Machine Learning for Finding Bugs: An Initial Report

Timothy Chappell[†], Cristina Cifuentes[*], Padmanabhan Krishnan[*], Shlomo Geva[†]
[*]Oracle Labs, Brisbane, Australia
{cristina.cifuentes,paddy.krishnan}@oracle.com
[†]Queensland University of Technology, Brisbane, Australia
{timothy.chappell,s.geva}@qut.edu.au

***Abstract*—Static program analysis is a technique to analyse code without executing it, and can be used to find bugs in source code. Many open source and commercial tools have been developed in this space over the past 20 years. Scalability and precision are of importance for the deployment of static code analysis tools – numerous false positives and slow runtime both make the tool hard to be used by development, where integration into a nightly build is the standard goal. This requires one to identify a suitable abstraction for the static analysis which is typically a manual process and can be expensive.**

In this paper we report our findings on using machine learning techniques to detect defects in C programs. We use three off-the-shelf machine learning techniques and use a large corpus of programs available for use in both the training and evaluation of the results. We compare the results produced by the machine learning technique against the Parfait static program analysis tool used internally at Oracle by thousands of developers.

While on the surface the initial results were encouraging, further investigation suggests that the machine learning techniques we used are not suitable replacements for static program analysis tools due to low precision of the results. This could be due to a variety of reasons including not using domain knowledge such as the semantics of the programming language and lack of suitable data used in the training process.

Keywords: Machine Learning, Static Analysis, Defect Detection

I. Motivation

Static program analysis aims at determining if a program meets certain requirements without actually running the program. As the program is not executed, the static analyser creates an abstraction of the program on which the requirements are checked [1]. Depending on the underlying abstraction, it is possible to support sophisticated path analysis including inter-procedural analysis and object-oriented features. Static analysis techniques to identify potential bugs in large code-bases [2], [3] have been very successful. To make the analysis tractable, the abstraction chosen (e.g., for all values computed by a program) is finite. The key challenges that need to be overcome in any static analysis include choosing the right abstraction to detect the relevant defect and tuning the analysis to reduce the false positive rate. For instance, one can use the pentagon or octagon domain to detect illegal memory accesses [4], [5]. The process of identifying the abstraction and tuning the analysis can take a long time, as it is reliant on the insights available to the designer of the program analysis.

The goal of our work is to determine whether off the shelf machine learning techniques can be used to detect potential defects in programs. The idea was that, if an effective low-dimensionality representation of source code could be found that preserves all the necessary information, it could be used to assist in classification tasks when paired with a sufficiently large corpus of marked-up source code. This representation can then be used in conjunction with high performance retrieval and clustering approaches can be utilised for the purpose of creating a highly scalable and precise source code classification service.

As the long term aim of our work is to eventually deploy such a tool as part of the software development cycle, the performance of the tool as measured by the number of true positive, false positive, and false negative reports must be comparable to currently used tools such as Parfait [3]. Hence the focus of our work was to determine the ideal representation to use for source code for specific defect types similar to the human-specified abstraction in static analysis. This also means that our experiments compare the performance of machine learning (which is inductive) with static program analysis (which is deductive). This is in contrast to other work such as ALETHEIA [6] which compare different machine learning techniques which refine outputs produced by static analysers. To summarise, our ultimate goal is to use machine learning techniques to derive automatically the abstraction of programs for specific bug types.

This paper is organised as follows. In Section II we present the data-set used in our experiments. This includes the programs used for training as well as testing the effectiveness of the machine learning techniques. In Section III we describe the features used to classify programs. Sections IV–VII present the details of our experimental setup and the results that we obtained. We wrap up with the conclusions we can draw from our experiments and outline some of our ideas towards improving the results.

II. Data sets

The extent to which meaningful classification of source code can be achieved is limited by the training data we have available. We have used both publicly available code-bases (including benchmarks and applications) as well as some purely internal code-bases in our experimentation. These are now described. *Begbunch* is a collections of test data sets used internally at Oracle for benchmarking Oracle's *Parfait* static analysis tool. Begbunch is divided into two main sub-collections, of which one is the *accuracy* collection. The *accuracy* collection is used for benchmarking the accuracy of Parfait, that is, its precision and recall. It consists of several suites of test programs which have had their source code marked up to indicate the location and nature of various bugs (such as buffer overflows, memory leaks and use of uninitialised data). This includes programs from well known suites including Cigital, Iowa [7] and NIST SAMATE [8]. These have been extended with some examples derived from various sources such as unit tests. All these examples are artificial in that each test case is just a few lines of code to illustrate the presence/absence of the defect. This artificial benchmark suite has been extended with some code derived from open systems for which bug reports are available. Table I summarises the contents of this collection. Approximately 20% of the functions in the Begbunch *accuracy* data set are real (from both open and closed code fragments), while the remaining functions are artificial. For our experimentation the real data is the most helpful, as it identifies what functions that have particular defects.

The second collection of programs in Begbunch is the *scalability* set of test suites. These consist of a set of independently compilable software projects that are intended to test the execution time of static analysis tools; as a result, while the code may or may not contain bugs, there is no mark-up within the source files in this data set to indicate the presence of bugs. It is a large data set with a total of 64,662 functions. However, for it to be used as training or test data for any classifier, it needs to be marked up using some other approach (such as a static analysis tool like Parfait) first. While this is not ideal, the use of static analysis to mark up the defects enables us to use these realistic programs in our experimentation.

Test suite	Suite type	# of functions
Cigital	Artificial	50
Iowa	Artificial	1686
Samate	Artificial	2366
OracleLabs-I	Artificial	25
OracleLabs-II	Real	1126

TABLE I
SUMMARY OF BEGBUNCH'S ACCURACY BENCHMARKS

OpenSolaris (release from 2008) is the third and the largest data sets used and has more than 600,000 functions. Although the entire code has not been marked up, there is a certain amount of independently-verified ground truth available for this set. This ground truth is in the form of 10,101 bugs that have been reported by a run of an old version of Parfait and manually evaluated for false positives, with bugs marked as "verified", "unverified", "false" or "wontfix".

Two limitations prevent this data set from being quite as useful as the Begbunch *accuracy* data set. The first reason is that the reports consist of classification of the defects reported by Parfait. So the defects that were not caught by Parfait (false negatives) are not included. The second is related to the granularity of the reports. The report only lists bugs, not the files and functions that Parfait was run on. Hence it is impossible to tell if a particular file is unrepresented in the data because Parfait did not detect any bugs in it, or because the file was never processed by Parfait (e.g., a conditional include that imports some files but not others).

Thus the data-set we have for training and testing include small (both artificial and real) programs for which the defects are marked up and two large collections for which the ground truth is not fully known. Apart from Parfait, the static analyzers Splint [9] and Uno [10] were also included in the benchmark for comparison purposes.

III. Feature Extraction

In order to perform machine learning on the available source code and determine what information any representation will need to incorporate, it is necessary to establish a set of features that can be extracted that contain the information needed. The initial approach taken in this instance was to extract a number of different features and progressively trim that list down until it was small enough that more computationally expensive machine learning tools could be run on it efficiently. We outline two specific source code representation techniques, viz., n-grams and code complexity features below.

For code to be analysed with Parfait it first needs to be translated into LLVM's intermediary bitcode format. This format also presents us with some useful cross-platform features that allow some of the control flow of a function to be captured without the syntax. In this case we make use of *n-grams* of the instruction op-codes (e.g. load, store, br), with each feature counting the number of times a particular sequence of *n* consecutive instructions appear in each function. While the amount of individual detail these features can give is low, the idea is that it may be possible that certain instructions or pairs of instructions present an elevated risk profile with respect to certain types of bugs; alternatively, the absence of certain instructions or pairs of instructions may make it impossible for certain types of bugs to exist in the code.

The *complexity* tool is a feature of Parfait and it computes a set of standard complexity metrics for each function in the input file. These metrics relate to how complex the control flow of the code is, among other factors. These features are relatively inexpensive to compute complexity metrics are often correlated to defects [11]. The complexity features used include lines of code, def-use chains, nesting depth and McCabe's cyclomatic complexity [12].

IV. Experiments and Results

Once a set of features is available, machine learning can be performed on these feature sets to classify functions as possessing a particular bug or not. As many machine learning tools are geared towards classifying data into one class or another, while a function can have multiple different bugs, this is handled by training a machine learning model for one specific class of bugs at a time. The models for each bug type can then be run on any input data to classify it. Each model is run entirely independently and a function is presumed to have each bug type identified; for instance, if a function is marked as positive by the buffer-overflow and memory-leak classifiers, it is presumed to contain both bugs.

To support any combination of feature sets, a number of tools and scripts were built to deal with the data using a common set of intermediary file formats. The FunctionList file format maps function IDs to functions in source files; for each function ID the path to the source code, the function name and the first and last line numbers of the function are stored. The BugList file stores the bugs that are associated with each function; each line contains the function ID, the bug type, the line number the bug occurred on and some other metadata, such as whether the bug was inter-procedural, security-related etc. The suite of tools constructed for this purpose was named 'Biscotti'. The RandomForest approach from the Weka [13] toolkit was used. RandomForest creates a large number of decision trees and combines their findings to classify an input and is used because there is no natural order of the features.

To evaluate the approach initially, 10-fold cross-validation was used. This means that the model was trained and tested 10 times for each bug type, each time testing on a different subset (a randomly selected set of 10% of the functions in the input data) and training on the remaining data. This ensures that a function is never used to both train and test the same model. The classifiers were used as-is with no tuning. Table II shows a comparison of Biscotti against static analysis tools Parfait and Splint on the Begbunch *accuracy* data set. The accuracy of each system is shown in the form "X/Y (X/Y%), Z FP", where X is the number of successfully detected instances of that bug type, Y is the total number of instances of that bug type and Z is the number of false positives (functions that were classified by the system as possessing that bug type when in fact they did not.)

V. Training and testing splits

One problem that was discovered early on is the fact that standard training and testing splits; holding back a randomly selected portion of the data set, or performing X-fold cross-validation on random subsets of the data set; is insufficient for properly evaluating machine learning static analysis approaches [14], [15]. In general, these algorithms work well within one data set, while do not work so well when migrated to other projects. In our context when a model is trained on one set of source code, it is found to have far worse performance when tested on another set of source code. This is almost certainly due to the analysis tools detecting duplicate or near-duplicate functions and assuming identical defects are found in each copy. Within the Begbunch *accuracy* data set, this problem exhibits itself largely as a result of the artificial training sets that make up the majority of the available data. As these testing suites consist of a large number of small functions designed to test for specific bugs, rather than possessing characteristics similar to buggy real-world code they instead possess characteristics similar to other functions in the same testing set that were designed to test for the same bug. As a result, the machine learning algorithms learned to recognise functions that fit a particular form and was therefore unable to recognise real bugs. Conversely, when training on real-world data the resulting model had trouble with classifying bugs in artificial data for certain bug types. Table III shows the outcome of training the machine learning approach on real-world data (that is, OracleLabs-II data) and testing the approach on artificial data (Iowa, Cigital, SAMATE, and OracleLabs-I data sets).

VI. Text Features and Dimensionality Reduction

To extend the expressiveness of the feature sets, text features were also added to the data set. While these have the effect of adding a lot of features that identify particular functions, they are nonetheless a rich source of data and problems with the data can be taken care of during training (e.g. by ensuring that the same Begbunch *accuracy* suite is never used for both training and testing). These text features consist of tokens taken directly from the text of the source code from each function. This is performed with a custom tokenisation approach that attempts to preserve C language tokens (for example, keeping != and >> together) and the output acts as a term frequency table of each identifier and syntactical atom that appears in a given function. These features should function similarly to bag-of-words features in document classification, allowing duplicate or near-duplicate functions (which may possess the same defect) to be identified.

As an aside, the source code text also consists of the mark-up that was added to the Begbunch accuracy data set in order to mark the presence of particular bugs. As these text features would be an immediate giveaway as to the presence of a bug, they were removed beforehand. The text features that were extracted from the Begbunch *accuracy* data set include tokens such as reserved words, parenthesis, constant strings and identifiers in the program. LLVM instructions that feature in the learning include alloca, bitcast and call. Overall there are more than 2500 features that are extracted. Table IV lists some of the text features (i.e., tokens in the program) extracted from the Begbunch *accuracy* data set.

We now discuss the identification of the most useful features via dimensionality reduction. With the additional text features, the full set of features has reached a considerable size which poses problems for many machine learning methods. It therefore makes sense to develop some initial method of ranking them in terms of their representational capacity as far as their effectiveness at successfully classifying source code is concerned. This way the low-value features can be excluded without much additional work. To this end, the Leave One Out Nearest Neighbour Error (LOONNE) method is employed [16]; the nearest neighbour error is calculated as the number of errors that exist when each function is classified as possessing the same bug as the nearest function in terms of Euclidean distance in the current feature space. For each feature in the current feature set, the nearest neighbour error is calculated for the feature set with that feature omitted and the feature with the least error is excluded. This process is repeated until all the features have been considered for removal. The result is a ranked list of features in terms of their value.

Bug type	Biscotti	Parfait	Splint
buffer-overflow	1065/1305 (82%), 20 FP	1073/1305 (82%), 21 FP	821/1305 (65%), 356 FP
double-free	0/23 (0%), 0 FP	16/23 (70%), 3 FP	0/23 (0%), 0 FP
format-string	0/17 (0%), 0 FP	0/17 (0%), 0 FP	4/17 (24%), 17 FP
integer-overflow	0/24 (0%), 0 FP	1/24 (4%), 9 FP	0/24 (0%), 0 FP
memory-leak	17/189 (9%), 2 FP	54/189 (29%), 24 FP	0/189 (0%), 0 FP
null-pointer-deref	0/14 (0%), 0 FP	8/14 (57%), 3 FP	5/14 (36%), 69 FP
read-outside-array-bounds	171/267 (64%), 2 FP	179/267 (67%), 4 FP	232/267 (87%), 890 FP
uninitialised-var	0/125 (0%), 6 FP	81/125 (65%), 83 FP	79/125 (63%), 134 FP
use-after-free	0/31 (0%), 0 FP	17/31 (55%), 3 FP	18/31 (58%), 7 FP
Total	**1253/1995 (63%), 30 FP**	**1429/1995 (72%), 150 FP**	**1159/1995 (58%), 1473 FP**

TABLE II
COMPARING THE EFFECTIVENESS OF BISCOTTI TO STATIC ANALYSIS TOOLS ON DIFFERENT BUG TYPES

Bug type	Biscotti	Parfait
buffer-overflow	852/1234 (69%), 5 FP	1042/1234 (84%), 19 FP
double-free	0/22 (0%), 0 FP	16/22 (73%), 3 FP
format-string	0/11 (0%), 0 FP	0/11 (0%), 0 FP
integer-overflow	0/18 (0%), 0 FP	1/18 (6%), 9 FP
memory-leak	0/179 (0%), 0 FP	46/179 (26%), 13 FP
null-pointer-deref	0/12 (0%), 0 FP	8/12 (67%), 0 FP
read-outside-array-bounds	0/241 (0%), 0 FP	164/241 (68%), 1 FP
uninitialised-var	0/121 (0%), 0 FP	79/121 (65%), 68 FP
use-after-free	0/31 (0%), 0 FP	17/31 (55%), 3 FP
Total	**852/1869 (46%), 5 FP**	**1373/1869 (73%), 116 FP**

TABLE III
COMPARING BISCOTTI AGAINST PARFAIT WITHOUT SAME-SYSTEM TRAINING

!	10	Con_GetByte	Expired	HITN	RESERVED	SLC_IP
)	20	Con_GetCommand	FILE	In	SDSSC_OKAY	
,	320x200	Con_GetFloat	FSM_IDADD	MagickExport	SGE_INTR_MAXBUCKETS	
1	4G	DDKEY_ENTER	FSM_MACRO	PATH_SEP	SGE_PL_INTR_MASK	

TABLE IV
EXAMPLE OF SOURCE CODE TEXT FEATURES

Feature type	Feature name
Text	!
Splint	Undocumented global use
Parfait	buffer-overflow
Parfait	Uninitialised var
Complexity	FuncStartLine
Complexity	Nesting
LLVM 2-gram	load, fpext
LLVM 2-gram	add, call

TABLE V
LIST OF THE MOST IMPORTANT FEATURES, AS DETERMINED BY LOONNE

The output from the LOONNE tool is a ranked list of features. Table V lists the features that were found most useful for classifying functions in this data set. The utility of some of these features is obvious; Parfait and Splint both report suspected buffer overflow bugs and there is clearly a strong correlation between this reporting and the presence of buffer overflow bugs in the code. Splint's "Fresh store not released before return" is also clearly indicative of a certain type of memory leak and a strong association with memory leaks can therefore be expected. For other features the association is far less clear.

Biscotti, using the new feature set, was once again evaluated on the Begbunch *accuracy* data set. Keeping in mind the problems associated with training and testing on the same test suite, in this next evaluation Biscotti was trained on the artificial test suites (Cigital, Iowa, Samate and OracleLabs-I) and evaluated on the real-world test suites (OracleLabs-II). The results of this evaluation are included in Table VI. This shows that, while under some circumstances the machine learning approach employed by Biscotti is capable of combining enough evidence to perform better than other static analysis tools with a low false positive rate, this success is only seen with certain types of bugs and has limited portability. The other issue is the high level of reliance the model has on the features from static analysis tools. For discovering bugs in previously unseen code these features are *insufficient*, and in these cases it is only by combining the analysis results from tools like Parfait and Splint that Biscotti is able to perform adequately.

VII. NEURAL NETWORK EXPERIMENTS

A limitation of the machine learning approaches described previously is the requirement that the feature set be specified in

Bug type	Biscotti	Parfait	Splint	Uno
buffer-overflow	39/71 (1 FP)	31/71 (2 FP)	37/71 (133 FP)	3/71 (7 FP)
format-string	0/6 (0 FP)	0/6 (0 FP)	3/6 (6 FP)	0/6 (0 FP)
memory-leak	0/8 (0 FP)	8/8 (11 FP)	0/8 (0 FP)	0/8 (0 FP)
null-pointer-deref	0/2 (0 FP)	0/2 (3 FP)	0/2 (34 FP)	0/2 (0 FP)
read-outside-array-bounds	7/25 (0 FP)	15/25 (3 FP)	16/25 (225 FP)	0/25 (0 FP)
uninitialised-var	0/4 (2 FP)	2/4 (15 FP)	1/4 (32 FP)	0/4 (13 FP)
Total	**46/116 (3 FP)**	**56/116 (34 FP)**	**57/116 (430 FP)**	**3/116 (20 FP)**

TABLE VI
EVALUATING BISCOTTI AND STATIC ANALYSIS TOOLS ON REAL-WORLD CODE

Bug type	NeuralNet	Parfait
buffer-overflow	213(TP), 56,095(FP)	221(TP), 81(FP)
memory-leak	497(TP), 47,414 (FP)	506(TP), 94 (FP)

TABLE VII
COMPARING NEURAL NET APPROACH AGAINST PARFAIT ON OPENSOLARIS

advance. This works well in cases where only a limited number of features are actually available from the data in question, but as the real data sets here are source code there could be a wealth of additional information that is being left out of the classification models to the detriment of our results. To this end a small number of initial experiments were conducted using neural networks. The idea is to construct a model that is able to learn the features to use, rather than simply classifying on a set of existing features.

While typically used for image recognition tasks, convolutional neural networks are a particular type of neural network that features one or more layers of convolutional kernels. These kernels are a (typically 2D) array of weights that are trained to fit various features of the input data, then feed their results into one or more fully connected layers in order to learn how the learned features correspond to classes.

The convolutional neural network architecture used for these experiments is a simple model based on LeNet [17] consisting of a convolutional layer of 16 neurons, followed by a fully connected layer of 40 neurons, which in turn fed into a convolutional layer of 2 neurons to perform classification. Once again, the approach used here was to train the model to recognise one particular kind of bug. To transform the data into a form that the network can handle, the text was converted into a bitmap image with one pixel per bit of input text.

Unfortunately, this approach had similar problems to the problems Biscotti had when features obtained by static analysis tools were removed from the equation; only bugs common across multiple copies of bugs were able to be consistently identified. When the evaluation is performed carefully to avoid this, the results are only marginally better than random chance. For instance, training the network on the Begbunch *scalability* data set and testing it on the OpenSolaris data set is shown in Table VII. While the true positive rates are comparable the false positive rates are prohibitive for this approach to actually be useful.

Another form of neural network that was briefly experimented with was the recurrent neural network with long short-term memory [18]. The recurrent neural network consists of nodes that are connected to themselves temporally, with these links to the past having associated weights just like normal connections between nodes in a neural net. As a result, the network can learn how much to remember and how to use the stored data. The long short-term memory variation on the recurrent neural network features memory nodes that hold information between steps and other nodes can communicate with those nodes to store or retrieve this information.

Unfortunately preliminary testing has shown that the RNN approach has similar problems to the other models that rely on superficial features for classifying source code. Multiple models were tested; both classifying the non-buggy RNN on non-buggy functions and on buggy functions with the buggy code removed, but neither was found to be acceptable. Once again, the approach is able to successfully identify duplicate functions, but this limits the portability of the approach to other data sets. The applicability of the approach to find defects is also unclear.

VIII. CONCLUSION AND FUTURE WORK

While these machine learning approaches show some potential for refining the output of other static analysis engines to reduce the rate of false positives among certain bug types, the effectiveness of all discussed approaches drops greatly when this static analysis data is not available as features. This means that the approaches, as currently designed and with no extra data, cannot compete against handcrafted static analysis tools. Any machine learning technique that relies on the results of a static analysis to perform well is not useful. The main benefit of using machine learning to reduce the time to develop the abstractions and tune the analysis is defeated. This is further compounded as each defect type needs its own training.

More specifically the lessons learned from our experiments include the following. When only structural source code features are available, only similar-function detection is feasible. This allows some functions to be classified correctly, even in data sets without significant duplication; however, the accuracy is too low and false positive rate too high to be useful as a defect finding technique. The features used in the classification process, some of which are listed in Table IV, are mainly constant tokens that appear in the programs. It is clear from Table IV that no human will identify any of these features as the cause of the bugs. Furthermore, the important features, (ignoring the mark ups which will not be available on real code) identified by LOONNE has items such as the start line and !.

There are not enough examples of marked-up real-world data in the available data sets for machine learning approaches to sufficiently generalise in order to detect new bugs. The problem is that the data set is too limited for the machine learning approaches used to learn features that generalise; when the non-general features perform better than the general features it is difficult for the model to learn those general features. This is even more of a problem for neural network approaches that provide no manual control over the features that are selected; as a result, the non-general features are almost guaranteed to be preferred. While generating input programs is not a challenge, marking real bugs is too time consuming to be useful in practice. As this has to be done for each bug-type, the generality of the approach is unclear. In other words, the time spent in marking up real bugs for the training can be used to develop the specific static analysis. Another approach is to have both the program with the bug and the program without the bug in the machine learning process [19]. But their technique also relies on test suites from which invariants are derived raising the issue of scalability. The neural network approach does not seem to yield practical results. Note that for the experiments different versions of Parfait were used for the training and for the comparison. While this could bias the results towards Parfait, the results for the neural network approach is not promising as the false positive rate is not comparable to Parfait's rate and overall, is only better than random.

Our experiments lead us to wonder if techniques used in unrelated domains such as image recognition or natural language processing, are suitable for defect detection. Programming languages have a notion of semantics and these defects can be defined using this semantics. However, the syntax based approach ignores the underlying semantics and it is not clear if such semantics can be learned. Machine learning techniques using program features such as abstract syntax trees (ASTs) have been proposed [20], [21]. While they are useful in detecting similar code, it is not clear if they can be used to detect defects in general code. It is important to note that the work reported here is only an initial investigation and has many limitations. Exploring different choices of machine learning techniques, representation of input programs and training sets could yield more positive results.

Given that ground truth generation via manually marking up the large representative source code is infeasible, the following are considered to be potentially useful alternatives. One could use semi-supervised learning. Given the large amount of unlabelled input data, one potential strategy is to build a classifier trained on the labelled data and use it to classify the rest of the data so that a larger corpus of training data is available. While this approach still has the problem of potentially giving misleading results due to the input data not being general enough, semi-supervised learning is known to provide superior results compared to only training on the smaller subset.

It is also necessary to determine another means of evaluating code for the purposes of producing training data. While this may not be as high quality nor as versatile as simply having more manually marked up input data, other tools can potentially be used to detect bugs for the purposes of marking up input source automatically.

In summary, while the initial results of our tool Biscotti looked encouraging, closer investigation revealed the interdependence between static analysis and the training phase for the machine learning techniques. Thus we have not made significant progress towards our goal of replacing handcrafted static analysis tools with machine learning techniques.

References

[1] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, 2nd ed. Springer, 2005.

[2] 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," *Communications of the ACM*, vol. 53, pp. 66–75, 2010.

[3] C. Cifuentes, N. Keynes, L. Li, N. Hawes, and M. Valdiviezo, "Transitioning Parfait into a development tool," *IEEE Security and Privacy*, vol. 10, no. 3, pp. 16–23, May/June 2012.

[4] F. Logozzo and M. Fähndrich, "Pentagons: a weakly relational abstract domain for the efficient validation of array accesses," in *Proceedings of SAC*, 2008, pp. 184–188.

[5] A. Miné, "The octagon abstract domain," *Higher-Order and Symbolic Computation*, vol. 19, no. 1, pp. 31–100, 2006.

[6] O. Tripp, S. Guarnieri, M. Pistoia, and A. Aravkin, "ALETHEIA: Improving the usability of static security analysis," in *CCS*. ACM, 2014, pp. 762–774.

[7] G. R. Luecke, J. Coyle, J. Hoekstra, M. Kraeva, Y. Li, O. Taborskaia, and Y. Wang, "A survey of systems for detecting serial run-time errors," *Concurrency and Computation – Practice and Experience*, vol. 18, no. 15, pp. 1885–1907, 2006.

[8] "NIST SAMATE – software assurance metrics and tool evaluation," http://samate.nist.gov.

[9] D. Evans and D. Larochelle, "Improving security using extensible lightweight static analysis," *IEEE Software*, pp. 42–51, 2002.

[10] G. J. Holzmann, "UNO: Static source code checking for userdefined properties," in *Integrated Design and Process Technology*, 2002.

[11] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, "Software fault prediction metrics: A systematic literature review," *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.

[12] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, pp. 308–320, 1976.

[13] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2016.

[14] K. H. Esbensen and P. Geladi, "Principles of proper validation: use and abuse of re-sampling for validation," *Journal of Chemometrics*, pp. 168–187, 2010.

[15] U. M. Braga-Neto, A. Zollanvari, and E. R. Dougherty, "Cross-validation under separate sampling: Strong bias and how to correct it," *Bioinformatics*, 2014.

[16] S. Geva, "Boosting the performance of nearest neighbour methods with feature selection," in *Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining*. Springer, 2001, pp. 210–221.

[17] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," in *Proceedings of the IEEE*, vol. 86, 1998, pp. 2278–2324.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735– 1780, 1997.

[19] Y. Brun and M. D. Ernst, "Finding latent code errors via machine learning over program executions," in *ICSE*. ACM, 2004, pp. 480–490.

[20] F. Y. M. L. K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *ACSAC*. ACM, 2012, pp. 359–368.

[21] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Conference on AI*, 2016.