2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)

Classifying False Positive Static Checker Alarms In Continuous Integration Using Convolutional Neural Networks

Seongmin Lee
KAIST
Daejeon, Republic of Korea
bohrok@kaist.ac.kr

Shin Hong
Handong Global University
Pohang, Republic of Korea
hongshin@handong.edu

Jungbae Yi
Samsung Electronics
Seoul, Republic of Korea
yi_jungbae@samsung.com

Taeksu Kim
Samsung Electronics
Seoul, Republic of Korea
taeksu.kim@samsung.com

Chul-Joo Kim
Samsung Electronics
Seoul, Republic of Korea
chuljoo1.kim@samsung.com

Shin Yoo
KAIST
Daejeon, Republic of Korea
shin.yoo@kaist.ac.kr

***Abstract*—Static code analysis in Continuous Integration (CI) environment can significantly improve the quality of a software system because it enables early detection of defects without any test executions or user interactions. However, being a conservative over-approximation of system behaviours, static analysis also produces a large number of false positive alarms, identification of which takes up valuable developer time. We present an automated classifier based on Convolutional Neural Networks (CNNs). We hypothesise that many false positive alarms can be classified by identifying specific lexical patterns in the parts of the code that raised the alarm: human engineers adopt a similar tactic. We train a CNN based classifier to learn and detect these lexical patterns, using a total of about 10K historical static analysis alarms generated by six static analysis checkers for over 27 million LOC, and their labels assigned by actual developers. The results of our empirical evaluation suggest that our classifier can be highly effective for identifying false positive alarms, with the average precision across all six checkers of 79.72%.**

***Index Terms*—False alarms; Static analysis; Classification; Machine learning;**

I. Introduction

Detecting faults as early as possible is a critical task. The cost of fixing uncaught faults increases by orders of magnitude as they escape into the later stages of software development [1], eventually consuming a daunting portion of software maintenance cost [2]. Static analysis can be an effective quality assurance technique, because it can be applied directly to the source code, without any need to generate test input [3]–[5] or oracles [6]. Many static analysis techniques have been developed [7]–[9] and practically applied [10], [11]. Samsung Electronics has adopted static analysis techniques as part of its Continuous Integration (CI) pipeline [12]. Every developer commit will go through a range of static analysis checkers: any warning has to be reviewed, labelled, and stored by developers into a defect management system.

While the use of static checkers in the CI pipeline is known to be a highly effective method for identifying source code defects early in the development life cycle [13], one inherent limitation of static analysis is amplified particularly in the context of CI: its conservative over-approximation of program behaviour. By definition, this can result in a significant number of false positive alarms. If developers are repeatedly sent a large number of false positive alarms, there is a risk of developer habituation to *all* future warning. Consequently, for static analysis to be effective, it is critical to reduce the number of false positive alarms delivered to developers.

While one way of reducing false positive alarms would be to improve the accuracy of the static analysis itself, we instead focus on handling of already generated alarms [14], in particular the classification of false positive alarms. Given that zero false positive rate is practically infeasible, an *a posteriori* classification can always further improve the state of the art static analysis checkers.

Previous work on classifying static analysis alarms require either syntactic or structural feature extraction [15], [16], human assigned true positive likelihood weights for checkers [17], or user feedback on warning labels [18]. Our approach is similar to that of Tripp et al. [18], in the sense that our classifier learns from warning labels assigned by human developers. However, instead of defining a fixed set of syntactic features (whose effectiveness may be limited to a set of features), our classifier learns the lexical patterns correlated with false positive alarms at the source code token level via `word2vec` vector embedding [19] and Convolutional Neural Networks (CNNs) [20].

Our empirical evaluation of the CNN based classifier uses a historical static analysis warning datasets containing a total of about 10k manually labelled alarms, generated by six static analysis checkers for over 27 million LOC of multiple open source projects to which Samsung contributes. When trained with developer assigned labels, our classifier can classify false positive alarms with 79.72% precision and 51.09% recall on

978-1-7281-1736-2/19/$31.00 ©2019 IEEE
DOI 10.1109/ICST.2019.00048

average in cross validation studies. The results suggest that the trained classifier can be used in the CI pipeline to reduce the number of false positive alarms.

The technical contributions of this paper are as follows:

- We present a language independent classifier for static analysis alarms that does not require any feature engineering or extraction: it only requires lexical tokenization and human assigned labels to learn from.
- As a feasibility study, we conduct a large empirical evaluation of our classifier, using almost 10K alarms generated by six static checkers. These alarms are collected from large scale open source projects to which Samsung contributes.
- Results of our empirical evaluation show that token level machine learning classification can be feasible. We report an average of 79.72% precision and 51.09% recall.

II. Background

A. Motivation

Integration of automated static analysis into the Continuous Integration (CI) pipeline has been proven to be an effective quality assurance method and taken up by many organisations [13]. Samsung Electronics has also adopted a practice that all projects should be set up with a set of static analysis checkers for automated code inspection, which are to be triggered in a per-release or per-commit basis to detect faults as early as possible.

However, one inherent limitation of static analysis is particularly amplified when it is used in the context of CI: the existence of false positive alarms. Experiences at Samsung Electronics show that developers receive a non-trivial number of false positive alarms. Repeated exposures to false positive alarms can have negative impact on project productivity. Not only developers' time is wasted, but the static analysis checkers may also lose credibility, eventually resulting in developer habituation to *any* alarms.

The defect management system stores all static analysis results, together with the follow up actions taken by field engineers (such as bug fix or false positive labelling), in a database. The original purpose of this database was to systematically monitor and analyse false positive alarms. Such analyses can provide input data for better configuration of off-the-shelf checkers, or case studies for improvements of in-house static analysis checkers.

The motivation of this work is to investigate whether the data collected by the defect management system can be used to automatically classify new static analysis alarms into true and false positive alarms. Such classification can be used in different ways to improve the efficiency of static analysis based quality assurance within Samsung. For example, the classification results can be delivered to developers along with the alarms, in order to assist the developer to identify false positive alarms before preparing revisions. Alternatively, the classification results can be provided to code reviewers (or *auditors*, a role in the defect management system) to assess the developer response to alarms (see Section II-B for more details on how static analysis alarms are treated within Samsung).

Initially, we have collected and analysed 56,036 alarm records, generated from over 27 million LOC in multiple open source projects to which Samsung contributes. Out of these, we eliminated duplicate and focus on roughly 10k alarms generated by six in-house checkers. By focusing on the in-house checkers, we can have better background information from Samsung's field developers.

From the analysis of the collected data, we had two findings:

- We saw that certain checkers repeatedly produce a large number of false positive alarms. This is mainly because checkers interpret widely used programming idioms or domain specific identifiers incorrectly for their analyses.
- However, even for the same checker, we observed that there can be multiple ways false positive alarms are generated. Since we do not have the exhaustive list of such possibilities, it becomes difficult to detect and filter out false positive alarms with hand crafted rules.

The motivation for machine learning based false positive alarm classifier is based on three factors. First, the historical data collected by the defect management system, especially the developer feedback on historical false positive alarms, renders itself very well to supervised learning, as they provide the absolute ground truth. Second, our observation of historical data suggests that many of false positive alarms can be classified by lexical patterns, without requiring any feature engineering that is common to many existing work [15]–[17]. Finally, assuming that our classifier is used in conjunction with the defect management system, regular (re-)training of the classifier will allow us to overcome the weakness of rule-based filtering, i.e. that of having to define multiple new rules whenever new checkers are introduced, or new ways in which false positive alarms are generated are discovered.

B. Static Analysis with Defect Management System

Let us look into the current development workflow at Samsung. Developers are required to run a predetermined set of static analysis checkers with each commit into the code repository. This step has been implemented into the development workflow by extending existing maintenance framework based on Gerrit [21]. Previously, Gerrit existed as a gate-keeping mechanism between local repositories of developers and the project master repositories: it enforced every commit to be code-reviewed, and allowed only the project owner to merge the commit after passing all reviews. The current process adds defect management system as a new, automated code reviewer.

Figure 1 shows the software development and maintenance workflow involving defect management system, VCS, CI, and human developers. Each interaction marked by alphabet characters in Figure 1 is detailed as follow:

1) A developer pushes a commit to the stationary repository in the code review system (action a) to request a code review.
2) As the code review system receives a commit, the defect management system is automatically assigned as an alarm

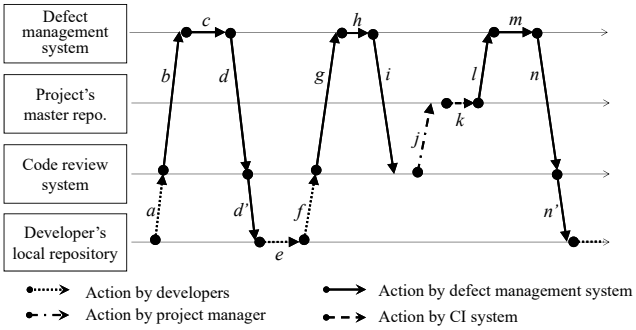

Fig. 1: A workflow of defect management system

inspector and takes the revised source code (action b). At this step, the project manager may assign other developers as code reviewers as well.

3) After running predefined static analysis checkers on the revised source code (action c), the defect management system registers all generated alarms as code review results (action d). If there are any alarms, the code review system forces the developer to resolve the issues (action d'). Otherwise, the code review system finds that the commit is ready to be merged into the master repository.
4) The developer inspects each alarm message via the code review system (action e). An alarm contains a description of the target defect pattern and a list of *witnesses*, which are source code locations associated with the detected fault. Each static analysis checker has its own definition of the witnesses (see examples in Section III-A). A developer can respond to each alarm (action f) either by submitting a new revision that fixes the defect (if the alarm is valid), or by declaring that the alarm is a false positive.
5) The defect management system automatically collects developer responses to alarms (action g). It records the alarm as either true or false positive based on the response (action h). If a developer declares a warning as a false positive, the defect management system records the feedback in the database. If, instead, the developer updates the source code, the system re-runs static analysis checkers to examine whether the alarm disappears or not. If the same checker does not generate the same warning as before, the system records the alarm as validated as true by the developer. Once all warnings are resolved, the defect management system updates its code review, stating that the revision is ready to be merged into the master repository (action i).
6) Once all code reviewers agree that the commit has no problem, the project manager merges the commit to the project master repository (action j).
7) The CI system re-builds whole projects on a regular basis (action k). Once a new build is finished, the defect management system automatically takes the new version (action l) and runs the set of static analysis checkers that check global properties (action m). These checkers examine whether any defect has been introduced due to the conflicts between different modules and packages. If a new alarm is found, the defect management system tracks the revision that introduced the alarm, and sends a code review request to the responsible developer with the new alarm message via the code review system (actions n and n').
8) The developer inspects the alarms from the newly built version and takes an action, starting again from the 1).

Note that a defect management system gathers all data of the static analysis checkers across all interactions, and stores the data in a database. The data include all input files, all generated warning messages, and the labels on each warning assigned by developers in 4).

III. Static Analysis Checkers and False Alarm Patterns

This section describes each of the studied static analysis checkers, and explains how we generate datapoints for them.

A. Static Analysis Checkers

We study six static analysis checkers in this paper. Here we explain the workings of each of them, as well as how false positive alarms can be generated by them.

1) Resource Handle Leak: The Resource Handle Leak occurs when an allocated resource handle expires before the release of the resource, resulting in a leak of the allocated resource. Its checker, `HANDLE_LEAK`, checks for two cases: functions returning without releasing resource handles stored in local variables, and functions overwriting local variables storing resource handles without releasing handles first. `HANDLE_LEAK` produces two witnesses, $w_{acquire}$ and w_{leak} : $w_{acquire}$ points to the instruction that acquires and stores a handler to a local variable, whereas w_{leak} points to the location where the handler expires. It employs an interprocedural path-sensitive analysis to discover corner cases with complicated execution paths from $w_{acquire}$ to w_{leak}.

However, `HANDLE_LEAK` can raise false positive alarms if the underlying static analyses fail. Figure 2 contains two simplified real world false positive alarms. In Figure 2(a), the checker generates a false alarm for the path from Line 5, where the handle is acquired, to the return statement in Line 8, where the function returns without releasing the allocated resource. However, this leak is infeasible because `dynamic_load()` returns zero only if it fails to acquire a dynamic library. Another similar case is described in Figure 2(b). `HANDLE_LEAK` concludes that `write_profile()` returns without closing a file handle acquired in Line 12. However, this is a false positive because the call to `_close()` in Line 18 closes the handler `fd`.

We conjecture that `HANDLE_LEAK` generates false alarms because it fails to predict the exact effect of a function call due to approximation and abstraction employed by the underlying static analysis. Simultaneously, we found that many false positive alarms share similar structural patterns. For example, the error handling paths (such as one shown in Figure 2(a)) typically start right after the resource acquisition and contain a debug or logging message. This provides supporting evidence to our conjecture that false positive alarms can be classified based on structural lexical patterns.

(a) A false alarm with a value-sensitive error-handling path

```
01: int create_file_attr() {
...
05:   ret = dynamic_load(&func_handle);
    //acquisition
06:   if (ret == 0) {
07:     debug("loading error\n");
08:     return FILE_ERROR;   // expiration
```

(b) A false alarm with a domain-specific resource-release operation

```
11: int write_profile() {
12:   fd = open(fpath, O_RDWR); // acquisition
...
18:   _close(fd);
19:   return n; }  // expiration
```

Fig. 2: Examples of false alarms from the Resource Handle Leak checkers

2) Double Free: The Double Free checker, `DOUBLE_FREE`, attempts to detect a path where the `free` operation is invoked with the same memory address twice. In such a path, the second `free` invocation would try to deallocate an invalid memory address, potentially resulting to a crash. Using an interprocedural data-flow analysis, the double free checker raises an alarm if the arguments of two `free` invocations alias each other. A double free alarm contains two witnesses, each pointing to a call site where the call sequence that includes an invocation of `free` begins.

```
01: len = length(node_list)
02: for (i = 0; i < len; i++) {
03:   node = get_element_at(node_list, 0);
04:   node_list = node_list->next;
05:   node_free(node);
06: }
```

Fig. 3: A Double Free false alarm

However, if the underlying analysis fails to capture aliases precisely, `DOUBLE_FREE` may produce false positive alarms. The example in Figure 3 shows one such case. For this alarm, the double free checker will place both witnesses on Line 5, which will executed repeatedly with the loop iteration. If the checker misses the update to `node` in Line 3, it will incorrectly conclude that `node` is being freed twice.

3) Null Pointer Dereference After Null: The Null Pointer Dereference checker, `DEREF`, raises an alarm if it detects an execution path that first evaluates a pointer `p` to `null` and subsequently uses `p`. A warning by `DEREF` has two contradictory witnesses, w_{check} and w_{ref}: w_{check} points to an instruction that checks whether a pointer `p` is `null`, while w_{ref} points to an instruction that uses `p`.

Figure 4 shows an example of a false positive alarm generated by `DEREF`. The checker decides that `obj_list` is *possibly* `null` at Line 1, because the **`if`** statement at Line 1 explicitly checks `obj_list` being `null`. Based on this, `DEREF` concludes that the value of `obj_list` passed to `g_list_append()` at Line 5 can be `null`, because a path can reaches Line 5 after taking the `false` branch at Line 1. However, the warning is a false positive because it is valid to pass `null` to `g_list_append()`: the `GList` library uses `null` to represent an empty list.

```
01: if (obj_list)
02:   length = g_list_length(obj_list);
03: for (i = length; i < capacity; i++){
04:   obj = g_new(obj_t, 1);
05:   obj_list = g_list_append(obj_list, obj);
06: }
```

Fig. 4: A Null Pointer Dereference After Null false alarm

4) Tainted Loop Termination Condition: The Tainted Loop Termination Condition checker, `TAINT_INT.LOOP`, checks whether the termination condition of a loop relies on a tainted value, such as unvalidated input from environment variables, files, or networks: such termination condition may cause the loop to go over the bound, resulting in a buffer overrun. To detect this, `TAINT_INT.LOOP` checks whether a value from unvalidated source (e.g., `getenv()`) may reach a loop condition without checking it against lower and upper bounds. If so, `TAINT_INT.LOOP` generates two witnesses: w_{taint} points to the introduction of the unvalidated value, and w_{use} points to the use of the tainted value in a loop termination condition.

`TAINT_INT.LOOP` may produces a false positive alarm when the loop itself is free from any harmful behaviour. Figure 5 shows such an example of this case. `TAINT_INT.LOOP` finds that `n` in Line 4 is a tainted value as it originates from an external source in Line 1, and is never validated. However, the loop at Lines 4–5 does not induce any error due to `n`, because the loop iterates on `arr` whose size is the same as `n`.

```
01: str = getenv("NUM_DATA");
02: n = strtoul(str, NULL, 10));
03: arr = malloc(sizeof(int) * n);
04: for (i = 0; i < n; i++)
05:   arr[i] = receive_data();
```

Fig. 5: A Tainted Loop Condition false alarm

5) Unintentional Fall Through: The checker for Unintentional Fall Through, `FALL_THROUGH`, detects a **`case`** block in a **`switch`** statement that does not end with a **`break`**. Since fall-through **`case`** blocks are often used intentionally, `FALL_THROUGH` is designed to skip analysis if the beginning of a code block is annotated with *`//fall through`*. Otherwise, the checker generates an alarm with two witnesses: w_{fall} points to the location of the **`case`** block without **`break`**, and w_{switch} points to the beginning of the **`switch`** statement.

Currently, apart from the annotation via comments, there is no way of determining whether a fall-through **`case`** block is intended or not. However, in the pilot study, we observed that there are lexical patterns developers use to communicate their intentions regarding fall-through **`case`** blocks. Figure 6 shows a simplified code snippet of a false alarm. Here, the fall-through **`case`** blocks at Line 2 and Line 6 are intended to fall-through to blocks at Line 3 and Line 7, respectively.

```
01:  switch (opt) {
02:    case SET_Opt1: a = val[Opt1];
03:    case GET_Opt1: ret = a;
04:                   break;
05:
06:    case SET_Opt2: b = val[Opt2];
07:    case GET_Opt2: ret = b;
08:                   break;
```

Fig. 6: An Unintentional Fall Through false alarm

The developer has used an empty line to communicate that the fall-through **case** blocks are intentional.

6) Unreachable Code: The checker for the Unreachable Code, UNREACHABLE, marks and reports a statement under an unsatisfiable condition as unreachable. Based on the data-flow analysis, UNREACHABLE identifies a branch predicate that is bound to a specific value. Although the unreachable code does not induce any illegal behaviour by itself, it is often a sign of flaws in the control structure.

```
01:  if (!(fp = fopen(filepath))) goto error;
02:  if (!(data = get_data(fp))) goto error;
03:  if (!(img = get_image(data))) goto error;
     ...
11:  error:
12:    if (fp) fclose(fp);
13:    if (data) free(data);
14:    if (img) free(img);
15:    return NULL;
```

Fig. 7: An Unreachable Code false alarm

However, our manual inspection revealed that many false positive alarms are generated because developers intentionally write unreachable code following the defensive programming principle. Figure 7 shows such a case. UNREACHABLE raises an alarm for Line 14, because `img` is always `null` and consequently `free(img)` cannot be executed in any case. However, the developer prefers to keep Line 14 because it is useful to explicitly release all resource handles within the function epilogue code block. Our manual inspection revealed that developers classified other similar alarms as false positives, so that they can keep defensive error handling or function epilogue code.

B. Datapoints Extraction

A datapoint is a set of lexical tokens extracted for each checker alarm based on the witnesses provided by the checker. We extract source code lines that are supposed to reflect partial intra-procedural control and data flows. For HANDLE_LEAK, DEREF, and TAINT_INT.LOOP, we extract ten lines: five lines starting from the first witness point, and another five lines leading up to and including the second witness point. For DOUBLE_FREE, we extract 20 lines in a similar way: ten lines starting from the `null` check witness point, and another ten lines leading up to the `free` witness point. For FALL_THROUGH, we extract ten lines surrounding the fall-through witness point: four preceding lines, the line that contains the witness point, and five following lines. Finally, for UNREACHABLE, we take ten lines of source code leading up to and including the unreachable point.

Note that datapoint definitions are flexible and can be tuned or modified for each project. For new checkers, the guideline for datapoint definition is that sufficient lexical information should be captured, so that humans can make similar judgement regarding false positive alarms.

IV. Vector Embedding and Classifier

This section describes how extracted datapoints are embedded into vector forms and used to train CNN based classifiers. While we only use tokens in datapoints as input to our classifiers, we need the contexts in which these tokens appear in order to correctly embed them into vector forms. Given a set of static analysis checker alarm datapoints, let us define a code chunk to be a set of all source code lines of all files that appear in the set of datapoints. First we break down the code chunk into tokens, which are then normalised. We compute the `word2vec` vector embedding [19] of all tokens in the given code chunk in order to embed to context of each token on the datapoints. The vector embedding allows us to represent a single datapoint as a matrix, i.e., a stacking of embedding vectors, each of which correspond to a normalised token in the datapoint. Finally, CNN takes the matrices of datapoints as an input and constructs a model classifying the false alarms.

A. Tokenisation and Normalisation

We use a basic C/C++ lexical tokeniser to break down source code lines in the given code chunk into tokens. Apart from separation of individual tokens, we currently do not perform any lexical analysis. However, we do perform token normalisation [22] by decomposing snake cases and camel cases. This is to provide our classifiers with generic terminology, and to avoid overfitting the training of classifiers to specific sets of identifiers. Finally, when appropriate, we allow specific whitespace characters as independent tokens. For example, classifiers for FALL_THROUGH use newline characters as tokens in order to capture the lexical pattern outlined in Section III-A5.

B. Word2Vec Embedding

The embedding of tokens into vector forms is performed by the widely used `word2vec` embedding technique [19]. `Word2vec` is a predictive modelling for learning vector embedding of words in a given corpus. It is essentially a neural network with a single hidden layer with M nodes, as shown in Figure 8.

We use the skip-gram model [23], which embeds words in a corpus into vectors in such a way that each word is identified by its neighbouring words. Let us sketch out the process intuitively. Suppose a corpus that contains a set of words, V. For the ith word $w_i \in V$, let $N(w_i)$ be the set of neighbouring words of w_i (i.e. all words that appear at most k words away from w_i in V). First, we encode each word using the one-hot vector scheme. For w_i, we get a vector of

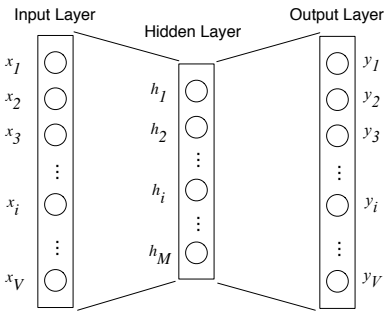

Fig. 8: A `word2vec` neural network with a single hidden layer. For the skip-gram model [23], the input vector $\vec{x}_{w_i} = (x_1, \ldots, x_V)$ for word w_i is the one-hot encoding vector for w_i. Intuitively, we train this neural network so that, given $\vec{x}_{w_i}$, the output $\vec{y}$ matches as closely as possible the sum of all one-hot encoding vectors of words that appear together with w_i in the corpus. When training is finished, we take the hidden layer weight vector, $\vec{h} = (h_1, \ldots, h_M)$, as the vector representation for the word w_i.

length $|V|$, $\vec{x}_{w_i}$, which is filled with zeros except a single one at the ith place. Now we train the neural network depicted in Figure 8, such that when $\vec{x}_{w_i}$ is given as input, the output $\vec{y}$ matches the sum of all one-hot embedding vectors of words in S (i.e. $\vec{y} = \sum_{w_j \in N(w_i)} \vec{y}_{w_j}$) as closely as possible. Once the training is finished, we take the hidden layer weights, $\vec{h} = (h_1, \ldots, h_M)$, as the vector embedding of w_i. This weight vector can be thought of as a vector that uniquely converts the one-hot encoding vector form of the original word, $\vec{x}_{w_i}$, to its neighbours, $\sum_{w_j \in N(w_i)} \vec{y}_{w_j}$. Note that the number of hidden layer weights is a configurable parameter called embedding length.

C. Convolutional Neural Network

A Convolutional Neural Network (CNN) is a neural network containing convolution layers [20]. A convolution layer contains sets of equivalent neurons, often called filters, that are connected only to a small local region in the input data: multiple instances of filters are applied to the entire input data, by moving them at given intervals (called strides). CNNs have been particularly successful in computer vision [20], [24], [25], as it can extract out the invariants that are independent from the absolute position of the data within the input region, such as translation invariants, rotation invariants, and size invariants, etc. Our conjecture is that CNNs can be also good at learning to identify the structural patterns discussed in Section III-A. The patterns for false positive alarms that we are after can often be summarised as partial orders between occurrences of specific tokens that should be size invariant (i.e. specific tokens can appear at varying distances from each other).

1) CNN Input: The datapoints extracted following the process described in Section III-B are converted to lists of tokens via tokenisation and normalisation. Subsequently, each token is embedded into a vector form using `word2vec` method described in Section IV-B. These vectors are then stacked together, resulting in one matrix per datapoint. Since every datapoint has different number of tokens, we add padding vectors on top of the matrix to make all matrix have same dimension.

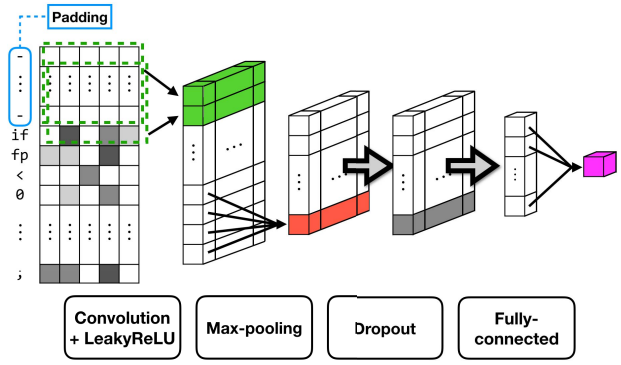

Fig. 9: CNN model

2) Network Architecture: Figure 9 shows the model of the classifier. In the convolution layer, each 32 filters of which the height is 16 and the width is equal to the `word2vec` embedding size, moves through the input matrix with stride 1×1. The one-dimension vector, retrieved from convolution and Leaky Rectified Linear Unit (LeakyReLU) layer [26], is passed into max-pooling (which forwards the maximum activation value with 4×1 filter, 2×1 stride) and drop-out layer (which drops a predetermined percentage of randomly chosen nodes during the training in order to avoid overfitting) [27], before being fed into a fully-connected layer with 16 nodes. We use the sigmoid activation function in the final output node, with binary cross-entropy loss for classification between true and false positive labels.

D. Validation Method

We aim to investigate how accurate the CNN classifiers can be and whether it can achieve a level of accuracy that is practically beneficial. To answer these questions, we train CNN based classifiers using historical static analysis alarms and developer assigned labels that show whether the given label was actually false positive. The empirical study uses the standard ten-fold cross validation method to evaluate the results of training: for each checker, the set of all alarms is divided into 10 equally sized subsets. Subsequently, a single instance of classifier is trained using nine of the subsets, and validated using the remaining one subset, resulting in ten independent evaluations. For each such fold, we compute the traditional evaluation metrics for classifiers: precision, recall, F1, and the accuracy metric.

Note that we consistently use terminology based on the nature of static analysis alarms, and not on the results of our classification: a false positive alarm means that a static analysis checker raised an alarm against a code without any defect. Consequently, precision and recall can be defined as follows:

TABLE I: Studied Static Analysis Checkers and Their Warning Dataset Size

Category	Checker	Alarms	TP	FP	FP Ratio
Call Sequence	HANDLE_LEAK	1,610	1,334	276	17%
	DOUBLE_FREE	733	622	111	15%
Dataflow	DEREF	2,101	1,919	182	9%
	TAINT_INT.LOOP	584	430	154	26%
Control Flow	FALL_THROUGH	1,680	1,559	121	7%
	UNREACHABLE	3,163	3,010	153	5%
Total		9,871	8,874	997	10%

$$\text{precision} = \frac{|\{ \text{ Actual FP alarms } \} \cap \{ \text{ Predicted FP alarms } \}|}{|\{ \text{ Predicted FP alarms } \}|}$$

$$\text{recall} = \frac{|\{ \text{ Actual FP alarms } \} \cap \{ \text{ Predicted FP alarms } \}|}{|\{ \text{ Actual FP alarms } \}|}$$

E. Data Sources

We use static analysis alarm data from multiple open source projects to which Samsung Electronics has contributed over the last two years. Samsung maintains the same CI pipeline for these open source projects, and collect alarms in a defect management system. During this two year period, the defect management system stored a total of 56,036 alarms that have been closed (i.e. either fixed or labelled as false positives) for these projects. After removing duplicates, we obtained 9,871 alarms, about 10% of which are labelled as false positives. These alarms were generated for over 27 million lines of C/C++ code. Table I presents the breakdown to individual checkers.

V. Experimental Setup

A. Configuration and Environment

We use the embedding size of 128 with `word2vec`. We use the skip-gram window size of one (i.e. we only consider adjacent tokens). `Word2vec` embedding algorithm has been implemented using `TensorFlow` [28], version 1.3, and Python version 3.6.0. The CNN classifier has the network architecture presented in Section IV-C. We use dropout rate of 0.8 and train the classifier for 150 epochs, using the mini-batch size of 10. We use the `adam` stochastic optimiser for the gradient descent optimisation [29]. All hyperparameters have been empirically tuned based on trials. The CNN classifier has been implemented using `Keras` [30], version 2.0 (using the `TensorFlow` backend), and Python version 3.6.0.

All experiments have been performed on Ubuntu 14.04 LTS, running on Intel Core i7-6700K with 32GB RAM. The `TensorFlow` backend used NVidia CUDA 8.0, running on NVidia GTX1080 GPU with 12GB memory.

VI. Results & Discussions

A. Classification Effectiveness and Efficiency

Table II reports average evaluation metric values across the ten-fold cross validation, along with the overall average across all six checkers. Figure 10 shows the distribution of evaluation metrics more clearly. The average precision across all six checkers is 79.72%, and the average recall across all six checkers is 51.09%. The cross validation mean precision is over 80% for three checkers (`HANDLE_LEAK`, `DEREF`, and `TAINT_INT.LOOP`); the cross validation mean precision is over 75% for five checkers (above three plus `DOUBLE_FREE` and `UNREACHABLE`). The highest cross validation mean precision is 86% for `DEREF` checker. Recall metric values are lower than those of precision, at around 50% on average with the lowest from `UNREACHABLE` at 31.41%. This suggests that certain false positive alarms either exhibit difficult-to-learn lexical patterns, or even lack one. However, from the point of developer effort reduction, we consider precision to be the more important metric for our use case, i.e. either assisting developers to quickly filter out false positive alarms or assisting code reviewers to understand developer commits more efficiently.

Figure 11 shows the correlation between the size of alarm datasets and the time it takes to train classifiers up to epoch 150. The training time increases linearly as the number of alarms increases. We performed linear regression analysis and obtained the following results: [time] ~ 0.031[# of alarms] $+ 9.611$. The adjusted R^2 is 0.88, and the p–value is less than $2.2e^{-16}$.

All training finished within 100 seconds. Considering that we use a single consumer grade GPU for training, we cautiously suggest from the observed data that regular (re)training of classifier neural nets would be feasible even within the context of continuous integration.

B. Discussion

Here we discuss potential issues and shortcomings observed in the results, and plan the future work.

1) Trivial Overfitting: One potential risk in every machine learning application is overfitting. In our case, there is the risk of the classifier learning to simply connect existence of specific tokens to prediction of false positive alarm labels. We intentionally designed our CNN to be as minimal as possible (see Section IV-C) to avoid overfitting. However, to investigate whether such trivial overfitting to specific tokens is actually happening, we undertook a small case study using the `HANDLE_LEAK` dataset.

Figure 12 shows the distribution of token occurrences in the code chunk for `HANDLE_LEAK`: the x axis represents the token id sorted by number of token occurrences, and the y axis shows the accumulative number of tokens in the `HANDLE_LEAK` checker code chunk that belong to the subset up to the xth token in the descending order of occurrence frequency. The corpus contains 4,987,049 tokens, of which there are 24,633 unique types. The distribution of token occurrences is heavily skewed with a significant long tail: the top 1,232 tokens (marked by the vertical red line) account for the 90% of all tokens in the corpus (marked by the horizontal red line).

With such a long tailed distribution, we conjecture that any tokens that may produce trivial overfitting will be in the tail region. Based on this, we trained multiple instances of our classifier for the `HANDLE_LEAK` checker, but using only the

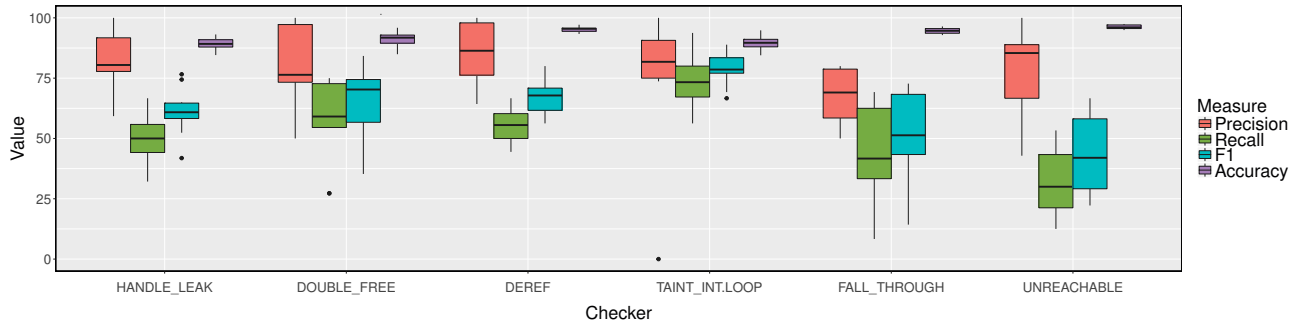

Fig. 10: Boxplots of precision, recall, F1, and accuracy metric from ten-fold cross validation of classifier training for each studied static analysis checkers. Median precision is over 75% for five out of six checkers, while median recall is over 50% for four out of six studied checkers.

TABLE II: Average accuracy results of ten-fold cross validation for 6 checkers

Checker	Precision		Recall		F1		Accuracy		Avg. # of Predicted / Actual	
	Mean	Var.	Mean	Var.	Mean	Var.	Mean	Var.	TP Alarms	FP Alarms
HANDLE_LEAK	81.80%	186.65	49.74%	90.54	61.24%	90.06	89.27%	7.15	143.9 / 133.4	17.1 / 27.6
DOUBLE_FREE	79.39%	293.09	57.50%	289.36	64.84%	229.52	90.99%	10.57	65.0 / 62.2	8.3 / 11.1
DEREF	85.70%	144.97	55.53%	53.56	66.87%	48.30	95.24%	1.08	198.1 / 191.9	12.0 / 18.2
TAINT_INT.LOOP	85.98%	101.06	73.95%	137.50	78.66%	47.64	89.50%	9.38	44.9 / 43.0	13.5 / 15.4
FALL_THROUGH	67.99%	108.47	44.42%	332.34	52.28%	293.16	94.64%	1.43	160.3 / 155.9	7.7 / 12.1
UNREACHABLE	77.48%	399.67	31.41%	216.05	43.20%	290.30	96.20%	0.84	310.0 / 301.0	6.3 / 15.3
Average	79.72%	-	51.09%	-	61.18%	-	92.64%	-	-	-

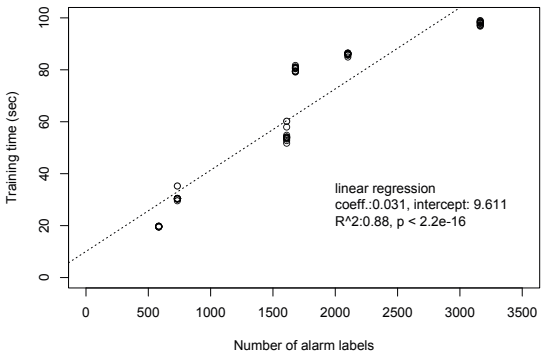

Fig. 11: Linear regression model for alarm dataset size and classifier training time: the training time increases linearly as more alarms are processed.

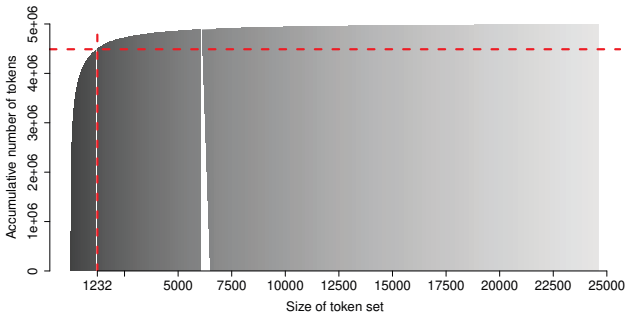

Fig. 12: Accumulative number of token occurrences in the HANDLE_LEAK code chunk

top k tokens as the vocabulary, with values of k ranking from 2,000 to 24,000 at the interval of 2,000. Figure 13 shows how average precision and recall from ten-fold cross validation of instances change as vocabulary sizes decreases. While there are fluctuations, the precision level does not drop much below 80%, while maintaining similar levels of recall values, even when we train with less than 10% of the full corpus vocabulary. Manual inspection of different vocabulary sets reveals that key tokens for the HANDLE_LEAK checker, such as fopen and errno, are indeed within the top 2,000 most frequent tokens. The result of this case study suggests that our classifiers are indeed learning structural patterns that consist of very frequently used tokens, rather than simply remembering the connections between source code specific identifiers and alarm labels.

2) Scopes of Datapoints: The quality of our results are strictly dependent on input fed into the classifiers. There are multiple decision points on how to define datapoints for checker alarms. Without customising the static analysis checkers to extract more precise information about each alarm, datapoint definitions may remain arbitrary to some degree. For example, DOUBLE_FREE provides call sequence information from the witness call sites to the actual invocation of free. However, we only included the target function in our datapoints (1) including the entire call sequence may create inhibitively large datapoints, and (2) inclusion of entire call sequences may introduce noises.

While more checker specific input may yield better results, our aim is to be as checker agnostic as possible. By being checker agnostic, we can lower the adoption cost for our

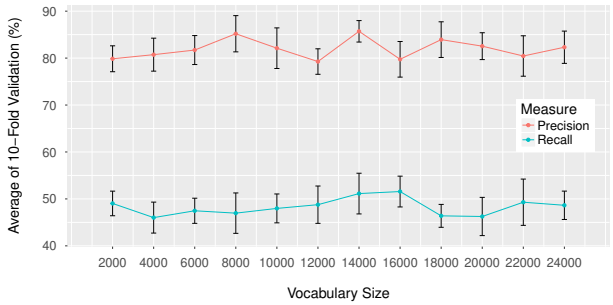

Fig. 13: Change of average cross validation precision and recall of `HANDLE_LEAK` classifier with varying vocabulary sizes: reducing the vocabulary size does not significantly damage the results of training.

classifiers (one only needs labelled historical alarm data to train classifiers) and become more flexible in terms of which static analysis checkers are used by the organisation.

3) Integration of Classifier to CI Pipeline: Based on our results, we propose two different ways of integrating our classifier to the current CI system.

1. Post-processor: The CI system can adopt our classifier as a post-processor for the static analysis checkers, by placing it between action c and d in Figure 1. With the post-processing approach, we expect to reduce the number of false positive alarms developers have to handle. However, there is the risk of losing true positive alarms, as the precision is not 100%.

2. Alarm Review Assistance: Another application would be simply to provide the classification results for the code reviewers to peruse, so that they can double-check developers responses to generated alarms more easily. This can be achieved by placing a classifier before action j in Figure 1. This approach will assist code reviewers not to miss any true positive alarms incorrectly rejected by developers.

VII. Threats to Validity

Threats to internal validity concern the extent to which the observed results from the empirical evaluation warrants our claims, such as selection biases or implementation correctness. Our datasets have been collected from an internal defect management system, without any selection criteria other than that selected static analysis checkers should contain sufficient number of warnings in the database. We plan to widen the scope of our study in the future, so that we can reduce any unintended and indirect selection biases in the choice of static analysis checkers. Human validation results of static analysis warnings not only are highly expensive to produce but also can be very sensitive information. Our classifiers have been implemented using widely studied neural network frameworks including Keras [30] and TensorFlow [28].

Threats to external validity concerns the extent to which our empirical evaluation results generalise. Since supervised learning results are directly dependent on the training data used for learning, our results are specific to the warnings generated, observed, and validated within Samsung. As with any other data driven research, our results may include a certain level of overfitting to the used training data. Furthermore, we accept developer assigned labels to be the ground truth: there is a possibility that the labels reflect preferences of Samsung engineers, instead of the absolute ground truth about the checker alarms. The question of generalisability can only be answered by future work that consider more data.

Threats to construct validity concerns how accurately the measurements we take are actually correlated to what they claim to measure. We assess the level of any threats to construct validity to be low, as all evaluation metrics we use are standard evaluation metrics for classification and are based on absolute counting of predicted labels.

VIII. Related Work

There are various existing attempt to process results from static analysis checkers so that developers can benefit from the produced warnings without suffering from a large number of false positives. Jung et al. applied a Bayesian statistical analysis to buffer overrun alarms generated from 5.3 million LOC of a commercial system and could filter out about 75% of false positive alarms [15]. However, their technique requires extraction of *syntactic symptoms*, such as whether loops exist before or after the location of alarms, etc. Yoon et al. applied Support Vector Machines (SVMs) to filter out false positive alarms [16], which were generated by a commercial static analysis tool called `Sparrow` [31]. Yoon et al. also depended on count-based features, such as occurrences of conditional and loop statements or null expressions. `EFindBugs`, developed by Shen et al. [17], prioritise static analysis checker alarms generated by the widely studied `FindBugs` checker [32], [33]. `EFindBugs` requires humans to manually assign quantitative likelihood of reporting true positive for each studied checker, using a sample warnings produced against a reference target project (Shen et al. used JDK). These *defect likelihood* weights are compiled into scores for each defect type detected by `FindBugs` and used to rank alarms. Flynn et al. generated a classification model which identifies false positive alerts on SEI CERT Coding Rule [34] using both of the features from the result of a static analysis and the data of CERT [35]. `ALETHEIA`, developed by Tripp et al. [18], is probably the closest to our approach. as it asks user feedback on a small sample of generated alarms to establish the ground truths. These are fed into a range of classifiers, all of which depend on extracted features.

Our approach differs from all of the above because we do not require any feature engineering or extraction. The input to our classifier is simply the lexical tokens from the source code lines pointed by the static analysis warnings. This eliminated any need to define new sets of features for new static analysis checkers: as long as the identification of false positive cases can be achieved based on lexical information (as we have shown in Section III for some checkers), our classifier can learn to recognise the false positive alarms without any features.

IX. Conclusion

We introduce a Convolutional Neural Network based classifier that can identify false positive static analysis checker alarms, without any need to define and extract complicated features. The classifier is trained and validated using a large historical dataset from multiple open source projects to which Samsung contributes. Out of an alarm database generated from 27 million LOC by six different checkers, we extract about 10k static analysis checker alarms, with true/false positive labels manually assigned by developers, for training and validation. Results of cross validation show that our classifier achieved an average precision of 79.72% and average recall of 51.09%. When used as aides for filtering out false positive static analysis checker alarms, we argue that high precision can have a positive impact despite low recall, as it will directly results in savings in developer time. Future work includes evaluation using a wider range of checkers and a larger set of alarms, as well as hyperoptimisation of classifier architecture and integration of classifiers into the current CI pipeline.

Acknowledgement

This work is supported by the Samsung Research, Samsung Electronics Co., Ltd. Shin Yoo, Seongmin Lee and Shin Hong are supported by the Next-Generation Information Computing Development Program (No. NRF-2017M3C4A7068179) and the Basic Science Research Program (No. NRF-2017R1C1B1008159 and No. NRF-2016R1C1B1011042) through the National Research Foundation (NRF) funded by the Korean government (MSIT).

References

[1] P. Copeland, "Google's innovation factory: Testing, culture, and infrastructure," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10, 2010, pp. 11–14.

[2] R. C. Seacord, D. Plakosh, and G. A. Lewis, *Modernizing legacy systems: software technologies, engineering processes, and business practices*. Addison-Wesley Professional, 2003.

[3] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *PLDI*, 2005, pp. 213–223.

[4] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[5] P. McMinn, "Search-based software test data generation: A survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004.

[6] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[7] J. Park, Y. Ryou, J. Park, and S. Ryu, "Analysis of javascript web applications using safe 2.0," in *Proceedings of the 39th International Conference on Software Engineering Companion*, ser. ICSE-C '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 59–62.

[8] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective context-sensitivity guided by impact pre-analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 475–484.

[9] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 73–85, 2006.

[10] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, *Moving Fast with Software Verification*, ser. LNCS. Cham: Springer International Publishing, 2015, vol. 9058, pp. 3–11.

[11] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler, "A few billion lines of code later: Using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, Feb. 2010.

[12] G. Booch, *Object-oriented design: with applications*. Benjamin-Cummings, 1991.

[13] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *International Conference on Software Engineering (ICSE)*, 2015.

[14] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation*, ser. SCAM 2016, 2016.

[15] Y. Jung, J. Kim, J. Shin, and K. Yi, "Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis," in *Proceedings of the 12th International Conference on Static Analysis*, ser. SAS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 203–217.

[16] J. Yoo, M. Jin, and Y. Jung, "Reducing false alarms from an industrial-strength static analyzer by svm," in *2014 21st Asia-Pacific Software Engineering Conference*, vol. 2, Dec 2014, pp. 3–6.

[17] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, March 2011, pp. 299–308.

[18] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "Aletheia: Improving the usability of static security analysis," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 762–774.

[19] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013.

[20] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998.

[21] Gerrit Open Source Project, "Gerrit: https://www.gerritcodereview.com," 2017.

[22] D. Lawrie, D. Binkley, and C. Morrell, "Normalizing source code vocabulary," in *2010 17th Working Conference on Reverse Engineering*, Oct 2010, pp. 3–12.

[23] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13. USA: Curran Associates Inc., 2013, pp. 3111–3119.

[24] Y. LeCun, F. J. Huang, and L. Bottou, "Learning methods for generic object recognition with invariance to pose and lighting," in *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, vol. 2, June 2004, pp. II–97–104 Vol.2.

[25] M. Osadchy, Y. LeCun, and M. L. Miller, "Synergistic face detection and pose estimation with energy-based models," *J. Mach. Learn. Res.*, vol. 8, pp. 1197–1215, May 2007.

[26] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *CoRR*, vol. abs/1505.00853, 2015. [Online]. Available: http://arxiv.org/abs/1505.00853

[27] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016.

[28] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[30] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[31] Fasoo Inc. http://www.fasoo.com/site/fasoo/menu/6900.do, "Sparrow," 2014.
[32] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *SIGPLAN Not.*, vol. 39, no. 12, pp. 92–106, Dec. 2004.
[33] N. Ayewah and W. Pugh, "The google findbugs fixit," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 241–252.
[34] SEI CERT Coding Standards. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards
[35] L. Flynn, W. Snavely, D. Svoboda, N. VanHoudnos, R. Qin, J. Burns, D. Zubrow, R. Stoddard, and G. Marce-Santurio, "Prioritizing alerts from multiple static analysis tools, using classification models," in *Proceedings of the 1st International Workshop on Software Qualities and Their Dependencies*, ser. SQUADE '18. New York, NY, USA: ACM, 2018, pp. 13–20. [Online]. Available: http://doi.acm.org/10.1145/3194095.3194100