Active Learning

Zhe Yu, Christopher Theisen, Laurie Williams, *Fellow, IEEE*, and Tim Menzies, *Fellow, IEEE*

Abstract—Software engineers can find vulnerabilities with less effort if they are directed towards code that might contain more vulnerabilities. HARMLESS is an incremental support vector machine tool that builds a vulnerability prediction model from the source code inspected to date, then suggests what source code files should be inspected next. In this way, HARMLESS can reduce the time and effort required to achieve some desired level of recall for finding vulnerabilities. The tool also provides feedback on when to stop (at that desired level of recall) while at the same time, correcting human errors by double-checking suspicious files.

This paper evaluates HARMLESS on Mozilla Firefox vulnerability data. HARMLESS found 80, 90, 95, 99% of the vulnerabilities by inspecting 10, 16, 20, 34% of the source code files. When targeting 90, 95, 99% recall, HARMLESS could stop after inspecting 23, 30, 47% of the source code files. Even when human reviewers fail to identify half of the vulnerabilities (50% false negative rate), HARMLESS could detect 96% of the missing vulnerabilities by double-checking half of the inspected files.

Our results serve to highlight the very steep cost of protecting software from vulnerabilities (in our case study that cost is, for example, the human effort of inspecting 28,750 $\times$ 20% = 5,750 source code files to identify 95% of the vulnerabilities). While this result could benefit the mission-critical projects where human resources are available for inspecting thousands of source code files, the research challenge for future work is how to further reduce that cost. The conclusion of this paper discusses various ways that goal might be achieved.

Index Terms—Active learning, security, vulnerabilities, software engineering, error correction.

1 Introduction

Software security is a current and urgent issue. A recent report [1] from the US National Institute of Standards and Technology (NIST) warns that "current systems perform increasingly vital tasks and are widely known to possess *vulnerabilities*". The vulnerabilities discussed in this paper are defined as follows:

- A mistake in software that can be directly used by a hacker to gain access to a system or network; or
- A mistake that lets attackers violate a security policy [2].

Government and scientific bodies stress the need for reducing software vulnerabilities. In a report to the White House Office of Science and Technology Policy, "Dramatically Reducing Software Vulnerabilities" [1], NIST encourages more research on approaches to reduce security vulnerabilities. The need to reduce vulnerabilities is also emphasized in the 2016 US Federal Cybersecurity Research and Development Strategic Plan [3].

To protect against software vulnerabilities, teams need to detect and fix the vulnerabilities before deployment. In this paper, we focus on the "detect" part of this process. Code inspection is a key process for vulnerability detection. Such inspections require software engineers to inspect large amounts of code (e.g. check that no call to the "C" `printf` function can be supplied a format string that is mismatched to the type of the items being printed). Resource limitations often preclude software engineers to inspect all source code files [4]. Making informed decisions on what source code files to inspect can improve a team's ability to find vulnerabilities. Vulnerability prediction models (VPMs) make such informed decisions by learning a machine learning model from known vulnerabilities. The model is then applied to classify source code files as "vulnerable" or "non-vulnerable". If software engineers only inspect the predicted "vulnerable" files, then human effort is reduced. That is, a good VPM helps human to find *more* vulnerabilities by inspecting *less* code.

Although the state-of-the-art VPMs have shown promising results on finding vulnerabilities with reduced human inspection effort [5], [6], [7], [8], [9], these approaches have limitations:

- VPMs learn models using training data which describes known vulnerabilities. Prior to the initial release, such data may not be available[1].
- Existing VPMs do not allow users to choose what level of recall (percentage of vulnerabilities found) to reach. Moreover, when software engineers finish inspecting the selected source code files, they do not know how many more vulnerabilities are as-yet-undetected in the remaining code.
- Humans may inspect a file but fail to find vulnerabilities [11]. Double checking (where files are inspected by other humans) is needed to cover such errors. Current VPM research does not discuss how to design cost-effective double-checking strategies.

The central insight of this paper is that the vulnerability prediction problem (finding more vulnerabilities by inspecting less code) belongs to a class of information retrieval problem—the *total recall* [12] problem (described in §2.2). An active learning-based framework has been shown effective to resolve all the above-mentioned limitation for total recall problems in another domain—

- *Z. Yu, L. Williams, and T. Menzies are with the Department of Computer Science, North Carolina State University, Raleigh, USA. E-mail: {zyu9, lawilli3}@ncsu.edu, timm@ieee.org.*
- *C. Theisen is with Microsoft, Seattle, USA. E-mail: theisen.cr@gmail.com*

1. Transferring training data may not solve this problem. Cross project vulnerability prediction has been shown to perform worse than within project vulnerability prediction [10].

selecting relevant papers for literature reviews [13]. This paper checks the conjecture that this active learning-based framework for selecting relevant papers can be applied and adapted to help find vulnerabilities efficiently while mitigating the limitations of current vulnerability prediction approaches. That said, in this paper, a novel direction is presented to assist software engineers in identifying vulnerabilities efficiently. A vulnerability inspection tool HARMLESS is built by applying and adapting the active learning-based framework for selecting relevant papers. Using HARMLESS, engineers inspect some source code files while HARMLESS trains/updates a VPM based on the inspection results. Then HARMLESS applies the VPM to suggest which files should be inspected next. Through iterating this process, HARMLESS guides the human inspection efforts towards source code files that are more likely to contain vulnerabilities. An interesting finding here is that the active learning-based framework can be applied to vulnerability prediction problem (as HARMLESS) with barely any modification. This means two things—firstly we can improve vulnerability inspection efficiency by recasting the problem in terms of total recall. Secondly, in the future, innovations in the solution of any total recall problem, including vulnerability prediction, can improve all other total recall problems including vulnerability prediction. This is an exciting line of research since this suggests a synthesis of many (seemingly different) lines of research.

This paper assesses the effectiveness of HARMLESS by simulating code inspections on C and C++ files from Mozilla Firefox project. The authors tagged the known vulnerabilities of the Mozilla Firefox project from Mozilla Foundation Security Advisories blog [14] and bug reports from Bugzilla[2] up to November 21st, 2017. Among the 28,750 unique source code files, 271 files contain vulnerabilities. These known vulnerable files are treated as ground truth. Note that our dataset does not provide information on whether an actual incident was caused by the identified ground truth vulnerabilities. During our file-level simulations, when a human is asked to inspect a source code file and tell whether it contains vulnerabilities or not, the ground truth is applied instead of a real code inspection. This enables our experiments to be repeated multiple times with different algorithm setups and provides reproducibility of this paper. For full details on that data, see §4.

Using this data, we explore four research questions to see whether HARMLESS can better resolve the aforementioned limitations of traditional VPMs:

RQ1: Can human inspection effort be saved by applying HARMLESS to find a certain percentage of vulnerabilities? Simulated on the Mozilla Firefox data without prior known vulnerabilities as training data, we show that 60, 70, 80, 90, 95, 99% of the known vulnerabilities can be found by inspecting around 6, 8, 10, 16, 20, 34% of the source code files, respectively. These results show that a good amount of human effort can be saved by applying HARMLESS. This result also highlights the very steep costs associated with protecting software from vulnerabilities. Even with our best methods, for the Mozilla Firefox dataset studied here, humans still need to manually inspect 28,750 × 20% = 5,750 source code files to identify 95% of the vulnerabilities. While this result could benefit the mission-critical projects where human resources are available for inspecting thousands of source code files (R-4a1 as done in [15]), the clear research challenge for future work is how to further reduce that cost. The conclusion of this paper offers some notes on how that goal might be achieved.

Note that these **RQ1** results assume that humans are infallible; i.e. they make no mistakes in their inspections (see **RQ3, RQ4** for what happens when humans become fallible).

RQ2: Can HARMLESS correctly stop the vulnerability inspection when a predetermined percentage of vulnerabilities has been found?

We show that HARMLESS can accurately estimate the number of remaining vulnerable files in a project. Based on the estimation, HARMLESS can tell humans when they should stop the inspection (i.e. when they have found the predetermined percentage of vulnerabilities). Our results also suggest that HARMLESS tends to slightly over-estimate the number of remaining vulnerable files, e.g. when targeting 95% recall, the inspection stopped at 97% recall with 30% cost, which means 97% of the vulnerable files can be identified by inspecting 8,625 source code files. Compared to the result of RQ1, about 3,000 more files need to be inspected due to the small estimation error of HARMLESS. Hence, it is very important to further improve the estimation since a tiny reduction in its error can lead to the saving of a large amount of effort.

RQ3: Is HARMLESS's performance affected by human errors? Hatton [11] warns that when inspecting codes for defects, human fails to detect 47% of the defects (47% false negative rate and 0% false positive rate) in average. We assume the same error rate for vulnerability inspections. By adding in 10 to 50% false negative rate to the simulated human oracles randomly, we simulate the influence of growing human error rate on HARMLESS's performance. Such human errors drastically and negatively impact the inspection result, thus motivating the next research question.

RQ4: Can HARMLESS correct human errors effectively? As seen in our simulations in §5.4, HARMLESS outperforms the state-of-the-art error correction mechanisms by only double-checking 50% of the inspected files but covering 96% of the missing vulnerabilities. According to our results, when targeting 95% recall with human having 50% chance of missing a vulnerability during the inspection, HARMLESS can reach $0.95 \times 0.85 = 81\%$ recall for $0.21 \times 1.86 = 39\%$ cost, which means 81% of the vulnerable files can be identified by inspecting 11,230 files (including double checks). If the reader finds this to be an excessive amount, then they might want to consider what has to happen without our tools:

- Without HARMLESS, one engineer would only find 50% of the vulnerabilities, and only after inspecting 28,750 files (100% of the files);
- Without HARMLESS, two engineers would find 75% of the vulnerabilities, but only after inspecting 57,500 files (200% of the files including double-checking effort).

1.1 Contributions of this Paper

1) Here in this paper we present a different direction to assist software engineers identifying potential vulnerabilities efficiently. Instead of training a model to predict vulnerabilities like traditional VPMs do, we apply an active learning-based framework (HARMLESS) to learn from human inspection results and to better select which source code files should be inspected next.
2) Simulations on Mozilla Firefox vulnerability dataset show that, without any prior knowledge, vulnerability inspections with HARMLESS can identify most vulnerabilities by inspecting only a small portion of the codebase.

2. https://bugzilla.mozilla.org/

3) An estimator that can be applied along with the execution of HARMLESS to estimate the remaining number of vulnerabilities in the codebase. This estimator can be used to provide guidance to stop the vulnerability inspection when a desired percentage of vulnerabilities have been found.
4) An error correction mechanism that detects vulnerabilities that humans failed to detect without imposing excessive extra human inspection effort.
5) All code and data used in this analysis are available[3], allowing other researchers to replicate, improve, or even refute our findings.

The rest of this paper is structured as follows. Some background and related work is discussed in §2. Our methodology is described in §3. This is followed by the details on how to simulate the HARMLESS inspection process on Mozilla Firefox vulnerability dataset in §4. Details of the experiment (simulation) designs and answers to the research questions are presented in §5. Threats to validity and limitations to this work are analyzed in §6 while conclusion and future work are provided in §7.

Before beginning, we digress to comment that in this case study we do *not* show that in practice, tools like HARMLESS are effective in reducing vulnerabilities in real projects. Before we can ask projects to work with us on such a study, we must first certify our technique on carefully controlled laboratory problems (hence this paper).

2 Background and Related Work

2.1 Security Vulnerability Prediction and Prioritization

In this subsection, we discuss the existing vulnerability prediction models (VPMs). These prediction models are used to prioritize validation and verification efforts. Based on what information is required and utilized in those approaches, we categorize them into three groups:

1) *Supervised methods*, which trains a model on historical inspection results (known vulnerable files and known non-vulnerable files) to predict the likelihood of new/changed files containing vulnerabilities;
2) *Semi-supervised methods*, which trains a model on historical inspection results as well as the new/changed (unlabeled) files to predict the likelihood of new/changed files containing vulnerabilities;
3) *Unsupervised methods*, which do not rely on historical inspection results, but utilizes other indicators for vulnerability proneness.

2.1.1 Supervised Methods

Most of the previous work in vulnerability prediction are supervised; i.e. they use known vulnerabilities to train a supervised learning model for predicting unknown vulnerabilities. Focusing on statistical classifiers on source code artifacts, such as binaries, files, and functions, these approaches classify code artifacts as either vulnerable or not vulnerable. Some of the datasets used in this process include the Windows operating system [7], [8], the Mozilla Firefox web browser [16], [9], and the RedHat Enterprise Linux Kernel [16]. The two types of features they use for predicting vulnerabilities are software metrics and text mining features:

3. https://github.com/ai-se/Mozilla_Firefox_Vulnerability_Data

Software metrics: Zimmermann *et al.* [7] reported a median recall of 20% after using all available source code metrics to build a Random Forest [17] classifier. Similarly, Chowdhury *et al.* [18] report that their vulnerability predictors had low recalls of 23%. Shin *et al.* [19], [16], [6] focused on churn and complexity with a different set of software metrics and achieved a higher recall of 83%. However, they also reported a precision as low as 5%. Hovsepyan *et al.* [20] extracted software metrics from design churn and achieved 71% precision, but only 27% recall on average.

Text mining features: Scandariato *et al.* [5] applied text mining to predict security vulnerabilities. Their hypothesis was that specific tokens found in source code files may indicate whether this file is more prone to vulnerabilities; for example, the presence of "nullptr" might mean a specific source code file is more prone to vulnerabilities resulting from null references. In their case studies [5], a static code analysis tool was used to decide the ground truth labels. Such static code analysis tools have a notoriously large false positive rate and may incorrectly decide that many code components are "vulnerable". Neuhaus *et al.* [21] extracted only the imports and calls information from source code as text mining features to predict vulnerabilities. They reported that their approach correctly predicted about half of all vulnerable components, and about two-thirds of all predictions were correct. Walden *et al.* [10] compared software metric-based approaches for vulnerability prediction to text mining approaches and found that text mining performed better in terms of precision and recall than software metric approaches. Later in §5, we show the same trend with HARMLESS that text mining features outperform software metrics. Walden *et al.* [10] also reported that cross-project prediction performances were generally poor—49%, 36% of the source code files need to be reviewed for finding 66%, 70% of the vulnerabilities, respectively. These results suggest that training data from the same project are required for these VPMs. As a result, these supervised VPM methods cannot be used before the first release since no training data from the same project is available.

2.1.2 Semi-supervised Methods

Semi-supervised VPMs not only learn from the known vulnerable and non-vulnerable files, but also extrapolate their conclusions to files that have not been inspected yet [22], [23], [24], [25]. For example, Meng *et al.* [26] (a) applied a label propagation algorithm [22] to assign labels to the unlabeled data; then (b) used those guesses to train a model for buffer overflow prediction. To the best of our knowledge, Meng's work is the only prior work on vulnerability prediction that used semi-supervised learning. Note that their work only predicts for one type of security vulnerabilities (buffer overflow) while in our work, we can target at any type of vulnerabilities.

2.1.3 Unsupervised Methods

Unsupervised approaches do not rely on human oracles (of which files contain vulnerabilities) to build the VPM. Instead, they are fully automatic methods that rely on other indicators of vulnerability proneness. For example, Gegick *et al.* [27] trained a decision tree classifier on non-security failures to rank software components. They found that, 57% of the vulnerable components were in the top 9% of the total component ranking, but with a 48% false positive rate. A better result was achieved by Theisen *et al.* [28], [8], [29] using crash features extracted from crash dump stack traces, which are collected after the software's first

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2949275, IEEE Transactions on Software Engineering

release. The crash features were used to approximate the attack surface and predict which parts of the software are more prone to be vulnerable. Theisen *et al.* found that crash history is a strong indicator of vulnerabilities—48.4% of the "crashed" binaries in Windows contain 94.6% of known vulnerabilities [8], and 15.8% of the "crashed" source code files in Mozilla Firefox contain 73.6% of known vulnerabilities [29]. The advantage of this approach is that it does not require the labeling of training data. However, one major drawback of Theisen *et al.* approach is that higher recall cannot be achieved since no information is provided in those source code files without crash history.

2.2 Active Learning

The methods listed above represent some of the best VPM results seen to date. While they are all significant research results, there is much remaining to be done. In this paper, we see if incrementally adding human insights (via active learning) to supervised/semi-supervised methods can mitigate any of the shortcomings discussed in §1.

The key idea behind active learning is that a machine learning algorithm can train faster (i.e. using less data) if it is allowed to choose the data from which it learns [30]. To understand active learning, consider the decision plane between the positive and negative examples in Figure 1. Suppose we want to find more positive examples and we have access to the Figure 1 model. One tactic for quickly finding those positive examples would be to inspect unlabeled data that fall into the red region of this figure, as far as possible from the green ones (this tactic is called *certainty sampling*). Another tactic would be to better improve the current model by verifying the position of the boundary; i.e. inspecting unlabeled data that are closest to the boundary (this tactic is called *uncertainty sampling*). The experience from explored total recall problems is that such *active learners* outperform supervised and semi-supervised learners and can significantly reduce the effort required to achieve high recall [31], [32], [33], [34], [35], [36], [37], [38], [39], [40], [41], [42], [13].

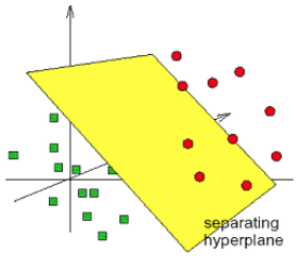


Fig. 1: Separating positive (red circles) from negative (green squares) examples.

For example, in electronic discovery, attorneys are hired to review a large number of documents looking for relevant ones to a certain legal case and provide those as evidence. Cormack and Grossman [34], [35], [36] designed and applied continues active learning to save attorneys' effort from reviewing non-relevant documents, which further can save a large amount of the cost of legal cases. Cormack and Grossman also proposed "knee methods"[4] as stopping rule [32] and as a way of efficiently correcting human errors [31].

Also, in evidence-based medicine, researchers screen titles and abstracts to determine whether one paper should be included in a certain systematic review. Wallace *et al.* [37] designed patient active learning to help researchers prioritize their effort on papers to be included. With patient active learning, half of the screening effort can be saved while still including most relevant papers [37]. Similarly, Miwa *et al.* [43] used active learning but with certainty sampling and weighting to achieve better efficiency. In addition, Wallace *et al.* [41] tried to estimate the number of relevant papers to indicate what recall has been reached during the process.

4. Where performance is monitored until the performance growth curve "turns a corner".

3 Methodology

The central insight of this paper is that the vulnerability prediction problem (finding more vulnerabilities by inspecting less code) along with the electronic discovery and evidence-based medicine problems all belong to a class of information retrieval problem called the *total recall* problem [12].

3.1 Total Recall

The target of *total recall* is to optimize the cost for achieving very high recall (ideally, very close to 100%) with a human assessor in the loop [12]. It differs from ad hoc information retrieval where the objective is to identify the best, rather than all relevant information, and from classification or categorization where the objective is to separate relevant from non-relevant information based on previously labeled training examples [44]. More specifically, the total recall problem can be described as follows:

> **The Total Recall Problem:**
> Given candidates E with a small positive fraction $R \subset E$, each $x \in E$ can be inspected to reveal its label as positive ($x \in R$) or negative ($x \notin R$) at a cost. Starting with the labeled set $L = \emptyset$, the task is to inspect and label as few candidates as possible ($\min |L|$) while achieving very high recall in finding positives ($\max |L \cap R|/|R|$).

The core problem of vulnerability prediction is how to find most of the vulnerabilities with least code inspected. Therefore the vulnerability prediction problem can be generalized as the total recall problem with:

- E: the entire codebase of a software project.
- R: set of source code files that contain at least one target type vulnerability.
- L: set of source code files already inspected by humans.
- $L_R = L \cap R$: set of source code files already inspected by humans and contain at least one target type vulnerability.

In addition, we believe it is even more appropriate to treat the vulnerability prediction problem as a total recall problem rather than a classification problem, given the severe consequences missing vulnerabilities might have. Therefore we conjecture that the active learning-based framework for total recall problems can better address the vulnerability prediction problem. The rest of this paper checks this conjecture.

3.2 Active learning-based Framework

In practice, active learning-based solutions of total recall problems focus on the following three targets [41], [32], [31], [13]:

- **Target 1 efficiency**: achieving higher recall with a lower cost than other solutions. This is the main target of total recall problems as defined in §3.1. This target measures the performance of a total recall solution with the true recall $|L_R|/|R|$ in an "ideal" scenario when every inspected examples are correctly labeled.
- **Target 2 stopping rule**: stopping at a predetermined recall rather than an arbitrary stopping point varying from different

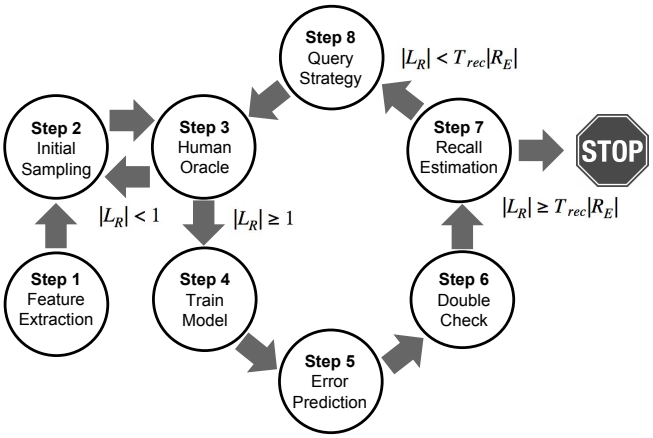


Fig. 2: Block diagram of the active learning-based framework for total recall problems

projects. Given that the total number of positives $|R|$ is unknown before all the candidates E are inspected, stopping at a predetermined recall is especially hard [41]. Stopping too late would waste inspection effort while stopping too early would miss potential positives (e.g. vulnerabilities).

- **Target 3 human error correction**: correcting human errors efficiently. Given the "human-in-the-loop" nature of the total recall problems, human errors affect the inspection results a lot. How to design strategies to efficiently fix such human errors is also an important problem.

In our previous work [42], we refactored techniques from Cormack, Wallace, and Miwa *et al.* [35], [37], [43] and created an active learning-based framework to solve one specific total recall problem (identifying relevant papers for literature reviews while minimizing the number of candidate literature being reviewed by humans). In that work, an SVM model incrementally learns which papers are more likely to be "relevant" based on previous decisions from humans and then guides the humans to read those papers next. When tested on four datasets, the proposed active learning framework outperforms the prior state-of-the-art approaches in terms of **Target 1 efficiency** [42]. Furthermore, SEMI, an estimator for the total number of relevant papers and DISAGREE, a technique to efficiently correct human classification errors were developed as part of the framework for **Target 2 stopping rule** and **Target 3 human error correction**, respectively [13]. When combined, all these techniques result in a general active learning-based framework shown in Figure 2 where **Step 4** and **Step 8** focus on **Target 1 efficiency**, **Step 7** decides **Target 2 stopping rule**, and **Step 5** and **Step 6** resolve **Target 3 human error correction**. The framework helps humans retrieve a target percentage (T_{rec}) of relevant information (R) with reduced human effort (L) as follows:

Step 1 **Feature Extraction:** Given a set of candidates E, extract features from each candidate. Initialize the set of labeled data points as $L \leftarrow \emptyset$ and the set of labeled positive data points as $L_R \leftarrow \emptyset$.

Step 2 **Initial Sampling:** Sample without replacement N_1 unlabeled data points to generate a queue $Q \subset (E \setminus L)$ [5].

Step 3 **Human Oracle:** Seek human oracles for data points in the queue $x \in Q$: label x as positive or negative after a human inspects the source code file; add file x into the labeled set L and remove x from Q; if x is labeled positive, also add x into set L_R. Once the queue is cleared $Q = \emptyset$, proceed to **Step 4** if at least one positive data point has been found ($|L_R| \geq 1$), otherwise go back to **Step 2**.

Step 4 **Train Model:** Train a classifier on the current labeled set L_R and L with features extracted in **Step 1** to predict whether one data point is positive or negative.

Step 5 **Error Prediction:** Apply the classifier trained in **Step 4** to predict on the labeled set L. Select N_2 data points with highest predicted probability to have been wrongly labeled for double-checking. *This step aims at correcting human errors more efficiently by suggesting which data points are more likely to have been wrongly labeled.*

Step 6 **Double check:** Each file in the queue from **Step 5** is assigned to different humans for double-checking.

Step 7 **Recall Estimation:** Estimate the total number of positive data (output $|R_E|$ as an estimation of $|R|$).

- **STOP** the process and output L_R if $|L_R| \geq T_{rec}|R_E|$ (where T_{rec} represents the user-defined target recall).
- Otherwise proceed to **Step 8**. *This step aims at providing an accurate estimation of the current recall, so that the user could choose to stop the process when target recall T_{rec} is reached.*

Step 8 **Query Strategy:** Apply the classifier trained in **Step 4** to predict on unlabeled set $(E \setminus L)$ and select N data points as the queue Q, then go to **Step 3** to acquire human oracles. *This step aims at achieving highest efficiency (higher recall and lower cost) by utilizing the classifier trained in **Step 4** to suggest which should be reviewed/inspected by human next.*

In the above, N_1, N_2 are engineering choices. N_1 is the batch size of the process, larger batch size usually leads to fewer times of training but also worse overall performance. $N_2 = \alpha N_1$ where $\alpha \in [0, 1]$ reflects what percentage of the labeled data are double-checked. Larger N_2 means more double checks, which leads to more human errors covered but also higher cost.

The active learning-based framework differs from the traditional semi-supervised/supervised learning methods by (1) starting training very early (with at least 1 positive), (2) always utilizing the trained model for sampling the data to inspect, and (3) refining the model with new inspection results. In this way, the active learning-based framework avoids wasting inspection efforts on random sampling and utilize the new inspection results to improve the model and make better predictions.

3.3 HARMLESS

This paper is based on the above general active learning framework, and extends it for vulnerability prediction. When applied to vulnerability prediction problem, HARMLESS follows the active learning-based framework shown in Figure 2 and the detailed techniques applied in each step will be presented in this subsection.

3.3.1 Feature Extraction

Features are not restricted in HARMLESS. Any types of features extracted from the source code can be used to predict vulnera-

5. Let A and B be two sets. The set difference, denoted $A \setminus B$ consists of all elements of A except those which are also elements of B.

bilities in HARMLESS. That said, users could extract their own features, but here we present three example types of features which are popular in existing vulnerability researches.

- Software metrics: followed by the work of Shin *et al.* [16], SciTools' Understand[6] is used to measure the static features of software.
- Crash features: followed by the work of Theisen *et al.* [8], the crash features measure the number of time each source code file has crashed. Such information is extracted from the crash dump stack trace data.
- Text mining features: different from the work of Scandariato *et al.* [5], where labels (vulnerable or non-vulnerable) are required to select tokens, we apply the same text mining feature extraction as the total recall approaches [42], [13]. Specifically, we:
  1) Tokenized all source code files without stop/control words removal or stemming.
  2) Selected tokens with largest tf-idf score across all source code files. For token t, its tf-idf score:
  $$Tfidf(t) = \sum_{d \in D} Tfidf(t, d),$$
  in which for token t in document d,
  $$Tfidf(t, d) = w_d^t \times (\log \frac{|D|}{\sum_{d \in D} sgn(w_d^t)} + 1),$$
  where w_i^t is the number of times token t appears in document d.
  3) Built a term frequency matrix with M tokens selected—according to our prior work in retrieving relevant papers [42], [13], we use $M = 4000$.
  4) Normalized each row (feature vector for each file) with their L2-norm[7].

3.3.2 Initial Sampling

Two different sampling strategies are applied in the initial sampling step of HARMLESS:

- **Random sampling**: random sample without replacement until the first "vulnerable" file is found.
- **Domain knowledge**: apply domain knowledge to guide the initial sampling so that the first "vulnerable" file can be found earlier. As Theisen *et al.* [8] suggested, files that have crashed before are more likely to contain security vulnerabilities. Therefore HARMLESS selects files by descending order of their crash feature counts when crash data is available.

3.3.3 Human Oracle

Human inspectors (software engineers) are employed to inspect the selected source code files. If vulnerabilities are found, the source code file x containing the vulnerabilities is labeled as positive and is added into L_R.

3.3.4 Train Model

The train model step in HARMLESS includes the following three procedures:

1) Generate **presumptive non-relevant examples** to alleviate the sampling bias in "non-vulnerable" examples ($L \setminus L_R$).
2) Apply **aggressive undersampling** to balance the training data.
3) Train a soft-margin linear **support vector machine** (SVM) on the current labeled files L_R and L, using features extracted in **Step 1** to predict whether one file is vulnerable or not.

Presumptive non-relevant examples is a technique created by Cormack and Grossman [34] to alleviate the sampling bias of negative (non-vulnerable) training data. With certainty sampling, only the data close to the positive labeled data in feature space are selected for inspection. As a result, the labeled negative data are mostly close to the positive data, not spreading all over the feature space. This sampling bias of negative training data will deteriorate the performance of the trained model [34]. To alleviate such sampling bias, each time before training, *presumptive non-relevant examples* samples randomly from the unlabeled data ($P \subset E \setminus L$) and presumes the sampled data to be negative (non-vulnerable) in training. The rationale behind this technique is that given the low prevalence of positive (vulnerable) data, it is likely that most of the presumed ones are negative (non-vulnerable).

Aggressive undersampling is a data balancing technique first created by Wallace *et al.* [37]. During the training process, aggressive undersampling throws away majority (non-vulnerable) training data ($P \cup L \setminus L_R$) closest to SVM decision plane until reaching the same number of minority (vulnerable) training data $|L_R|$. Aggressive undersampling is applied only when $|L_R| \geq N_3$ [42]. Here, N_3 is also an engineering choice to avoid aggressive undersampling reducing the number of training data too much. Later in §3.3.8 we will show that N_3 is also the threshold where the query strategy is switched from uncertainty sampling to certainty sampling. Therefore N_3 also features the trade-off between faster building a better model and greedily applying the model to save inspection effort.

Support vector machines (SVMs) are a well-known and widely-used classification technique. The idea behind SVMs is to map input data to a high-dimension feature space and then construct a linear decision plane in that feature space [45]. Soft-margin linear SVM [46] is selected as the classifier because it has been proved to be a useful model in SE text mining [47] and is applied in the state-of-the-art total recall methods [42], [43], [37], [35] as well as the state-of-the-art vulnerability prediction methods [21], [7], [48]. The soft-margin linear SVM is implemented with the SVC package from scikit-learn[8]. Default parameters are applied while kernel is set to be "linear" and penalty is set to be "balanced" (balancing the penalty of misclassifying a training data point by the population of its class, i.e. penalize more if misclassifying a minority class training data point) as suggested by Miwa *et al.* [43].

Quasi-Newton Semi-Supervised Support Vector Machine (QN S3VM) [49], [25] is a semi-supervised algorithm. It makes perfect sense to apply a semi-supervised learning in **Step 4** since there are a large number of unlabeled data available that are not utilized by supervised learners like linear SVM. S3VM is a semi-supervised version of the linear SVM, which makes it a fair comparison against the linear SVM. In addition, S3VM trains faster than other semi-supervised learners, e.g. label propagation as Meng *et al.* [26] suggested. Given that the inspector needs to wait for the model to be trained every N_1 files inspected, a faster learner like S3VM is preferred. In our experiments in §5.2, we test the S3VM learner implemented with the package[9] from Gieseke *et al.* [49], [25].

6. http://www.scitools.com
7. L2-norm for a vector x is $\sqrt{x^T x}$, where T denotes "transpose"
8. https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html
9. http://www.fabiangieseke.de/index.php/code/qns3vm

3.3.5 Error Prediction

In our prior work [13], an error correction mechanism was designed for correcting both false negatives and false positives. Its core assumption is:

> *Human classification errors are more likely to be found where human and machine classifiers disagree most.*

Since the error distribution of the vulnerability inspection process is different from that of selecting research papers (only false negatives are considered), it cannot be directly applied to vulnerability prediction. As a result, we adopt the same assumption and designed **DISPUTE**, which focuses on correcting false negatives only.

DISPUTE applies the model trained in **Step 4** to predict on the source code files that have only been inspected once and are labeled as "non-vulnerable" ($L \setminus L_R$). Select N_2 files with highest prediction probability of being vulnerable and add the selected files into a queue for double-checking.

3.3.6 Double check

Recall from the introduction that when humans inspect a file, they may fail to find vulnerabilities [11]. Some double checks (where files are examined by more than one human) are required to cover such errors. Later in this paper(§5.4), we will test two strategies for double-checking:

- **DISPUTE** double-checks each file selected in **Step 5** (of Figure 2) once.
- **DISPUTE(3)** double-checks each file selected in **Step 5** twice if the first double-checker still finds it to be "non-vulnerable".

DISPUTE(3) reduces the chance of the selected file being mislabeled again at the cost of doubling the double-checking effort. This strategy is especially useful when the human inspectors have high chance to miss vulnerable files.

3.3.7 Recall Estimation

In our prior work [13], a semi-supervised estimator called **SEMI** is designed for estimating current recall. SEMI utilizes a recursive *TemporaryLabel* technique. Each time the **SVM** model is retrained, SEMI assigns temporary labels to unlabeled examples and builds a logistic regression model on the temporary labeled data. It then uses the obtained regression model to predict the likelihood of being positive of the unlabeled data and updates the temporary labels. This process is iterated until convergence and the final temporary labels are used as an estimation of the total number of positive examples in the dataset. Detailed algorithm of SEMI is shown in Algorithm 1.

3.3.8 Query Strategy

Query strategy is a key part of active learning. The two query strategies applied in HARMLESS are: 1) **uncertainty sampling**, which picks the unlabeled examples that the active learner is most uncertain about (examples that are closest to the SVM decision hyperplane); and 2) **certainty sampling**, which picks the unlabeled examples that the active learner is most certain to be vulnerable (examples on the vulnerable side of and are furthest to the SVM decision hyperplane). Different researchers endorse different query strategies, e.g. Wallace *et al.* [37] applies uncertainty sampling in their patient active learning, Cormack *et al.* [35], [43] use certainty sampling from beginning to end. Followed by our previous design for literature review [13], HARMLESS applies uncertainty sampling early on and certainty sampling afterward.

Algorithm 1: Psuedo Code for SEMI [13]

Input : E, set of all candidates
L, set of labeled data
L_R, set of labeled positive data
CL, classifier trained in **Step 4**
Output : $|R_E|$, estimated number of positive data

Function *SEMI* (CL, E, L, L_R)
  $|R_E|_{last} \leftarrow 0$;
  $\neg L \leftarrow E \setminus L$;
  foreach $x \in E$ **do**
    $D(x) \leftarrow CL.decision\_function(x)$;
    if $x \in |L_R|$ **then**
      $Y(x) \leftarrow 1$;
    else
      $Y(x) \leftarrow 0$;
  $|R_E| \leftarrow \sum_{x \in E} Y(x)$;
  while $|R_E| \neq |R_E|_{last}$ **do**
    `// Fit and transform Logistic Regression`
    $LReg \leftarrow LogisticRegression(D, Y)$;
    $Y \leftarrow TemporaryLabel(LReg, \neg L, Y)$;
    $|R_E|_{last} \leftarrow |R_E|$;
    `// Estimation based on temporary labels`
    $|R_E| \leftarrow \sum_{x \in E} Y(x)$;
  return $|R_E|$;

Function *TemporaryLabel* $(LReg, \neg L, Y)$
  $count \leftarrow 0$;
  $target \leftarrow 1$;
  $can \leftarrow \emptyset$;
  `// Sort` $\neg L$ `by descending order of` $LReg(x)$
  $\neg L \leftarrow SortBy(\neg L, LReg)$;
  foreach $x \in \neg L$ **do**
    $count \leftarrow count + LReg(x)$;
    $can \leftarrow can \cup \{x\}$;
    if $count \geq target$ **then**
      $Y(can[0]) \leftarrow 1$;
      $target \leftarrow target + 1$;
      $can \leftarrow \emptyset$;
  return Y;

More specifically, **Step 8** (of Figure 2) applies the model trained in **Step 4** to predict on unlabeled files $(E \setminus L)$ and

- select top N_1 files with vulnerability prediction probability closest to 0.5 (**uncertainty sampling**) if $|L_R| < N_3$; or
- select top N_1 files with vulnerability prediction probability closest to 1.0 (**certainty sampling**) if $|L_R| \geq N_3$.

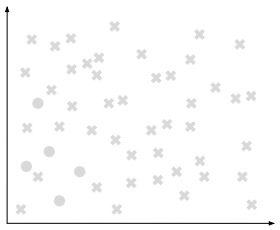
Push the selected files into the queue Q, then go to **Step 3** for another round of inspection.

3.3.9 Summary

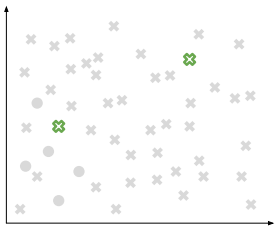
A simple demonstration of the HARMLESS inspection process is shown in Figure 3. In this figure, HARMLESS incrementally updates its model and uses it to guide the human inspector to only inspect the most informative files:

- The demo is progressive, i.e., from top left, we see the start of the process. In those initial plots, every item is a file. The subsequent plots show how HARMLESS guides the human inspector to reveal vulnerabilities.
- Any item with a solid color represents a file where humans have inspected.
- The hollow green items in Figure 3 are files that HARMLESS suggests the human to inspect. For example, these hollow green items appear in (b), (g). Subsequently, in (c), (h) we see the labels offered by humans (see the blue and red solid items).

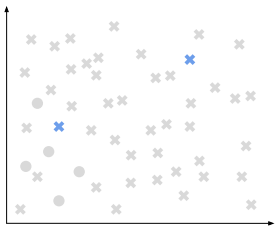
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2949275, IEEE Transactions on Software Engineering



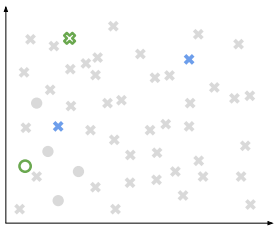
(a) **Step 1**: initialize each data points with 2 dimensionality features extracted from the source code of each file.



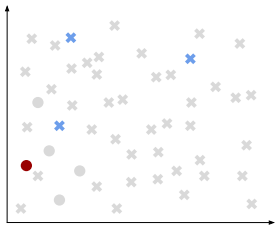
(b) **Step 2**: random sample $N_1 = 2$ data points (in hollow green) for human oracles.



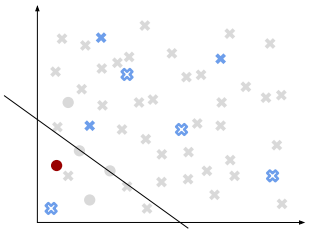
(c) **Step 3**: human inspects the selected $N_1 = 2$ points and labels them as negative (blue). Since $|L_R| < 1$, return to **Step 2**.



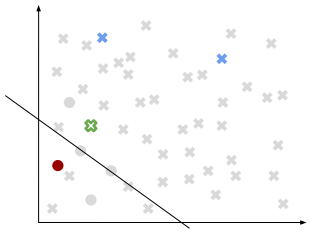
(d) **Step 2**: random sample $N_1 = 2$ data points (in hollow green) for human oracles.



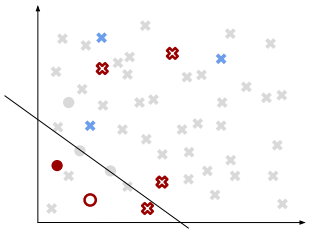
(e) **Step 3**: human inspects the selected $N_1 = 2$ points and labels them as positive (red) and negative (blue). Since $|L_R| \geq 1$, proceed to **Step 4**.



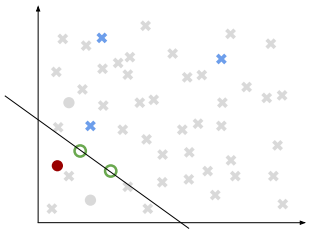
(f) **Step 4**: since $|L_R| < N_3 = 2$, train a linear SVM model with balanced weights and presumptive negative examples (hollow blue **X**s). The SVM decision plane is shown as the black line.



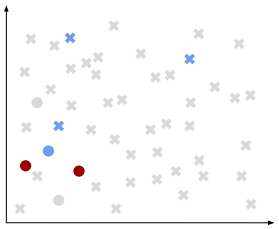
(g) **Step 5**: apply the SVM model for error prediction, select $N_2 = 1$ data point (in hollow green) with highest prediction probability to be positive for double-checking.



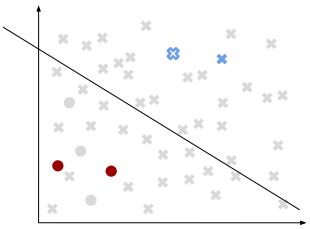
(h) **Step 6**: human double-checks the selected $N_2 = 1$ data point and labels it as negative (blue). **Step 7**: when converges, SEMI temporarily labels 5 data points as positive (hollow red), so $|R_E| = 5 + |L_R| = 6$. Since $|L_R| < T_{rec}|R_E| = 4.8$, proceed to **Step 8**.



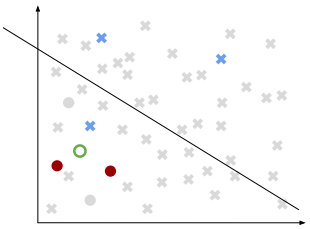
(i) **Step 8**: since $|L_R| < N_3 = 2$, apply uncertainty sampling to select $N_1 = 2$ points (in hollow green) closest to the SVM decision plane for human oracles.



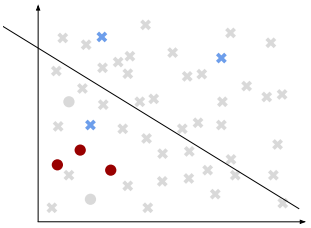
(j) **Step 3**: human inspects the selected $N_1 = 2$ points and labels them as positive (red) and negative (blue). This time, human wrongly labels one data point.



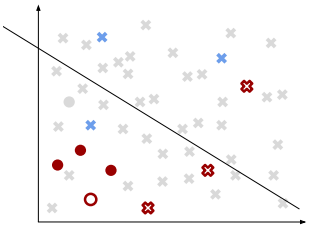
(k) **Step 4**: since $|L_R| \geq N_3 = 2$, train a linear SVM model with presumptive negative examples (hollow blue **X**s) and aggressive undersampling (keep $|L_R| = 2$ negatives).



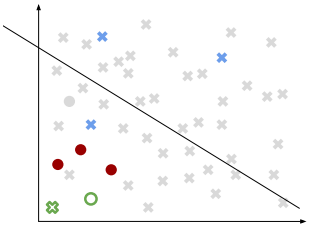
(l) **Step 5**: apply the SVM model for error prediction, select $N_2 = 1$ data point (in hollow green) with highest prediction probability to be positive for double-checking.



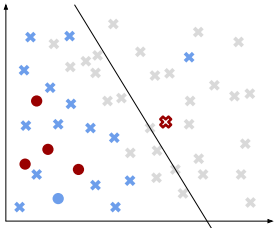
(m) **Step 6**: human double-checks the selected $N_2 = 1$ data point and label it as positive (red). This time, false negative is corrected.



(n) **Step 7**: when converges, SEMI temporarily labels 4 data points as positive (hollow red), so $|R_E| = 4 + |L_R| = 7$. Since $|L_R| < T_{rec}|R_E| = 5.6$, proceed to **Step 8**.



(o) **Step 8**: since $|L_R| \geq N_3 = 2$, apply certainty sampling to select $N_1 = 2$ points (in hollow green) with highest prediction probability to be positive for human oracles.



(p) Iterate for 7 more rounds, until **Step 7**: when converges, SEMI temporarily labels 1 data points as positive (hollow red), so $|R_E| = 1 + |L_R| = 5$. Since $|L_R| \geq T_{rec}|R_E| = 4$, stop and output the $|L_R| = 4$ points (in red).

Fig. 3: Consider $|E| = 50$ source code files in a software project. If human inspects all the files correctly, they will find $|R| = 5$ files (**O**s) being positive (vulnerable) and $|E \setminus R| = 45$ files (**X**s) being negative (vulnerability-free). In this demo, HARMLESS guides humans to inspect 20 (40%) source code files and find 4 (80%) of the vulnerable files. Data points colored in gray are unlabeled $E \setminus L$, while data points colored in red are labeled as positive L_R and data points colored in blue are labeled as negative $L \setminus L_R$. As for the hollow ones, green are the data points selected for inspections/double checks while R-4b red/blue are the data points temporarily labeled as positive/negative by SEMI/presumptive negative examples. Black lines are the decision hyper-planes of the linear SVM model.

- The solid diagonal lines show updates to the decision hyperplane of the SVM model. Note that this line first appears in (f) and is then updated in (k) and (p).

In the demo, for simplicity, parameters are set to $N_1 = 2, N_2 = 1, N_3 = 2$ and the feature dimensionality is set to be 2, i.e. each data point (source code file) is represented by 2 features. Target recall is set to be $T_{rec} = 0.8$ and positive data points can be sometimes mislabeled as negative to demonstrate human errors.

4 Mozilla Firefox Case Study

While §3 shows how HARMLESS should be applied in practice with humans inspecting codes, it is too expensive for human inspectors to test different treatments and answer all the research questions. As a result, the performance of HARMLESS is tested through simulations on the Mozilla Firefox vulnerability dataset.

4.1 Dependent Variables

The Mozilla Firefox dataset focuses on C and C++ files in Mozilla Firefox. Metrics were collected for 28,750 unique source code files in the project, within which 135 vulnerabilities affecting 271 files are manually labeled as the ground truth. These ground truth vulnerabilities were manually collected from Mozilla Foundation Security Advisories blog [14] and bug reports from Bugzilla[10] up to November 21st, 2017. As for Mozilla Foundation Security Advisories blog, vulnerability type and source code file(s) modified to fix the vulnerability were recorded. From this source, 247 files are found to contain vulnerabilities. Mozilla's bug database was then mined for bugs that were not reported publicly on the blog as vulnerabilities. Two human raters individually read through each bug report and classified each as "vulnerability" or "not a vulnerability". The two raters looked at the description of each bug, along with the comment history and the diffs describing the fix for the vulnerability to determine a classification. This resulted in 111 files involved with vulnerabilities. The inter-rater reliability between the two raters when classifying vulnerable files was $\kappa = 0.6$. These 111 files were then added to the 247 files found on the Mozilla Security Advisories blog, resulting in a final list of 358 source code files that were involved with vulnerabilities. Fourteen source code files were discarded and 8 source code files were changed based on the movement of code to another source code file or the removal of code from the system. Finally, a list of 271 unique files remained with at least one vulnerability.

After identifying the vulnerabilities for inclusion in the dataset, two humans then classify each vulnerability using an existing vulnerability classification scheme—the Common Weakness Enumeration (CWE) set of most commonly seen weaknesses in software[11]. After classifying each vulnerability, the humans then convene and resolve any differences that have occurred between the two of them. If they could not come to a consensus, a third party arbitrator was used to resolve the conflict.

After this classification, the Mozilla Firefox dataset contains 14 different types of vulnerabilities and some files are associated with multiple types of vulnerabilities. To simulate vulnerability inspections targeting specific types of vulnerabilities, we grouped the vulnerabilities into five categories to ensure that each category contains enough data (#Vulnerable Files) for a simulation, as shown in Table 1. In the following sections §5, we show results on simulations targeting each vulnerability category with file-level granularity. For example, when targeting "Protection Mechanism Failure", only the 119 files associated with this category are considered vulnerable.

4.2 Independent Variables

Here we briefly describe the three types of features extracted from the Mozilla Firefox vulnerability dataset and used in our simulations.

4.2.1 Software Metrics

SciTools' Understand[12] was used to measure the metrics from the Mozilla Firefox source code files. The list of the software metrics provided by the dataset is shown below:

- *CountClassBase* - Number of subclasses in a file.
- *CountClassCoupled* - Coupling of the classes in a file.
- *CountClassDerived* - Number of subclasses derived from classes originating in a file.
- *CountDeclInstanceVariablePrivate* - Number of private variables declared in a file.
- *CountDeclMethod* - Number of methods declared in a file.
- *CountInput* - Number of incoming calls to the source code file.
- *CountOutput* - Number of outgoing calls from the source code file.
- *Cyclomatic* - Cyclomatic Complexity of a file.
- *CountLine* - Number of lines of code (excluding comments and whitespace) in a file.
- *MaxInheritanceTree* - The size of the maximum leaf in the inheritance tree leading from a file.

Some metrics in Shin *et al.* [16] are not covered due to the following reasons: (1) metrics that no public tool (to our knowledge) provides an equivalent to, e.g. incoming closure and outgoing closure; (2) metrics that do not apply to the Mozilla Firefox project, e.g. organization intersection factor; (3) metrics that are identical to some already covered ones in Mozilla Firefox project, e.g. edit frequency.

4.2.2 Text Mining Features

As described in §3.3.1, text mining features were extracted by tokenizing the source code files, selecting top 4000 tokens based on tf-idf score, and then applying L2 normalization on files.

4.2.3 Crash Features

Crash dump stack trace data was collected from Mozilla Crash Reports[13]. We collected crashes from January 2017 to November 2017, with a total of 1,141,519 crashes collected. For each crash, the field marked "crashing thread" is observed and each file that appeared in the thread is added to the dataset. We also kept track of how many times a file was observed in different crashes since files that crash more often have a higher chance to contain vulnerabilities.

4.3 Simulation on Mozilla Firefox Dataset

Code inspection with HARMLESS, as described in §3, is simulated on the Mozilla Firefox dataset with the following settings:

Step 1 Feature Extraction: E is the set of all 28,750 source code files, R is the set of source code files containing the target

10. https://bugzilla.mozilla.org/
11. http://cwe.mitre.org/data/definitions/1003.html
12. http://www.scitools.com
13. https://crash-stats.mozilla.com/home/product/Firefox

TABLE 1: Descriptive statistics for vulnerabilities types grouping

Vulnerability Type	#Vulnerable Files	Containing Types
Protection Mechanism Failure	119	protection mechanism failure.
Resource Management Errors	85	uncontrolled resource consumption, improper resource shutdown or release, resource management errors, use after free, resource leak.
Data Processing Errors	35	data processing errors.
Code Quality	29	code quality.
Other	32	race conditions, configuration, environment, traversal, link-following, other.
All	271	All 14 types of vulnerabilities.

The Mozilla Firefox dataset contains 14 different types of vulnerabilities and some files are associated with multiple types of vulnerabilities. To simulate security reviews targeting specific types of vulnerabilities, we group the vulnerabilities into five categories to ensure that each category contains enough data (#Vulnerable Files) for a simulation. Column #Vulnerable Files reports the number of unique files containing the specific type of vulnerabilities. Files containing multiple vulnerabilities still count for 1 but can contribute to multiple groups.

TABLE 2: Experiments design

	Target research questions	Human error rate E_R in **Step 3**	Use of **Step 5** and **Step 6**?	Recall estimation $\lvert R_E\rvert$ in **Step 7**
Target 1 efficiency	**RQ1**	$E_R = 0$	no	$\lvert R_E\rvert = \lvert R\rvert$
Target 2 stopping rule	**RQ2**	$E_R = 0$	no	$\lvert R_E\rvert$ is estimated by SEMI as described in §3.3.7
Target 3 human error correction	**RQ3, RQ4**	$E_R \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5\}$	yes	$\lvert R_E\rvert$ is estimated by SEMI as described in §3.3.7

type of vulnerabilities shown in Table 1. $L \leftarrow \emptyset$ is the set of labeled files, and $L_R \leftarrow \emptyset$ is the set of labeled target vulnerable files. Extract **features** from each source code file in E.

Step 2 Initial Sampling: The batch size is chosen as $N_1 = 100$. We use $N_1 = 100$ during our simulations to speed up the simulation process. However, it is suggested to use a smaller batch size, e.g. $N_1 = 10$, in practice since the cost of training a model more often is usually much less than the cost of inspecting more code.

Step 3 Human Oracle: Rather than asking human inspectors to inspect source code files, as specified in §3.3.3, the ground truth labels from Mozilla Firefox dataset are applied to simulate the inspection. Each vulnerable file $x \in Q \cap R$ has E_R chance of being wrongly labeled as "non-vulnerable" and $1 - E_R$ chance of being correctly labeled as "vulnerable". Every non-vulnerable file $x \in Q \setminus R$ will be labeled as "non-vulnerable". Here $E_R = 0, 10, 20, 30, 40, 50\%$ simulates the probability of a human expert failing to detect the vulnerabilities (false negative rate).

Step 5 Error Prediction: Half of the inspected files are selected for double checks ($N_2 = 0.5N_1$) based on the DISPUTE principle. We show later in §5 that by double-checking half of the labeled files, most (96%) of the human false negatives can be covered.

Step 8 Query Strategy: Uncertainty sampling if $\lvert L_R\rvert < N_3 = 10$; certainty sampling when $\lvert L_R\rvert \geq N_3 = 10$.

Here, N_1, N_2, N_3 are selected based on our experience in solving other total recall problems [42], [13].

5 Experiments and Results

This section describes experiments assessing HARMLESS's performance. All the following experiments are simulated on the Mozilla Firefox case study, as described in §4 and are used to answer the research questions listed in §1.

5.1 Performance Metrics

HARMLESS's task is to optimize the inspection effort for achieving very high recall. Therefore HARMLESS focuses on minimal cost, maximal recall, defined as follows:

$$recall = \frac{\#\ of\ vulnerable\ files\ found}{\#\ of\ vulnerable\ files\ exist} = \frac{|L_R|}{|R|}. \quad (1)$$

$$cost = \frac{\#\ of\ source\ code\ files\ reviewed}{\#\ of\ source\ code\ files\ exist} = \frac{|L|}{|E|}. \quad (2)$$

Since our simulations are in file level granularity, the numerator of cost in (2) counts the number of times the source code files being reviewed, i.e. it still increases when the same file is reviewed for a second time by a different reviewer. Using these two metrics, one treatment is considered better than another if it reaches the same target recall with a lower cost.

Beside recall and cost, we use the Estimation vs Cost curve to assess the accuracy for estimating the total number of vulnerable files in **Step 7**. Figure 2:

$$estimation = \frac{\#\ of\ vulnerable\ files\ estimated}{\#\ of\ vulnerable\ files\ exist} = \frac{|R_E|}{|R|}. \quad (3)$$

The sooner this estimation converges to 1.0, the better the estimator is. As described in **Step 7**, the inspection process will stop when

$$|L_R| \geq T_{rec}|R_E|,$$

where T_{rec} represents the target recall. The closer the inspection stops to the target recall, the better the stopping rule.

As shown in Table 2, three experiments are designed to answer the research questions listed in §1 and evaluate HARMLESS based on the three targets described in §3.2. More details on each experiment's design and its corresponding result are presented in the rest of this section.

TABLE 3: Treatments for **Target 1 Efficiency**

Treatment	Learner	Learn From			
		software metrics	text mining features	crash features	human oracle
Metrics	SVM	✓			✓
Text	SVM		✓		✓
Hybrid	SVM		✓	✓	✓
S3VM	S3VM		✓		✓
Crash	n/a			✓	
Random	n/a				

5.2 Target 1 Efficiency

5.2.1 Simulation Design

Cost for reaching different levels of recall is used to assess the vulnerability inspection efficiency of HARMLESS. As a result, stopping rule and human errors are not considered, i.e. human error rate E_R in **Step 3** is set to be 0, **Step 5** and **Step 6** are disabled, and in **Step 7**, real recall is used for the stopping rule $|R_E| = |R|$, as shown in Table 2.

We first need to decide (1) what feature in **Step 1** serves best for predicting vulnerabilities in the active learning-based framework, and (2) what type of learner (supervised, unsupervised, semi-supervised) in **Step 4** performs best inside the active learning-based framework. As a result, we first test the following feature types with supervised learner (linear SVM as described in §3.3.1) to find the best feature set:

- **Metrics**: a linear SVM trained on file level software metrics features (described in §4.2.1) which quantifies different types of software complexity and are collected from the source code.
- **Text**: a linear SVM trained on file level text mining features (described in §4.2.2) which treats the source code as raw text and performs standard text mining feature extraction (term frequency with L2 normalization [42]) on it.
- **Hybrid**: a linear SVM trained on the combination of text mining features and crash features (described in §4.2.3). This feature set is built by adding one column (number of times the source code file has crashed) to the term frequency matrix of the text mining features set before normalization. In **Step 2**, crash counts are applied as domain knowledge to first sample files which have crashed most frequently.

Then we test the following semi-supervised and unsupervised learners within the active learning-based framework:

- **S3VM**: a linear kernel QN S3VM trained on file level text mining features, similar to **Text** except that it utilizes unlabeled data in training to build its model.
- **Crash**: an unsupervised approach where no model is trained and source code files are selected in descending order of the number of times they crashed [28] (described in §4.2.3). Note that this unsupervised approach does not require the active learning-based framework of HARMLESS.

As a baseline, the simulation also includes the following treatment:

- **Random**: a baseline approach where source code files are inspected in random order. This treatment serves as a baseline where code inspection is performed without any help from HARMLESS.

Summaries of the treatments are provided in Table 3. HARMLESS framework includes all the treatments learned from human oracles.

5.2.2 Experimental Result

RQ1: Can human inspection effort be saved by applying HARMLESS to find a certain percentage of vulnerabilities?

Table 4 shows the cost to reach different levels of recall. One method is considered better than another if it costs less to reach the same recall. Based on these results, we make the following observations:

- **Metrics**, **Text**, **Hybrid**, and **S3VM** all perform better than **Random**, indicating the effectiveness of the proposed active learning framework in HARMLESS.
- **Crash** performs well in the early stages (when recall<80%) but is unable to achieve certain target recall values (see the numerous "n/a" entries of Table 4). For vulnerabilities of type "Code Quality", **Crash** performs better than any other approach. Overall, this unsupervised approach has limitations compared to active learning-based approaches.
- Targeting different types of vulnerabilities does not affect much of the performance of HARMLESS with active learning.

Therefore active learning-based HARMLESS approach is recommended to reliably achieve high recall with low cost. Among the active learning-based approaches:

- Software metrics features (**Metrics**) perform the worst—they always cost more to reach the same recall. Therefore we do not recommend using static software metrics as features to predict vulnerabilities in HARMLESS.
- When crash features are unavailable, e.g. before a software's first release, **Text** performs the best.
- When crash features are available, **Hybrid** utilizes both **Text** and **Crash** and performs slightly better than **Text** in terms of median performances and greatly reduces the variance.
- A semi-supervised learner with text mining features (**S3VM**) performs worse than its supervised counterpart (**Text**), especially in the early stages when training data is few. Therefore semi-supervised learners are not recommended inside the active learning-based framework.
- These results also show that the human effort required is still high even with the help of HARMLESS, e.g. to identify 271 × 0.95 = 257 vulnerable files, 28,750 × 0.2 = 5,750 files need to be inspected by humans. While this means 28,750 - 5,750 = 23,000 fewer files to be inspected compared to inspecting all the files, it is still a tedious and frustrating task (one vulnerable file may be found within every 22 files inspected).

Summing up this work on **RQ1**, we say:

> **RQ1: Can human inspection effort be saved by applying HARMLESS to find a certain percentage of vulnerabilities?**
>
> Yes. Simulated on Mozilla Firefox dataset, applying HARMLESS with linear SVM trained on text mining features, 60, 70, 80, 90, 95, 99% of the target vulnerabilities can be found by inspecting only 6, 8, 10, 16, 20, 34% of the source code files, respectively. If available, crash features can further boost the inspection efficiency when applied to guide the inspection order in the early stage.

5.3 Target 2 Stopping Rule

5.3.1 Simulation Design

In this experiment, the SEMI estimator (described in §3.3.7), is applied to estimate the number of vulnerabilities when running HARMLESS with the best feature set picked from **RQ1**. The estimation from SEMI is then used as an indicator of whether the target recall has been reached and thus the inspection can be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2949275, IEEE Transactions on Software Engineering

TABLE 4: Experimental Results for Target 1 Efficiency

		Target recall							
		60	70	80	85	90	95	99	100
	Vulnerability Type	Cost to reach target recall							
Metrics	Protection Mechanism Failure	50 (7)	56 (6)	60 (8)	63 (8)	67 (8)	79 (7)	94 (3)	99 (2)
	Resource Management Errors	49 (25)	57 (15)	63 (13)	67 (12)	71 (10)	75 (9)	99 (0)	99 (0)
	Data Processing Errors	35 (4)	41 (8)	45 (9)	61 (23)	70 (24)	78 (9)	82 (5)	82 (5)
	Code Quality	38 (6)	42 (20)	61 (19)	61 (18)	64 (12)	66 (9)	72 (3)	72 (3)
	Other	38 (19)	45 (17)	58 (7)	60 (6)	64 (8)	70 (10)	92 (12)	92 (12)
	All	49 (17)	53 (7)	60 (4)	63 (4)	66 (3)	72 (3)	94 (6)	99 (0)
	Median	42 (18)	50 (17)	59 (12)	62 (10)	67 (8)	74 (10)	91 (15)	94 (19)
Text	Protection Mechanism Failure	6 (1)	9 (2)	12 (1)	16 (1)	20 (3)	28 (3)	37 (9)	43 (7)
	Resource Management Errors	6 (1)	8 (2)	9 (1)	9 (2)	11 (3)	13 (2)	33 (4)	33 (4)
	Data Processing Errors	12 (4)	13 (2)	14 (4)	15 (4)	16 (4)	20 (4)	42 (15)	42 (15)
	Code Quality	4 (4)	5 (4)	8 (3)	10 (4)	19 (5)	21 (5)	30 (7)	30 (7)
	Other	5 (3)	6 (4)	11 (5)	12 (5)	14 (4)	18 (3)	33 (2)	83 (2)
	All	5 (0)	7 (0)	9 (1)	12 (1)	14 (1)	20 (2)	34 (2)	85 (0)
	Median	6 (2)	8 (3)	10 (4)	13 (4)	16 (5)	20 (7)	34 (7)	43 (49)
Hybrid	Protection Mechanism Failure	7 (0)	9 (0)	12 (0)	14 (0)	16 (1)	21 (3)	58 (1)	59 (1)
	Resource Management Errors	5 (0)	7 (0)	8 (0)	9 (0)	11 (0)	13 (2)	39 (0)	39 (0)
	Data Processing Errors	7 (0)	10 (0)	15 (0)	15 (0)	16 (0)	29 (2)	37 (3)	37 (3)
	Code Quality	3 (1)	4 (1)	4 (2)	5 (1)	10 (0)	12 (0)	16 (1)	16 (1)
	Other	4 (0)	9 (0)	11 (0)	12 (0)	14 (0)	14 (0)	37 (9)	61 (14)
	All	6 (0)	8 (0)	10 (0)	11 (0)	14 (0)	18 (2)	36 (0)	59 (0)
	Median	6 (2)	8 (1)	10 (4)	12 (5)	14 (4)	17 (7)	37 (5)	41 (22)
S3VM	Protection Mechanism Failure	20 (44)	22 (43)	25 (45)	28 (41)	33 (37)	52 (23)	62 (16)	70 (15)
	Resource Management Errors	21 (22)	23 (21)	24 (22)	25 (21)	30 (20)	32 (18)	51 (8)	51 (8)
	Data Processing Error	25 (37)	26 (38)	28 (40)	29 (39)	32 (36)	37 (29)	43 (28)	43 (28)
	Code Quality	29 (20)	29 (19)	30 (19)	30 (19)	35 (26)	41 (23)	53 (21)	53 (21)
	Other	28 (20)	29 (21)	30 (19)	32 (17)	33 (17)	36 (17)	53 (40)	53 (40)
	All	20 (8)	21 (8)	25 (7)	27 (7)	31 (5)	41 (5)	64 (10)	80 (7)
	Median	23 (24)	25 (23)	27 (22)	29 (20)	32 (22)	40 (23)	56 (27)	60 (35)
Crash	Protection Mechanism Failure	5 (0)	8 (0)	13 (0)	n/a	n/a	n/a	n/a	n/a
	Resource Management Errors	4 (0)	5 (0)	8 (0)	10 (0)	11 (0)	n/a	n/a	n/a
	Data Processing Errors	8 (0)	11 (0)	n/a	n/a	n/a	n/a	n/a	n/a
	Code Quality	1 (0)	2 (0)	4 (0)	4 (0)	10 (0)	11 (0)	12 (0)	12 (0)
	Other	6 (0)	9 (0)	12 (0)	n/a	n/a	n/a	n/a	n/a
	All	5 (0)	8 (0)	11 (0)	14 (0)	n/a	n/a	n/a	n/a
	Median	5 (1)	8 (3)	11 (4)	n/a	n/a	n/a	n/a	n/a
Random	Protection Mechanism Failure	59 (7)	70 (4)	80 (4)	85 (3)	90 (2)	95 (2)	98 (1)	99 (0)
	Resource Management Errors	59 (4)	69 (5)	77 (5)	84 (4)	88 (3)	94 (2)	99 (1)	99 (1)
	Data Processing Errors	57 (12)	70 (11)	78 (10)	84 (9)	86 (10)	94 (5)	97 (3)	97 (3)
	Code Quality	58 (8)	70 (10)	77 (11)	82 (8)	90 (9)	94 (6)	98 (2)	98 (2)
	Other	59 (5)	67 (5)	78 (6)	83 (5)	88 (6)	93 (5)	97 (2)	97 (2)
	All	59 (3)	69 (3)	79 (3)	84 (2)	89 (2)	94 (1)	98 (0)	99 (0)
	Median	59 (6)	69 (6)	79 (5)	83 (5)	89 (4)	94 (3)	98 (1)	98 (2)

This table shows the cost (i.e. percent code reviewed) required to reach different levels of recall. Medians and IQRs (75th-25th percentile) are shown for 30 repeated simulations (IQR results are shown in brackets). **Crash** has many empty cells since it provides no information on vulnerabilities located in the files that have never crashed. In this table, *lower* median values are *better* so all the **Random** results are worse than anything else (as might be expected). Of the remaining results, **Text** and **Hybrid** perform better than **Metrics**. Important note: this table does not consider how to stop at the target recall, it only shows the cost when first reaching that recall. For experiments considering stopping rules, see Table 5.

safely stopped. This experiment is designed to test the accuracy of the SEMI estimations and whether the inspections can stop close to the target recall. **Uniform random sampling** is applied as a baseline estimation. **Uniform random sampling** samples from unlabeled data randomly and queries the labels, then estimates $|R|$ as $|R_E| = |E| \times |L_R|/|L|$. It was believed to be the most accurate estimation method before the introduction of SEMI [41].

Human errors are considered in the next experiment so, for this experiment, the human error rate E_R in **Step 3** is set to be 0, and **Step 5** and **Step 6** are disabled, as shown in Table 2.

5.3.2 Experimental Result

RQ2: Can HARMLESS correctly stop the vulnerability inspection when a predetermined percentage of vulnerabilities has been found? To answer this research question, we first show the accuracy of the estimations from SEMI by presenting the Estimation vs Cost curve in Figure 4. In these plots, estimations, calculated as Equation (3), are considered accurate and useful if they converge to estimation=1.0 (denoted as the "true" line) early (when cost is small). According to Figure 4, the estimations from SEMI using either **Hybrid** or **Text** feature converge to 1.0 earlier than **Uniform random sampling** on every target vulnerability types. Usually the estimation error of SEMI becomes $\leq 5\%$ after cost ≥ 0.3.

Table 5 shows results using the SEMI stopping rule. In most cases, the SEMI estimator slightly over-estimates ($|R_E| > |R|$) to ensure the target recall is reached. Also, the higher the target recall, the more effective the stopping rule (i.e. estimation becomes more accurate when cost increases). Further, the stopping rule works better with **Text**, where the inspection usually stops with less than 6% error from target recall. Finally, **Hybrid** over-estimates ($|R_E| > |R|$) too much on "Code Quality" and under-estimates ($|R_E| < |R|$) on "All". Overall, when using SEMI, **Text** is better.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2949275, IEEE Transactions on Software Engineering

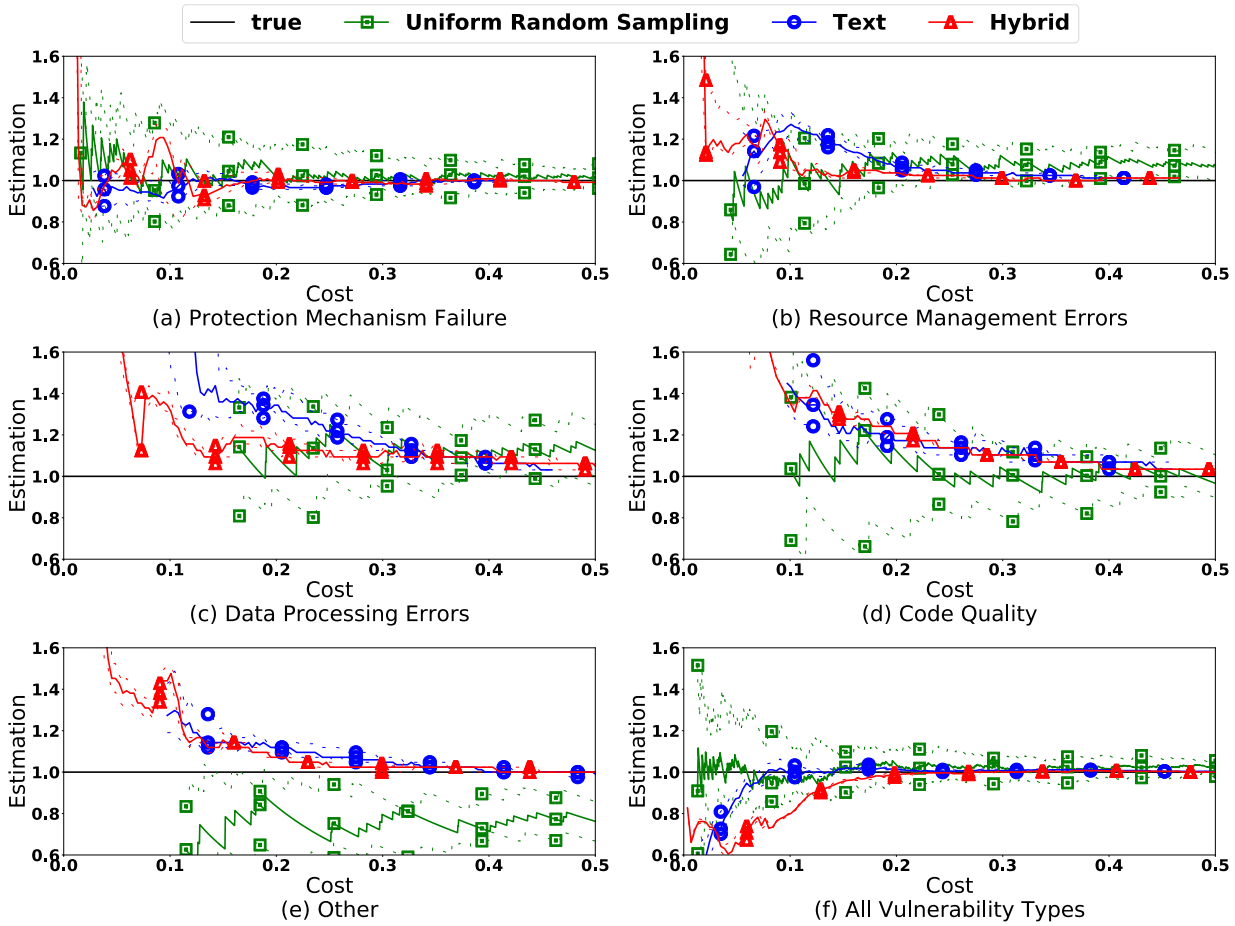


Fig. 4: Estimation vs Cost curves. Estimations (calculated as Equation (3)) are considered accurate and useful if they converge to estimation=1.0 early (when cost is small). **Solid lines** represent the median estimations from 30 simulations while **dashed lines** show the 75th to 25th percentile range. In this figure, estimation=1.0 is denoted as the "true" line. When an estimation converges to this line, it means that the estimated number of vulnerabilities are equal to the true number. Also, when cost reaches 1.0, all source code files have been reviewed. For example, the estimations from **Text** and **Hybrid** on the entire project converge to 1.0 when cost is about 0.25, thus providing accurate estimation on whether the target recall has reached for cost ≥ 0.25.

TABLE 5: Experiment Results for Target 2 Stopping Rule

		target recall					
		90		95		99	
	Vulnerability Type	Recall	Cost	Recall	Cost	Recall	Cost
Text	Protection Mechanism Failure	87 (2)	18 (2)	91 (2)	22 (3)	98 (3)	33 (5)
	Resource Management Errors	96 (0)	19 (2)	98 (1)	26 (3)	100 (0)	46 (3)
	Data Processing Errors	96 (2)	36 (5)	100 (3)	46 (5)	100 (0)	56 (6)
	Code Quality	96 (3)	30 (7)	100 (0)	42 (6)	100 (0)	54 (7)
	Other	97 (0)	25 (6)	97 (0)	32 (6)	97 (0)	50 (5)
	All	90 (2)	15 (2)	95 (1)	21 (1)	99 (0)	43 (0)
	Median	96 (6)	23 (12)	97 (4)	30 (17)	99 (2)	47 (10)
Hybrid	Protection Mechanism Failure	84 (19)	14 (7)	94 (2)	20 (6)	98 (1)	44 (12)
	Resource Management Errors	93 (1)	12 (0)	98 (1)	17 (1)	100 (0)	48 (0)
	Data Processing Errors	96 (3)	31 (3)	100 (0)	45 (2)	100 (0)	58 (2)
	Code Quality	100 (0)	27 (0)	100 (0)	38 (0)	100 (0)	58 (1)
	Other	95 (0)	20 (0)	95 (0)	28 (1)	97 (0)	52 (3)
	All	67 (2)	8 (0)	70 (2)	8 (0)	99 (0)	46 (0)
	Median	93 (11)	20 (14)	96 (5)	27 (21)	100 (1)	50 (10)

This table shows the actual achieved recalls and cost when using the SEMI estimator to stop at target recall. Same as in Table 4, numbers in brackets denote IQR values (75th-25th percentile ranges) and the other numbers are median values across 30 repeated simulations.

RQ2: Can HARMLESS correctly stop the vulnerability inspection when a predetermined percentage of vulnerabilities has been found?

Yes. Through accurate estimation of the number of vulnerabilities, HARMLESS can stop the inspection process close to the target recall. We recommend using text mining features and setting the target recall as high as 95% or 99% to guarantee the detection of most vulnerabilities.

The above results also suggest that it is very important to further improve the estimation since a tiny reduction in its error can lead to the saving of a large amount of effort. For example, when targeting 95% recall, the inspection stopped at 97% recall with 30% cost, which means 97% of the vulnerable files can be identified by inspecting 8,625 source code files. Compared to the result of RQ1, about 3,000 more files need to be inspected due to the small estimation error (2%) of HARMLESS.

5.4 Target 3 Human Error Correction

5.4.1 Simulation Design

Hatton [11]'s report suggested that each "vulnerable" file has in average 47% chance of being mislabeled as "non-vulnerable" while "non-vulnerable" files can never be mislabeled as "vulnerable". Following this report, we simulate how vulnerability inspection would be affected by randomly injecting human errors:

- No false positives: "non-vulnerable" files can never be mislabeled as "vulnerable".
- False negative rate E_R: each "vulnerable" file has the same chance of being mislabeled as "non-vulnerable".

Our simulations explore five ways to handle human errors while injecting false negatives at the rate of $E_R = 0, 10, 20, 30, 40, 50\%$):

- **None**: No error correction. This will be our baseline result.
- **Two-persons** [11] check each label. In **Step 3**, if one file is labeled as "non-vulnerable" and has been reviewed by only one reviewer, it goes back into the queue to wait for a different reviewer to double-check it.
- **Cormack'17** [31]: This is an advanced error correction method for citation screening. We reproduce half of Cormack'17 to correct false negatives only and incorporate it into the distributed framework: In **Step 8**, Cormack'17 utilizes a different stopping rule. This method detects the inflection point[14] i of the current recall curve, and compare the slopes before and after i. If $slope_{<i}/slope_{>i}$ is greater than a specific threshold $\rho = 6$, the review should be stopped. For details about this stopping rule, please refer to Cormack and Grossman [32]. After the stopping rule is satisfied in **Step 8**, send all files reviewed before the inflection point i and labeled as "non-vulnerable" to the queue. The security review and test stops after all the files in the queue have been reviewed by a different reviewer.
- **DISPUTE**: As described in §3.3.5, every iteration, $M = 0.5N$ of the labeled "non-vulnerable" files are selected based on how much the active learner disagrees with their current labels. It then pushes the selected files back into the queue and asks a different human expert for double-checking. In simulations, these double checks have the same false negative rate E_R.
- **DISPUTE(3)**: Similar to DISPUTE but selected files are inspected by *two* humans (so "vulnerable" files are less likely to be missed again albeit doubling the double-checking cost).

5.4.2 Experimental Result

RQ3: Is HARMLESS's performance affected by human errors? To show how human errors affect the performance, we take the results from HARMLESS with **Text**, target recall = 95% (Table 5) as baseline result and test the same algorithm with human error rate increasing from 0% to 50%. Column None in Table 6 shows the relative performances against the baseline result if no error correction is applied. As we can see, with 50% human error rate, the final recall becomes half of that of baseline recall.

> **RQ3: Is HARMLESS's performance affected by human errors?**
> Yes. Human errors during the inspection process can adversely affect the performance of HARMLESS. For example, human agents with 50% recall result in a deterioration of recall from 96% to 48%.

14. The inflection (or "elbow") point has the longest distance to the line between the recall starting point and endpoint [32].

The large deterioration in recall motivates the next question.

RQ4: Can HARMLESS correct human errors effectively? Table 6 compares DISPUTE and DISPUTE(3), the error correction methods from HARMLESS, against two state-of-the-art error correction methods, Two-person and Cormack'17. One method is considered better than another if it achieves higher recall with a lower cost. From the last line of Table 6, we observe that all error correction methods achieve much higher recall than **None** (i.e. no error correction). That is, error correction is strongly recommended when human inspectors are fallible. Also, in Table 6:

- **Cormack'17** costs least among all the error correction methods, but also achieves lower recall. Given recall as highest priority, **Cormack'17** is not recommended.
- **DISPUTE** is better than Two-person; i.e. it achieves similar recalls at less cost. For example, when $E_R = 50\%$, DISPUTE reaches $0.74 \times 0.96 = 71\%$ recall with $1.51 \times 0.26 = 39\%$ cost while Two-person reaches $0.75 \times 0.96 = 72\%$ recall with $2.06 \times 0.26 = 54\%$ cost.
- **DISPUTE(3)** is also better than Two-person; i.e. it achieves higher recall with similar cost (achieves $0.86 \times 0.96 = 83\%$ recall with $1.96 \times 0.26 = 51\%$ cost when $E_R = 50\%$).
- Compared to **DISPUTE**, **DISPUTE(3)** costs more but also corrects more errors. Therefore one can always increase the number of humans rechecking the files selected by **DISPUTE** to reach higher recall with higher cost.
- The coverage of false negatives by **DISPUTE** is $\frac{Relative\ Recall-(1-E_R)}{(1-E_R)E_R} = \frac{0.74-0.5}{0.25} = 96\%$ when $E_R = 50\%$. This supports the core hypothesis of **DISPUTE**; i.e. most human errors (96%) are in files selected by **DISPUTE** (50% of the labeled files). Hence, we say:

> **RQ4: Can HARMLESS correct human errors effectively?**
> Yes. HARMLESS corrects more human errors with fewer double checks when compared to other state-of-the-art error correction methods. It double-checks 50% of the labeled files but covers 96% of the missing vulnerabilities.

According to our results, when targeting all types of vulnerabilities at 95% recall with human having 50% chance of failing to detect a vulnerability during the inspection, HARMLESS with DISPUTE(3) can reach $0.95 \times 0.85 = 81\%$ recall for $0.21 \times 1.86 = 39\%$ cost, which means 81% of the vulnerable files can be identified by inspecting 11,230 files. Although this is still a large amount of human effort, **R-4a2** it is much better than the traditional vulnerability inspection applied in mission-critical projects without HARMLESS [15]:

- Without HARMLESS, one engineer would only find 50% of the vulnerabilities, and only after inspecting 28,750 files (100% of the files);
- Without HARMLESS, two engineers would find 75% of the vulnerabilities, but only after inspecting 57,500 files (200% of the files including double-checking effort).

6 Discussion

6.1 Limitations

There exist several limitations in this work that we plan to resolve in the future:

- All the experiments in this work are conducted at file level granularity. Currently, $Cost$ is measured as the number of files

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2019.2949275, IEEE Transactions on Software Engineering

TABLE 6: Experimental Results for Target 3 Human Error Correction

Error Rate (E_R)	Vulnerability Type	Relative Recall = Observed Recall / Baseline Recall (from Table 5)					Relative Cost = Observed Cost / Baseline Cost (from Table 5)				
		None	Two-person	Cormack'17	DISPUTE	DISPUTE(3)	None	Two-person	Cormack'17	DISPUTE	DISPUTE(3)
0%	Protection Mechanism Failure	100 (1)	100 (1)	91 (8)	98 (2)	98 (2)	100 (11)	200 (22)	60 (22)	128 (27)	171 (36)
	Resource Management Errors	100 (0)	100 (0)	96 (0)	100 (0)	100 (0)	100 (5)	200 (10)	56 (13)	149 (7)	198 (9)
	Data Processing Errors	100 (0)	100 (0)	96 (0)	100 (0)	100 (0)	99 (12)	199 (24)	69 (21)	137 (13)	181 (17)
	Code Quality	100 (0)	100 (0)	89 (6)	100 (0)	100 (0)	100 (6)	200 (12)	43 (13)	145 (8)	192 (12)
	Other	100 (0)	100 (0)	78 (17)	100 (0)	100 (0)	100 (10)	200 (20)	43 (34)	144 (11)	190 (10)
	All	100 (0)	100 (0)	95 (0)	100 (0)	100 (0)	100 (0)	200 (0)	47 (6)	150 (1)	201 (2)
	Median	100 (0)	100 (0)	95 (7)	100 (0)	100 (0)	100 (6)	200 (13)	52 (22)	146 (13)	192 (19)
10%	Protection Mechanism Failure	90 (3)	99 (1)	87 (12)	96 (2)	97 (2)	97 (8)	199 (26)	58 (31)	116 (14)	148 (35)
	Resource Management Errors	91 (4)	98 (1)	95 (1)	99 (1)	99 (1)	102 (11)	199 (15)	57 (16)	147 (14)	197 (19)
	Data Processing Errors	90 (3)	100 (3)	90 (14)	98 (3)	100 (2)	105 (11)	201 (24)	61 (30)	140 (15)	180 (28)
	Code Quality	91 (6)	98 (3)	86 (5)	100 (2)	100 (0)	101 (8)	200 (17)	46 (28)	151 (10)	195 (7)
	Other	92 (4)	100 (0)	92 (12)	100 (1)	100 (0)	101 (8)	200 (16)	59 (44)	145 (10)	191 (8)
	All	90 (2)	99 (0)	92 (1)	98 (0)	100 (0)	99 (1)	200 (2)	49 (4)	149 (2)	200 (1)
	Median	90 (4)	99 (2)	91 (9)	98 (2)	100 (1)	100 (9)	200 (15)	54 (23)	146 (14)	193 (21)
20%	Protection Mechanism Failure	80 (1)	95 (2)	82 (9)	96 (4)	97 (3)	103 (11)	196 (26)	63 (26)	130 (23)	155 (37)
	Resource Management Errors	79 (2)	96 (2)	91 (5)	95 (2)	97 (2)	105 (11)	204 (22)	60 (4)	147 (15)	196 (19)
	Data Processing Errors	78 (9)	93 (5)	87 (28)	93 (3)	98 (3)	102 (9)	204 (28)	64 (38)	140 (15)	180 (20)
	Code Quality	86 (12)	96 (3)	79 (17)	96 (6)	98 (3)	105 (5)	210 (19)	39 (15)	154 (17)	197 (5)
	Other	82 (7)	97 (4)	86 (20)	96 (6)	100 (0)	98 (11)	202 (26)	49 (41)	147 (14)	191 (13)
	All	79 (2)	96 (1)	87 (2)	95 (1)	99 (0)	97 (2)	199 (3)	49 (7)	148 (4)	200 (3)
	Median	80 (6)	96 (3)	86 (11)	95 (3)	98 (3)	101 (11)	201 (22)	52 (28)	145 (15)	194 (19)
30%	Protection Mechanism Failure	68 (2)	91 (4)	65 (17)	88 (3)	93 (3)	101 (4)	201 (26)	40 (43)	116 (18)	160 (38)
	Resource Management Errors	74 (8)	90 (4)	84 (5)	91 (5)	96 (2)	104 (9)	199 (14)	69 (20)	147 (14)	209 (22)
	Data Processing Errors	75 (6)	90 (11)	84 (7)	90 (3)	95 (6)	105 (9)	204 (21)	59 (21)	148 (12)	179 (22)
	Code Quality	72 (3)	93 (6)	82 (8)	86 (6)	96 (2)	108 (5)	206 (11)	43 (43)	168 (13)	210 (19)
	Other	70 (4)	87 (4)	81 (14)	90 (6)	97 (2)	105 (8)	197 (26)	45 (52)	152 (16)	190 (12)
	All	70 (5)	91 (0)	82 (4)	90 (1)	97 (1)	96 (6)	197 (7)	47 (8)	142 (4)	196 (7)
	Median	70 (7)	91 (5)	82 (10)	90 (4)	96 (3)	103 (11)	200 (18)	52 (40)	148 (17)	192 (21)
40%	Protection Mechanism Failure	55 (11)	84 (3)	67 (10)	81 (4)	89 (5)	101 (13)	205 (35)	62 (50)	149 (19)	170 (57)
	Resource Management Errors	61 (7)	82 (2)	77 (7)	82 (4)	91 (3)	107 (11)	213 (15)	58 (14)	154 (38)	202 (13)
	Data Processing Errors	65 (0)	84 (6)	68 (9)	82 (8)	90 (4)	105 (5)	214 (12)	63 (29)	147 (19)	199 (13)
	Code Quality	75 (3)	86 (6)	72 (6)	82 (6)	86 (6)	108 (6)	209 (17)	45 (32)	162 (16)	208 (17)
	Other	60 (10)	85 (4)	73 (12)	82 (8)	90 (1)	107 (12)	202 (24)	44 (29)	152 (19)	197 (13)
	All	59 (1)	82 (1)	72 (3)	83 (2)	93 (2)	89 (3)	191 (5)	56 (13)	145 (16)	198 (8)
	Median	61 (9)	83 (4)	71 (9)	82 (5)	90 (4)	105 (11)	204 (25)	57 (24)	151 (21)	199 (16)
50%	Protection Mechanism Failure	51 (5)	76 (3)	59 (18)	70 (4)	81 (5)	112 (14)	203 (27)	58 (43)	146 (41)	184 (11)
	Resource Management Errors	50 (5)	73 (8)	68 (8)	72 (6)	86 (2)	112 (13)	216 (19)	81 (15)	154 (20)	205 (29)
	Data Processing Errors	48 (5)	73 (8)	31 (34)	81 (9)	89 (6)	107 (8)	217 (23)	36 (30)	148 (7)	200 (11)
	Code Quality	51 (3)	81 (9)	62 (15)	75 (5)	87 (8)	110 (7)	219 (14)	61 (27)	166 (12)	215 (12)
	Other	49 (2)	74 (12)	65 (17)	71 (8)	85 (7)	108 (9)	207 (21)	34 (68)	162 (11)	202 (16)
	All	50 (1)	74 (4)	68 (3)	74 (1)	85 (3)	96 (4)	193 (19)	66 (9)	136 (17)	186 (18)
	Median	50 (4)	75 (7)	65 (20)	74 (7)	86 (6)	110 (11)	206 (25)	64 (45)	154 (21)	197 (23)

This table shows the experimental results when human error rate E_R increases from 0% to 50%. Baseline recall and cost come from Table 5, Row Text, Column 95. All the numbers are percentages and in the format of median(IQR) from 30 repeated simulations. Each row presents performances on one group of target vulnerability types in the Mozilla Firefox dataset as described in Table 1. The "Median" row summarizes the median performance across all groups. The "None" columns show the performance of HARMLESS without any error correction method while other columns each reports the performance with a different error correction method. One method is considered better than another if it has higher relative recall and lower relative cost.

inspected. This can be improved since the effort to inspect a file can vary significantly. Also, if HARMLESS can make predictions based on functions or specific lines of code instead of an entire file, it would save time and effort for the human inspector to look for potential vulnerabilities.

- Human errors are simulated randomly in the current work. In practice, many factors can affect the human error rate, e.g. the level of expertise of the inspector, or some certain type of vulnerabilities might be harder to find.
- Our simulation is based on data collected from reported security vulnerabilities and bugs. As a result, our dataset does not provide information on whether an actual incident was caused by the identified vulnerabilities. Therefore, we do not separate these two types of vulnerabilities in the simulation and we do not know what percentage of the identified vulnerabilities were involved in actual security incidents.
- Low precision issue: maintaining a considerably high precision is important for such an inspection tool to be accepted and used in practice, e.g. if the human inspector finds no vulnerabilities from 10 source code files suggested by the tool, then he or she may quickly lose faith in that tool and stop using it. Although the current HARMLESS tool looks promising in terms of recall and percentage of cost saved, it has low precision. As an example, in Table 4, HARMLESS reaches 95% recall with 20% cost when targeting all types of vulnerabilities using text mining features. This also means HARMLESS finds 257 vulnerable files by asking humans to inspect 5,750 files, which results in less than 5% precision (1 vulnerable file can be identified for every 22 files inspected). It might be acceptable in some mission-critical systems but generally speaking, users do not want to inspect 22 files only to find one vulnerability since it may take hours of careful work to inspect one source code file.

6.2 Threats to Validity

There are several validity threats [50] to the design of this study. Any conclusions made from this work must be considered with the following issues in mind:

Conclusion validity focuses on the significance of the treatment. To enhance conclusion validity, we ran each simulation 30 times with different random seeds.

Internal validity focuses on how sure we can be that the treatment caused the outcome. To enhance internal validity, we heavily constrained our experiments to the same dataset, with the same settings, except for the treatments being compared.

Construct validity focuses on the relation between the theory behind the experiment and the observation. This applies to our analysis of which feature set provides the best performance. There are two concerns here.

Firstly, when we conclude that software metrics features perform worst (i.e., it contributes little to the vulnerability prediction), other reasons such as the choice of classifier (different classifiers might work better with different feature sets) might be the real cause of the observation.

Secondly, like any other data mining paper we are susceptible to biases in the labeling of the data used to test/train this method. We have some confidence in that labeling (since two graduate students spent months of work carefully collecting those ground truth labels). Nevertheless, these two oracles could have made some systematic errors in their labeling (e.g. favoring certain kinds of vulnerabilities, and not others). To partially mitigate this problem, we make our data and methods available[15] and commit to helping (as requested) other research groups working on this data.

External validity concerns how widely our conclusions can be applied. All the conclusions in this study are drawn from the experiments running on the Mozilla Firefox vulnerability dataset. When applied to other case studies, the following concerns might arise: 1) crash dump stack trace data may not be available; 2) the same settings that work on Mozilla Firefox dataset might not provide the best performance on other datasets. One possible solution to this problem can be hyperparameter tuning which adapts the parameter settings to the target dataset.

7 Conclusion and Future Work

Reducing software security vulnerabilities is a crucial task of software development. However, inspecting code to find vulnerabilities is a tedious and time-consuming task. Hence, this paper introduces and evaluates HARMLESS, an active learning-based vulnerability prediction framework. HARMLESS focuses on (1) saving cost when reaching different levels of recall, (2) providing a practical way to stop at the target recall, and (3) correcting human errors efficiently.

This approach was tested on a Mozilla Firefox dataset using a simulation methodology. Given actual vulnerabilities reported, we run multiple what-if simulations where we run over that data using a variety of code inspection strategies. The goal of these inspections is to determine what might have happened if these strategies were applied for finding vulnerabilities before the software is released.

What we found was that using text mining features alone, HARMLESS decreases the cost to reach high recall for finding vulnerabilities before deployment, and if runtime data is available (e.g. the crash features used above), then that can boost vulnerability prediction in the early stages. We also showed that the total number of vulnerabilities in one software project can be accurately estimated during the active learning process, thus providing a reliable stopping rule for the approach. Based on our results, HARMLESS can save 77, 70, 53% of the cost (compared to reviewing and testing source code files in a random order) when applying the SEMI estimator to stop at 90, 95, 99% recall, respectively. Note that these results indicate that HARMLESS can result in, by practitioner choice, a recall value close to 100% given trade-offs in recall and cost. Practitioners can have confidence that, if they so choose and have the resources to spare, 99% of the vulnerabilities would be identified without inspecting most of the source code. Meanwhile, HARMLESS can cover 96% of the human missing vulnerable files by double-checking 50% of the inspected files, thus saving more effort when correcting human errors.

TABLE 7: Comparison to the state-of-the-art frameworks

	HARMLESS	Supervised or Semi-supervised	Unsupervised
Can reach any target recall	✓	✓	
Guide on how to reach target recall	✓		
Can start without known vulnerabilities	✓		✓
Work without other indicators	✓	✓	
Utilize continuous human feedback	✓		
Correct human errors efficiently	✓		

Table 7 compares the advantages of HARMLESS with other approaches to vulnerability prediction, i.e. supervised or semi-supervised approaches [16], [7], [26] and unsupervised approaches [28], [8], [29]. We assert that HARMLESS does best due to (a) the capability to make full use of continuous human feedback, (b) the guidance of when to stop inspections, and (c) the human error correction strategy.

That said, HARMLESS still suffers from the low precision problem as discussed in §6.1. Even after the reduction, the inspection effort required is still very high (inspecting 5,750 files in our case study). Therefore much research is required to further reduce the effort associated with vulnerability inspection. There are two possible directions for further cost reduction: (a) increasing the precision by improving the model's prediction, and (b) reducing the cost of every false alarm.

As for improving the model's prediction:

- Combining text mining features with other vulnerability-proneness metrics might improve the model's performance as suggested by the **Hybrid** results.
- Although our current way of using semi-supervised learning did not produce better results, it is still possible to improve HARMLESS's performance by better utilizing the information from unlabeled data. For example, semi-supervised learning could become a pre-filter that removes from consideration the files that are less likely to contain vulnerabilities.
- This work applies some best practice settings (featurization, classifier, active learning, etc.) from other domains, this does not necessarily be the best practice for proactive vulnerability prediction. Further testing of other settings and tuning of the parameters might provide improved results on this specific domain.

15. https://github.com/ai-se/Mozilla_Firefox_Vulnerability_Data

As for reducing the cost of every false alarm:

- One direction is to predict based on function level or lines of code level, so that the cost of every false alarm becomes the human effort of inspecting one line of code instead of one file.
- The other direction is to facilitate the inspection by highlighting which part of the code in the file contains terms used by the prediction model.

Apart from the low precision issue, considering the limitations and validity threats of the current work, other future works could include:

1) The community requires more dataset like the Mozilla Firefox vulnerability dataset we described in this paper. Hence we encourage software engineering researchers to collect more high-quality vulnerability datasets and test the generalizability of vulnerability prediction algorithms.
2) Solicit opinions on the results from the development team that would take action based on the results. In our study, we reached out to contacts at the Mozilla Foundation, but did not get a response. In the future, field research should be conducted with real human inspectors using HARMLESS.
3) As shown in §5.4, different level of human error rate E_R requires different error correction methods. E.g. if $E_R = 0$, using DISPUTE will cost 46% more than not using any error correction; and at high error rate $E_R = 50\%$, DISPUTE(3) is more desired than DISPUTE. Therefore how to accurately estimate or measure the human error rate and adjust the error correction method correspondingly becomes an important future work.
4) Check generality of the active learning framework by applying it to other total recall problems in software engineering. The authors are already exploring SE problems like test case prioritization, technical debt detection, and static warning identification. Preliminary results suggest that the active learning framework discussed here might be widely applicable to many SE problems.

We hope that this work will lay down a foundation and motivate further research on active learning-based vulnerability prediction.

ACKNOWLEDGEMENTS

This research was partially funded by a National Science Foundation Grant #1506586 and #1909516. The authors would like to thank the Mozilla Firefox team for making crash and defect data available to the public, which makes this research possible.

REFERENCES

[1] P. E. Black, L. Badger, B. Guttman, and E. Fong, "Nistir 8151: Dramatically reducing software vulnerabilities," 2016.
[2] N. Lewis, "Common vulnerabilities and exposures (cve)."
[3] L. J. Trautman, "Cybersecurity: What about us policy?" 2015.
[4] S. A. Slaughter, D. E. Harter, and M. S. Krishnan, "Evaluating the cost of software quality," *Communications of the ACM*, vol. 41, no. 8, pp. 67–73, 1998.
[5] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.
[6] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.
[7] T. Zimmermann, N. Nagappan, and L. Williams, "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista," in *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE, 2010, pp. 421–428.
[8] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams, "Approximating attack surfaces with stack traces," *IEEE/ACM 37th IEEE International Conference on Software Engineering*, pp. 199–208, 2015.
[9] C. Theisen, T. Zhu, K. Oliver, and L. Williams, "Teaching secure software development through an online course," in *Secure Software Engineering in DevOps and Agile Development, International Workshop on*. ESORICS, 2017.
[10] J. Walden, J. Stuckman, and R. Scandariato, "Predicting vulnerable components: Software metrics vs text mining," in *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE, 2014, pp. 23–33.
[11] L. Hatton, "Testing the value of checklists in code inspections," *IEEE software*, vol. 25, no. 4, 2008.
[12] M. R. Grossman, G. V. Cormack, and A. Roegiest, "Trec 2016 total recall track overview," in *TREC*, 2016.
[13] Z. Yu and T. Menzies, "Fast2: An intelligent assistant for finding relevant papers," *Expert Systems with Applications*, vol. 120, pp. 57 – 71, 2019.
[14] (2017) Mozilla foundation security advisories. [Online]. Available: https://www.mozilla.org/en-US/security/advisories/
[15] M. V. Zelkowitz and I. Rus, "Understanding iv & v in a safety critical and complex evolutionary environment: the nasa space shuttle program," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 349–357.
[16] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.
[17] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
[18] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *Journal of Systems Architecture*, vol. 57, no. 3, pp. 294–313, 2011.
[19] Y. Shin and L. Williams, "Is complexity really the enemy of software security?" in *Proceedings of the 4th ACM workshop on Quality of protection*. ACM, 2008, pp. 47–50.
[20] A. Hovsepyan, R. Scandariato, M. Steff, and W. Joosen, "Design churn as predictor of vulnerabilities?" *International Journal of Secure Software Engineering*, 2014.
[21] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 529–540.
[22] X. J. Zhu, "Semi-supervised learning literature survey," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2005.
[23] O. Chapelle, B. Scholkopf, and A. Zien, "Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]," *IEEE Transactions on Neural Networks*, vol. 20, no. 3, pp. 542–542, 2009.
[24] T. Le, P. Duong, M. Dinh, T. D. Nguyen, V. Nguyen, and D. Q. Phung, "Budgeted semi-supervised support vector machine." in *UAI*, 2016.
[25] F. Gieseke, A. Airola, T. Pahikkala, and O. Kramer, "Fast and simple gradient-based optimization for semi-supervised support vector machines," *Neurocomputing*, vol. 123, pp. 23–32, 2014.
[26] Q. Meng, S. Wen, C. Feng, and C. Tang, "Predicting buffer overflow using semi-supervised learning," in *2016 9th International Congress on Image and Signal Processing, BioMedical Engineering and Informatics (CISP-BMEI)*, Oct 2016, pp. 1959–1963.
[27] M. Gegick, P. Rotella, and L. Williams, "Toward non-security failures as a predictor of security faults and failures," *Engineering Secure Software and Systems*, pp. 135–149, 2009.
[28] C. Theisen, R. Krishna, and L. Williams, "Strengthening the evidence that attack surfaces can be approximated with stack traces," in *North Carolina State University Department of Computer Science TR2015-10, submitted to International Conference on Software Testing, Verification, and Validation (ICST) 2016*, 2015.
[29] C. Theisen, K. Herzig, B. Murphy, and L. Williams, "Risk-based attack surface approximation: how much data is enough?" in *Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), 2017 IEEE 39th International Conference on*. IEEE, 2017, pp. 273–282.
[30] B. Settles, "Active learning," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, no. 1, pp. 1–114, 2012.
[31] G. V. Cormack and M. R. Grossman, "Navigating imprecision in relevance assessments on the road to total recall: Roger and me," in *The International ACM SIGIR Conference*, 2017, pp. 5–14.
[32] ——, "Engineering quality and reliability in technology-assisted review," pp. 75–84, 2016.

[33] ——, "Scalability of continuous active learning for reliable high-recall text classification," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. ACM, 2016, pp. 1039–1048.

[34] ——, "Autonomy and reliability of continuous active learning for technology-assisted review," *arXiv preprint arXiv:1504.06868*, 2015.

[35] ——, "Evaluation of machine-learning protocols for technology-assisted review in electronic discovery," in *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. ACM, 2014, pp. 153–162.

[36] M. R. Grossman and G. V. Cormack, "The grossman-cormack glossary of technology-assisted review with foreword by john m. facciola, u.s. magistrate judge." *Federal Courts Law Review*, vol. 7, no. 1, pp. 1–34, 2013.

[37] B. C. Wallace, T. A. Trikalinos, J. Lau, C. Brodley, and C. H. Schmid, "Semi-automated screening of biomedical citations for systematic reviews," *BMC bioinformatics*, vol. 11, no. 1, p. 1, 2010.

[38] B. C. Wallace, K. Small, C. E. Brodley, and T. A. Trikalinos, "Active learning for biomedical citation screening," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2010, pp. 173–182.

[39] ——, "Who should label what? instance allocation in multiple expert active learning." in *SDM*. SIAM, 2011, pp. 176–187.

[40] B. C. Wallace and I. J. Dahabreh, "Class probability estimates are unreliable for imbalanced data (and how to fix them)," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 695–704.

[41] B. C. Wallace, I. J. Dahabreh, K. H. Moran, C. E. Brodley, and T. A. Trikalinos, "Active literature discovery for scoping evidence reviews: How many needles are there," in *KDD workshop on data mining for healthcare (KDD-DMH)*, 2013.

[42] Z. Yu, N. A. Kraft, and T. Menzies, "Finding better active learners for faster literature reviews," *Empirical Software Engineering*, Mar 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9587-0

[43] M. Miwa, J. Thomas, A. O'Mara-Eves, and S. Ananiadou, "Reducing systematic review workload through certainty-based screening," *Journal of biomedical informatics*, vol. 51, pp. 242–253, 2014.

[44] G. V. Cormack and M. R. Grossman, "The quest for total recall," in *Proceedings of the ACM Symposium on Document Engineering 2018*. ACM, 2018, p. 6.

[45] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

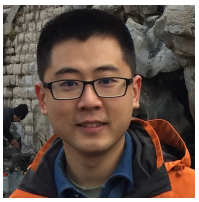
[46] T. Joachims, "Training linear svms in linear time," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 217–226.

[47] R. Krishna, Z. Yu, A. Agrawal, M. Dominguez, and D. Wolf, "The bigse project: lessons learned from validating industrial text mining," in *Proceedings of the 2nd International Workshop on BIG Data Software Engineering*. ACM, 2016, pp. 65–71.

[48] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proceedings of the 6th International Workshop on Security Measurements and Metrics*. ACM, 2010, p. 3.

[49] F. Gieseke, A. Airola, T. Pahikkala, and O. Kramer, "Sparse quasi-newton optimization for semi-supervised support vector machines." in *ICPRAM (1)*, 2012, pp. 45–54.

[50] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research-an initial survey." in *SEKE*, 2010, pp. 374–379.



Zhe Yu is a fifth year Ph.D. student in Computer Science at North Carolina State University. He received his masters degree in Shanghai Jiao Tong University, China. His primary interest lies in the collaboration of human and machine learning algorithms that leads to better performance and higher efficiency. http://azhe825.github.io/

Christopher Theisen is currently a Program Manager at Microsoft. He received his Ph.D. in Computer Science from North Carolina State University. His research is focused in software security and software engineering, specifically developing attack surface metrics. http://theisencr.github.io/

Laurie Williams (IEEE Fellow) received her Ph.D. in Computer Science from the University of Utah. Her research focuses on software security; agile software development practices and processes, particularly continuous deployment; and software reliability, software testing and analysis. Prof. Williams has more than 230 refereed publications. https://collaboration.csc.ncsu.edu/laurie/

Tim Menzies (IEEE Fellow) is a Professor in CS at NcState His research interests include software engineering (SE), data mining, artificial intelligence, search-based SE, and open access science. http://menzies.us