PREPRINT

QuASoQ 2022

10th International Workshop on
Quantitative Approaches to Software Quality

co-located with APSEC 2022
virtual, December 6th, 2022

Editors:

Horst Lichter, RWTH Aachen University, Germany
Selin Aydin, RWTH Aachen University, Germany
Thanwadee Sunetnanta, Mahidol University, Thailand
Toni Anwar, University Petronas, Malaysia

Table of Content

A Dynamic Model Selection Approach to Mitigate the Change of Balance Problem in Cross-Version Bug Prediction

Hiroshi Demanou[1], Akito Monden[1] and Masateru Tsunoda[2]

[1]*Okayama University, 1-1, Tsushima-Naka, 3-chome, Kita-Ku, Okayama 700-8530, Japan*
[2]*Kindai University, 3-4-1, Kowakae, Higashiosaka City, Osaka 577-8502, Japan*

Abstract
This paper focuses on the "change of balance" problem in cross-version bug prediction where the percentage of buggy modules changes between different versions. Such difference badly affects the prediction performance. To mitigate this problem, this paper employs a dynamic model selection approach equipped with two prediction models (always-buggy model and always-non-buggy model) and Bandit algorithm to select better models in each one-module-by-one prediction. An experiment with data sets of 20 releases of 10 open source software showed that the proposed approach can improve F1-measure compared with the conventional cross-version prediction.

Keywords
software quality assurance, defect-prone module prediction, Bandit algorithm

1. Introduction

Defect-prone software module prediction (or simply, bug prediction) has been studied for the effective software quality assurance [1][2][3]. Typically, prediction is held in cross-version situation where a prediction model is built from data of a past project, and it is applied to the next version of that project. Based on the prediction result, practitioners can allocate limited testing efforts to the defect-prone (buggy) modules to find more bugs with smaller effort.

However, it has been pointed out that such cross-version prediction very often does not work well because of concept drift [4][5]. As one of the factors of concept drift, this paper focuses on the "change of balance" between the number of buggy modules and not buggy modules. Indeed, such balance very often changes between different versions of software. For example, in case of Ant project, the percentage of buggy modules was 10.9% in version 1.5 while it becomes 29.3% in next version 1.6 (as shown in Table 2). Such a difference badly affects the prediction performance of the models in general.

To mitigate this problem, assuming that the modules are predicted one by one manner, this paper employs a dynamic model selection approach equipped with two prediction models: (1) always-buggy model and (2) always-non-buggy model. The always-buggy model outputs "buggy" to any modules input to the model, while the always-non-buggy model outputs "not buggy" to any modules. By employing Bandit algorithm to give scores to two models in each one-by-one prediction, we expect that the relatively better model is wisely selected regardless of the percentage of bugs in the target version.

To date, there are several attempts to employ Bandit algorithm in bug prediction [6][7], none of them tries to solve the "change of balance" problem in cross-version bug prediction.

To evaluate the proposed method, this paper conducts an empirical study using datasets of 20 releases of 10 open source software projects.

2. Cross-version bug prediction and its balance problem

2.1. Cross-version bug prediction

To date, various bug prediction techniques have been proposed and evaluated [1][2][3][7][8]. Bug prediction is carried out before software testing and/or code review. In this paper, we focus on bug module classification, which aims to classify a module as buggy (containing one or more bugs) or not buggy (containing no bug). The objective variable is the probability that a module belongs to the buggy class. Typically, prediction is held in cross-version manner where a prediction model is built from data of a past project, and it is applied to the next version of that project.

2.2. Problem of change of balance between versions

In cross-version bug prediction, balance between the number of buggy modules and not buggy modules is a dominant factor of prediction accuracy. For example, if

QuASoQ 2022: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual
demanouconstance@yahoo.fr (H. Demanou); monden@okayama-u.ac.jp (A. Monden); tsunoda@info.kindai.ac.jp (M. Tsunoda)

buggy modules are extremely less than not buggy modules, it is very difficult to gain high precision [9]. Moreover, such balance very often changes between different versions, and this cause bad prediction performance in cross-version prediction.

In case the older version has fewer bugs than the newer version, precision tends to become large and recall tends to become small. On the other hand, if the older version has more bugs than the newer version, precision tends to be small and recall tends to be large. For example, in case of Ant project, the older version has fewer bugs (as shown in Table 2), and in such a case, precision is high (.600) but recall becomes small (.163) as later shown in Table 4. Such change of balance between versions very often happens in cross-version prediction, making it difficult to obtain high prediction accuracy.

3. Proposed solution

3.1. Bandit algorithm

Here, we introduce Bandit algorithm which we employ in this paper to mitigate the problem of balance between versions. The K-arm bandit problem is a problem where a user faced with slot machines, must decide which machines to play. Each machine gives different average reward that the user does not know in advance. The goal is to maximize the user's cumulative reward. Consider K slot machines. In game turn n, the user will receive a reward which depends on the machine he chooses. A basic example is the case where machine i brings a reward of 1 with probability p and -1 with probability $1 - p$. In our study, considering that a user wants to conduct unit testing for a set of modules, instead of selecting a slot machine, the user selects a module (*i.e.*, a source file) one-by-one and tries to find the bug prediction model that brings the highest average reward (i.e. prediction performance) to conduct testing. The strategy for the armed bandit problem is an algorithm that chooses the next prediction model based on previous choices and the rewards obtained. This paper introduces the most basic algorithm called epsilon-greedy algorithm. In this algorithm, in each trial, an arm is selected at random for a proportion ϵ, and the best arm (having the largest total reward) is selected for a proportion $1 - \epsilon$. The proper ϵ value may depends on the context. As a simple example, here we consider two arms X and Y exist, and want to maximize the cumulative reward by selecting appropriate arm. An example of arm selection in each trial is illustrated in Table 1. Each trial is proceeded as follows.

1. In the initial trial, an arm is selected randomly. Arm X is selected in this case. Earned reward is -1.

Table 1
An example of applying a bandit algorithm.

Trial	Selected Arm	Earned reward	X's total reward	Y's total reward
1	X	-1	-1	0
2	Y	1	-1	1
3	X	1	0	1

2. Arm Y is selected for a proportion $1 - \epsilon$, as Arm Y's total reward is larger than that of X. Earned reward is 1.
3. The arm is randomly selected for a proportion ϵ. Arm X is selected in this case. Earned reward is 1.

As we illustrated above, arm X received the reward of -1 in the initial trial, and this makes arm X difficult to be selected in later trials. However, the parameter ϵ enables arm X to be selected sometimes to give it to receive positive reward.

3.2. Basic idea to solve the change of balance problem

The problem of change of balance between two versions can be classified into the following two cases (a) the newer version has fewer bugs, or (b) the newer version has greater bugs. The problem here is that it is not possible to determine in advance whether we are in case (a) or (b). However, if the bug prediction and testing is carried out on a one-by-one basis, we are gradually getting to be aware of it. That is, we assume the following process: (1) we conduct bug prediction to all modules, (2) we pick a single module that has highest probability of being "buggy", (3) conduct testing if the module is predicted as "buggy" and (4) now we know the prediction is correct or wrong. Repeating the above process, we expect that the false-positive will increase if we are in case (a). On the contrary, we expect that the true-positive will increase if we are in case (b). Based on the above expectation, our idea is to employ two different types of bug prediction models as follows:

1. Always-buggy model: It always predicts that there is a bug in a module.
2. Always-non-buggy model: It always predicts that there is no bug in a module.

Firstly, we employ a normal bug prediction model to predict all modules to obtain the probability of being "buggy" of each module. Then, a one-by-one prediction process is carried out such that: a module having the largest probability is picked, the prediction model is selected by Bandit algorithm, prediction result is obtained, the reward is given to all prediction models based on the correctness of the prediction

3.3. Proposed algorithm

Based on the basic idea above, we propose an algorithm to select a bug prediction model as follows.
(Step. 1) Probability computation

In this step, bug prediction is conducted to all modules using the ordinary model to obtain the probability of being "buggy" of all modules.
(Step. 2) Target module selection

We pick a single module that has the highest probability of being "buggy", from a list of unselected modules. The reason why we start with the buggiest module is that, it is natural for a practitioner to focus first on the riskiest part of the software and examine it to see if there is any bug or serious problem.
(Step. 3) Model selection based on the epsilon-greedy algorithm

Generate a random number x of $[0, 1]$; and,

3-1) if $x < \epsilon$, a bug prediction model is randomly selected from two models (always-buggy and always-non-buggy).

3-2) If $x \geq \epsilon$, select a bug prediction model with the largest sum of recent reward.
(Step. 4) Prediction

Conduct bug prediction with the selected model.
(Step. 5) Testing

Conduct testing if the selected model predicts the target module as "buggy." No test is carried out if "not buggy" is predicted. This is because bug prediction aims to reduce the cost of testing by testing only the modules likely to have a bug.
(Step. 6) Rewarding

Assuming that predictions were made by both two models, the reward +1 is given to a model if the prediction was correct, and -1 is given if the prediction was incorrect. Note that rewarding is conducted only if testing is conducted in Step 5. In Section 3-1, in the conventional Bandit algorithm, the reward was calculated for only one selected arm (*i.e.* bug prediction model), but in our proposal, the reward for all models is calculated. Because, in the case of a slot machine, we can only bet by putting money in one of them each time, but since bug prediction can be executed by all prediction models in every trial, there is no point in limiting the calculation of reward to a single prediction model. Therefore, we decided to use both models in each trial.
(Step. 7) Compute the sum of recent rewards for all models.

Here, we ignore "old" rewards because we want to select a model with good "recent" performance. Therefore, we set a threshold w on the number of trials, and the calculation of total reward includes only the recent w trials. We refer to this threshold w simply as "window size." The optimum w is experimentally determined.
(Step. 8) If the list of unselected modules is empty then end else go to Step. 2.

Table 2
20 releases of 10 data sets used in the experiment.

Project Name	Release	Modules	Modules with bugs	% of modules with bugs
Ant	1.5	293	32	10.9
	1.6	350	92	26.3
Camel	1.4	856	144	16.8
	1.6	945	188	19.9
Forrest	0.7	29	5	17.2
	0.8	32	2	6.3
Ivy	1.4	241	16	6.6
	2.0	352	40	11.4
Jedit	4.2	367	48	13.1
	4.3	492	11	2.2
Log4j	1.1	109	37	33.9
	1.2	205	189	92.2
Lucene	2.2	247	144	58.3
	2.4	340	203	59.7
Poi	2.5	384	248	64.6
	3.0	441	281	63.7
Prop	4	8702	840	9.7
	5	8506	1298	15.3
Synapse	1.0	157	16	10.2
	1.1	222	60	27.0

4. Evaluation

4.1. Data set

As shown in Table 2, this paper uses 20 releases of 10 open source software (OSS) project data sets to conduct cross-release prediction. Each project includes two releases where older release is used as a fit data set (for building a defect prediction model) and newer release is used as a test data set (for evaluation). The percentage of modules widely varies among projects and/or versions (smallest is 2.2% and largest is 92.9%.) Metrics included in these data sets are shown in Table 3. These data sets are donated by Jureczko et al. [10][11] and the details of the data measurement are described in [11]. We obtained these data sets from SeaCraft repository [12].

Table 3
Metrics used in the data sets.

Name	Definition
WMC	Weighted Methods per Class
DIT	Depth of Inheritance Tree
NOC	Number of Children of a class
CBO	Coupling Between Classes
RFC	Response for a Class
LCOM	Lack of Cohesion in Methods
LCOM3	Lack of Cohesion in Methods
NPM	The number of public methods
DAM	Data Access Metric
MOA	Measure of Aggregation
MFA	Measure of Functional Abstraction
CAM	Cohesion Among Methods of Class
IC	Inheritance Coupling
CBM	Coupling Between Methods
AMC	Average Methods Complexity
Ca	Afferent couplings
Ce	Efferent couplings
MaxCC	Maximum value of cyclomatic complexity of methods in a class
AvgCC	Arithmetic mean of cylcomatic complexity of methods in a class
LOC	Lines of Code

4.2. Bug prediction model

This paper employ random forest because it was shown as one of the best models in bug prediction [13] and it shows performance comparable to the modern auto-ML framework [14]. Although there exist various other predictors, improvement of defect prediction accuracy by employing them is out of scope of this study. To build random forest models, we use the statistical computing and graphics toolkit R and its randomForest library. We use the default parameter values of randomForest library, e.g. the number of trees to grow ntree = 500, and the number of variables randomly sampled as candidates as each split mtry = sqrt(p), where p is the number of predictor variables.

4.3. Accuracy measures

This paper employs three commonly used accuracy measures to evaluate the prediction performance: precision, recall and F1-measure.

4.4. Result and discussion

Table 4 shows the result of defect prediction by the conventional method, that is, cross-version bug prediction with random forest. For the project Ivy, the values of precision and recall are zero, in such a case we consider the F1-measure to be zero. The average of precision (0.488) is higher than that of recall (0.322). The average of F1-measure (0.317) is similar to that of recall.

Table 4
The bug prediction performance of the conventional method.

Project Name	Precision	Recall	F1
Ant	.600	.163	.256
Camel	.481	.266	.342
Forrest	.200	.500	.286
Ivy	0	0	0
Jedit	.132	.455	.204
Log4j	.947	.286	.439
Lucene	.650	.685	.667
Poi	.744	.722	.733
Prop	.493	.026	.050
Synapse	.636	.117	.197
Average	.488	.322	.317

Table 5 shows the result of the proposed method for window size $w = N/A$, 10, 50 and 100, and $\epsilon = 0$, .1, .2, .3 and .4. Here, $w = N/A$ means there is no window (it can be considered that $w = \infty$). The gray cells in the table have the highest values in each window size.

For all w and $\epsilon \geq .2$ cases, the average F1-measure was better than that of the conventional method, which suggests the effectiveness of the proposed method. Compared with the conventional method, the average precision was decreased, but the average recall was greatly improved, resulting in the improved F1-measure. Since the overlook of bugs is crucial in software testing, we believe improvement of recall is preferable from the practical point of view.

Interestingly, $\epsilon = .2$ or .3 showed the best performance for all window sizes. Since $\epsilon = .2$ and .3 cases are always better than $\epsilon = 0$ cases, this suggests the effectiveness of the epsilon-greedy algorithm for dynamic model selection. On the other hand, the window size was found to have negative effect on prediction performance since $w = N/A$ cases showed better performance than $w = 10$, 50 and 100 cases. Therefore, it can be said that the window is not necessary in the current form of the proposal. For more detailed analysis, Table 6 shows the prediction performance of the proposed method ($w = N/A$, $\epsilon = .2$) for each data set. Compared to the result of the conventional method (Table 4), 7 data sets (Ant, Camel, Ivy, Log4j, Lucene, Prop and Synapse) showed improvements in F1-measure, while 3 data sets (Forrest, Jedit and Poi) showed decrease in F1-measure. Looking at Table 2, it seems that the Forrest data set is too small to evaluate. It has only 2 buggy modules in new version. Also, the Jedit data set would be inadequate for evaluation since it contain only 11 buggy modules out of 492 modules. When ignoring these two data sets,

Table 5
The average bug prediction performance of the proposed method.

w	ϵ	Precision	Recall	F1
N/A	0	.380	.346	.310
	.1	.410	.437	.365
	.2	.411	.497	.393
	.3	.397	.497	.386
	.4	.393	.525	.390
10	0	.415	.261	.219
	.1	.400	.285	.302
	.2	.421	.383	.356
	.3	.413	.425	.359
	.4	.392	.430	.351
50	0	.371	.318	.263
	.1	.448	.423	.387
	.2	.427	.459	.374
	.3	.395	.401	.349
	.4	.394	.473	.363
100	0	.369	.331	.266
	.1	.408	.436	.366
	.2	.411	.485	.379
	.3	.376	.430	.350
	.4	.379	.460	.363

the average F1-measure by the conventional method is .357 and that in the proposed method is .459. Regarding the Poi data set, where the proposed method was not effective, the reason for this result may be that the newer version has fewer bugs than the older version. Based on the investigation of gained rewards in each trial in this data set, we found that always-buggy models received many minus rewards, while always-non-buggy models did not. This is considered to be not fair for always-buggy models because testing is conducted only if the prediction result is "buggy." Therefore, always-buggy models have larger chance to get minus rewards than always-non-buggy models. Resolving such asymmetries is an important issue for the future.

5. Threats to validity

In this section we discuss the threats to validity of our work. We used the single prediction method (random forest). Our important future work is to employ other prediction methods to increase the validity of the result. Another issue is that we conducted only one trial (i.e. no repetition) for each prediction. Since random forest can output different results for the same data set, it is our future work to conduct repetitions in predictions.

In this study we used data sets of 20 releases of 10 open source software donated by Jureczko et al. [10][11]. In future, we will consider using data sets from other data sources to increase the generalization of the results.

Table 6
Details of the result of the proposed method (no window, $\epsilon = .2$).

Project Name	Precision	Recall	F1
Ant	.437	.598	.505
Camel	.342	.356	.349
Forrest	.143	.500	.222
Ivy	.035	.025	.029
Jedit	.026	.091	.041
Log4j	.924	.904	.914
Lucene	.603	.897	.721
Poi	.623	.765	.687
Prop	.186	.129	.153
Synapse	.381	.267	.314
Average	.411	.497	.393

In addition, we used three commonly-used performance measures (precision, recall and F1-measure) for evaluation. However, there are several criticism to these measures [9]. Therefore, we will consider adding other performance measures such as probability of false alarm (pf) and Matthews Correlation Coefficient (MCC) [15].

Simulating a sectioning command by setting the first word or words of a paragraph in boldface or italicized text is not allowed.

6. Conclusion

In this paper, we proposed an approach to mitigate the "change of balance" problem in cross-version bug prediction. An experimental evaluation with 10 data sets showed that, by using the proposed approach, although the average precision was decreased, the average recall was greatly improved, resulting in the improved F1-measure. Since the overlook of bugs is crucial in general, we believe that improvement of recall helps practitioners in software quality assurance.

There are several future works as we denoted in the threats to validity session. In addition, this paper compared the proposed method with the most basic cross-version prediction using random forest. Since there are attempts to mitigate the class imbalance problem, such as over/under sampling [1], it is our important future work to compare our approach with these methods.

7. Acknowledgement

This work was supported in part by JSPS KAKENHI Grant number JP20K11749 and JP20H05706.

References

[1] K. E. Bennin, J. Keung, P. Phannachitta, A. Monden, S. Mensah, Mahakil: diversity based oversampling approach to alleviate the class imbalance issue in software defect prediction, IEEE Trans. Software Engineering 44 (2018) 534–550.

[2] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, A. E. Hassan, Revisiting common bug prediction findings using effort aware models, Proc. 26th IEEE Int'l Conference on Software Maintenance (ICSM2010) (2010) 1–10.

[3] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, K. Matsumoto, Assessing the cost effectiveness of fault prediction in acceptance testing, IEEE Transactions on Software Engineering 39 (2013) 1345–1357.

[4] J. Ekanayake, J. Tappolet, H. C. Gall, A. Bernstein, Tracking concept drift of software projects using defect prediction quality, Proc. IEEE Working Conference on Mining Software Repositories (2009).

[5] M. A. Kabir, J. W. Keung, K. E. Bennin, M. Zhang, Assessing the significant impact of concept drift in software defect prediction, Proc. IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC'19) (2019).

[6] T. Asano, M. Tsunoda, K. Toda, A. Tahir, K. E.Bennin, K. Nakasai, A. Monden, K. Matsumoto, Using bandit algorithms for project selection in cross-project defect prediction, Proc. International Conference on Software Maintenance and Evolution (ICSME) (2021) 19–33.

[7] T. Hayakawa, M. Tsunoda, K. Toda, K. Nakasai, A. Tahir, K. E. Bennin, A. Monden, K. Matsumoto, A novel approach to address external validity issues in fault prediction using bandit algorithms, IEICE Transactions on Information and Systems E104.D (2021) 327–331.

[8] T. M. Khoshgoftaar, A. Pandya, D. Lanning, Application of neural networks for predicting program fault, Annals of Software Engineering 1 (1995) 141–154.

[9] T. Menzies, A. Dekhtyar, J. Distefano, J. Greenwald, Problems with precision: A response to comments on data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 33 (2007) 637–640.

[10] M. Jureczko, L. Madeyski, Towards identifying software project clusters with regard to defect prediction, Proc. 6th International Conference on Predictive Models in Software Engineering (PROMISE'10) (2010) 9:1–9:10.

[11] M. Jureczko, D. Spinellis, Using object-oriented design metrics to predict software defects, In Models and Methods of System Dependability. Oficyna Wydawnicza Politechniki WrocÅCawskiej (2010) 69–81.

[12] T. Menzies, R. Krishna, D. Pryor, The seacraft repository of empirical software engineering data, https://zenodo.org/communities/seacraft (2017).

[13] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Trans. on Software Engineering 34 (2008) 485–496.

[14] K. Tanaka, A. Monden, Z. Yücel, Software defect prediction using automated machine learning, Proc. 20th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD 2019) (2019) 490–494.

[15] D. Chicco, G. Jurman, The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation, BMC Genomics 21 (2020).

A Composite Discover Method for Gadget Chains in Java Deserialization Vulnerability

Zhaojia Lai[1], Haipeng Qu[1,*] and Lingyun Ying[2]

[1]*Ocean University of China, Qingdao, China*
[2]*QI-ANXIN Techlology Research Institute, Beijing, China*

Abstract
The Java deserialization vulnerability is the most dangerous and widely affected. Since this vulnerability was proposed, numerous security practitioners have studied it and developed related detection and defence tools. The discovery of the program's potential gadget chains is the most effective defensive measure. Previously, gadget chains have relied on manual search. Automating discover gadget chains is essential for Java security. However, there are no practical tools to achieve this.

So, we propose a new composite discovery method that generates the corresponding byte streams based on the static analysis results and performs deserialization detection. Our innovation combines serialization protocols and reflection mechanisms to generate objects dynamically and implement attack injection and detection. The evaluation verified its effectiveness, where we found 52 available gadget chains in Apache Commons Collections.

Keywords
Java security, static analysis, dynamic verification

1. Introduction

Java serialization is a mechanism for converting an object to a byte stream, which significantly expands the ability of Java programs to transfer objects in networks[1] and provides the condition for RMI(Remote Method Invocation)[2]. Java deserialization is the reverse of Java serialization. It reconstructs a byte stream to an objects[3]. However, this process could trigger some magic methods that can spontaneously call other methods, perhaps even another magic method. Magic methods make up the gadget chain[4].

An attacker can construct a byte stream to control the method call chain during deserialization and trigger dangerous methods. It could cause a privilege escalation, information disclosure, and RCE(remote code execution)[5]. In addition, this vulnerability is also widely spread. It has dramatically affected many well-known programs such as Weblogic, Jboss, etc[6]. The total of Java deserialization vulnerabilities in CVE(Common Vulnerabilities & Exposures) is increasing yearly. In 2021, up to 17% of Java-related CVEs are related to Java deserialization. Most of such vulnerabilities are high-risk due to RCE(eg.CVE-2021-36981[7], CVE-2021-35464[8]).

Although there are already programs[9, 10, 11] to detect and intercept such attacks, we prefer to detect potential gadget chains during the development process to maintain the security and stability of the software. However, there is no good solution for it.

Gadget Inspector[12] is a tool based on static taint analysis. It uses a very efficient method of symbolic execution, which generates call graphs efficiently. However, the lack of static analysis and search strategies makes it very prone to false positives and negatives. In practice, it is challenging to generate an effective gadget chain.

Rasheed proposed[13] a Fuzzer based on static analysis bootstrap, which guides Fuzzer to generate byte stream through the heap access path. This method relies heavily on the initial results of static analysis. The problem of too few static analysis results and the byte stream's structural variation will lead to unsatisfactory results. Although it will not produce false positives, it will still produce many false negatives.

Therefore, we propose a novel approach to discovering gadget chains. This approach follows the static analysis of Gadget Inspector to obtain the gadget chains to be verified. We still use symbolic execution to generate call graphs in this work. These call graphs can be abstracted into a collection of <caller, callee array>. The gadget Inspector produces false negatives because it uses a breadth-first search algorithm (BFS) to traverse the call graph. This BFS does not consider that multiple gadget chains may share nodes, which leads to only one of the multiple gadget chains passing through the same node will be searched. Therefore, We use a depth-first search algorithm that traverses a single node multiple times to avoid this problem.

To remove the false positives in the static analysis, We propose a matching dynamic verification mechanism

QuASoQ 2022: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual
* Corresponding author
Redomichelan@stu.ouc.edu.cn (Z. Lai); Quhaipeng@ouc.edu.cn (H. Qu); yinglingyun@qianxin.com (L. Ying)
0000-0002-3402-1438 (Z. Lai)

We propose a matching dynamic verification mechanism based on the Java serialization protocol and reflection mechanisms. This dynamic verification can dynamically generate the object corresponding to each gadget chain. This method, called GCGM(Gadget Core Growth Method), collates a class list based on the gadget chain and generates an object bottom-up according to the gadget core. The implementation of GCGM relies on the Java reflection mechanism, which can get all methods and properties based on a class name. The objects dynamically generated by GCGM will be serialized and injected with malicious behaviour to generate a byte stream, which the deserialization portal can verify *readObject()*.

Our contributions are as follows:

- We have improved the search algorithm of Gadget Inspector to reduce false negatives.
- We propose a dynamic verification to remove false positives based on Java serialization protocol and reflection.

2. Related Work

This section will introduce the gadget chain in Java deserialization vulnerability and some detection efforts. These efforts can be divided into two types, fingerprinting-based detection, and active discovery detection.

2.1. Gadget Chain

The Java deserialization vulnerability and the first gadget chain were discovered and proposed by Chris Frohoff[14]. In the Java deserialization vulnerability, the gadget chain is the method call chain from the deserialization entry method *readObject()* to the command execution method *exec()*[12].

Java deserialization recovers a byte stream into a Java object[3]. This complex refactoring process may trigger some magic methods. A magic method will call another method, even another magic method. For example, *HashMap.put()* is a magic method because it can automatically call *HashMap.hash()*. The successive calls of these magic methods form a gadget chain. The byte stream determines the gadget chain. An attacker can construct a byte stream to control the direction of the gadget chain to trigger some specific methods to achieve remote code execution.

2.2. Fingerprinting-based Detection

Since many security researchers have discovered many gadget chains manually, fingerprinting-based detection implemented by integrating these gadget chains is an efficient detection method.

Ysoserial[14] is a detection program that integrates a large number of gadget chains. It can quickly generate payloads for specific Java libraries to detect Java deserialization vulnerabilities. Marshalsec[15] is a tool similar to Ysoserial, which supports a broader range of libraries but cannot discover gadget chains. The Java Deserialization Scanner[16] can confirm the effectiveness of this strategy. It is a plug-in for the well-known penetration testing tool Burp Suite. It can use Ysoserial to generate payloads for penetration testing of targets for deserialization vulnerabilities.

However, fingerprinting-based detection can only detect the presence of known gadget chains in the program, but not unknown gadget chains in the program.

2.3. Active Discovery Detection

Discovering unknown gadget chains in a program is suitable for software security.

Haken's proposed Gadget Inspector[12] in 2017 is the first to enable the active discovery of gadget chains.

Its implementation relies on the following two key steps:

1. Generate passthrough dataflow and passthrough callgraph using symbolic execution.
2. Search gadget chains in the passthrough callgraph by BFS(Breadth First Search).

Gadget Inspector is an effective tool because it discovers some new gadget chains in the evaluation. However, it produces many false positives and false negatives. False positives are because it is a static analysis tool that does not generate results in the actual deserialization process. False negatives are because its search algorithm does not consider the possibility of multiple gadget chains having common nodes.

In 2020, Rasheed[13] proposed a hybrid analysis strategy to avoid false positives. It uses static analysis results as a guide for fuzzing. The advantage is that it does not make false positives because it will execute a trampoline method and observe if the dynamic sink method is triggered. To get more results, it used fuzzing to mutate the byte stream. However, the byte stream of Java serialization is highly structured, which makes fuzzing challenging to perform effectively. This strategy of hybrid analysis provides new ideas for gadget chain discovery, but from its evaluation, it makes a lot of false negatives.

3. Propose Approach

This section proposes a new active discovery detection strategy to reduce false positives and negatives. It is implemented in two steps: static analysis and dynamic verification.

Firstly, the static analysis makes many gadget chains. This step is similar to Gadget Inspector, but we optimize the search algorithm to get as many gadget chains as possible to reduce false negatives. The dynamic verification dynamically generates the corresponding byte stream based on each gadget chain. These byte streams trigger the detector during deserialization, while false positives cannot complete this process.

3.1. Gadget Core

Our work relies on the fact that multiple gadget chains in the target program may have common vital nodes. We call such vital nodes *gadget core.* A gadget chain can be abstracted as *source->gadget core->sink.* The subchain *gadget core->sink* is called the *core chain.* The subchain *source->gadget core* is called the *edge chain.*

In a target program, gadget core is always sparse, *core chain* is always unique. So we simplify the discovery of the gadget chain to the discovery of the subchain *edge chain.*

Figure 1 shows a gadget chain of the ACC(Apache Common Collection) library. With the introduction of the gadget core, the search for gadget chain can be simplified to the *edge chain*(HashSet.readObject()->LazyMap.get()), whitch saves a lot of costs.

3.2. Static Analysis

Static analysis is a technique for fast white-box testing. Generally, it includes static tainted analyses and static symbolic execution. Static analysis techniques are commonly used in method call chain searches[17]. The symbolic execution algorithm used by Gadget Inspector[12] is good enough, and we rely on it for our work.

Our static analysis is divided into the following steps:

1. Obtain the class information and method information of the target program.
2. Generate call graph by symbolic execution.
3. Search all the *edge chain.*

In the first stage, all the class information, method information, and inheritance relationships of the target class will be obtained. This work will be implemented by ASM library[18], an excellent Java byte stream manipulation tool.

After that, we use the symbolic execution of the Gadget Inspector to obtain the call relationship for every method. These call relationships make up the call graph. This call graph is stored as a collection of <caller, callee array>.

In the last step, we want the search algorithm to discover all the *edge chain.* Figure 2 shows a typical call graph with four gadget chains. Gadget Inspector can only find two chains because it can only visit **E** and **F** once.

Therefore, we propose a DFS(Depth First Search).

This DFS has two key parameters. One is the MTV(maximum time of visits) per node, and the other is the MCL(maximum chain length) in the search. The MTV setting allows DFS to search as many gadget chains as possible by visiting a node multiple times in a search. MCL limits the search depth, preventing DFS from searching too long and meaningless gadget chains. It backtracks when loops are encountered, when chain lengths exceed limits and when the visit times to a node exceed the limit. The DFS keeps a temporary chain in the search, saves the temporary chain to the result, and backtracks when the search reaches the sink method (gadget core).

This strategy ensures we get as many results as possible without timeouts or memory overflows.

3.3. Dynamic Verification

In this work, we dynamically generate byte streams and deserialize them to verify availability based on each gadget chain. Our work relies on the Java serialization protocol[19] and the Java reflection[20].

3.3.1. Gadget Core Growth Method

Java reflection allows the program to load the **Class** object based on a class name[20]. The **Class** object contains the metadata of the class, including all the methods and properties. This mechanism and the proposal of the gadget core led to the design of GCGM (Gadget Core Growth Method).

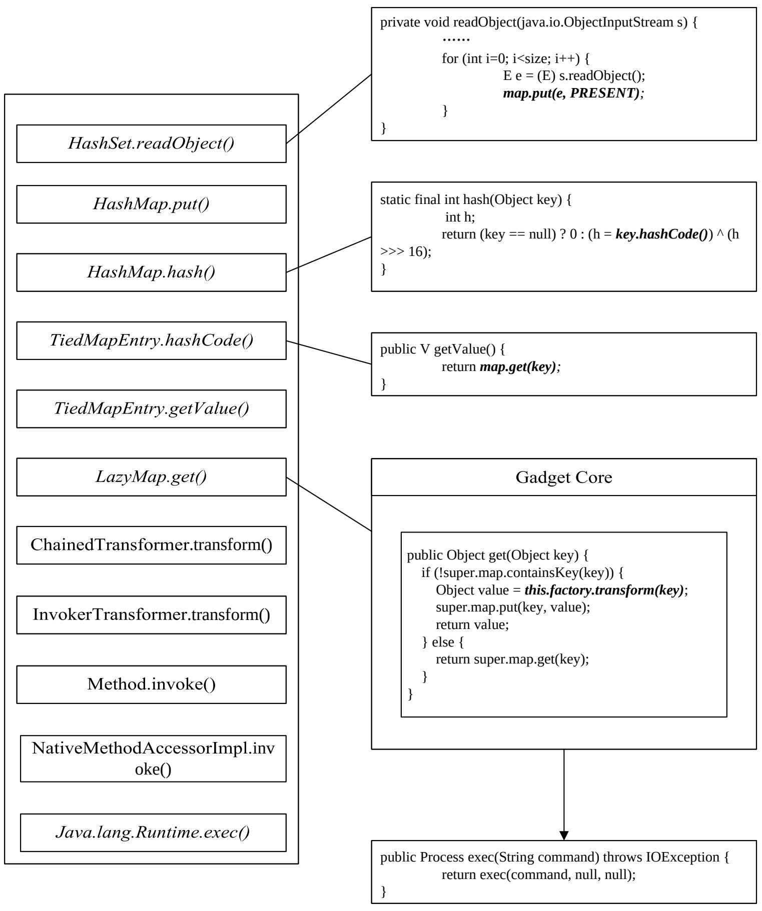
The GCGM will generate an object for each *edge chain* based on the static analysis results, which works as follows:

1. Get the manually constructed gadget core object as **current object** according to the gadget chain, which does not contain malicious behaviour.
2. Get the class name of the node above **current object** in the gadget chain as **clazz**.
3. Use reflection to get the constructor of **clazz**.
4. Call the constructor of **clazz** with the **current object** as an argument to construct a **new object**.
5. Set **new object** to **current object**.
6. Repeat steps 2-5.

The pseudocode for this method is shown in Algorithm1:

3.3.2. Deserialization Verification

The GCGM allows us to generate objects dynamically based on an *edge chain.* The ultimate goal of dynamic



```
private void readObject(java.io.ObjectInputStream s) {
        ......
        for (int i=0; i<size; i++) {
                E e = (E) s.readObject();
                map.put(e, PRESENT);
        }
}
```

```
static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h
>>> 16);
}
```

```
public V getValue() {
        return map.get(key);
}
```

```
public Object get(Object key) {
    if (!super.map.containsKey(key)) {
        Object value = this.factory.transform(key);
        super.map.put(key, value);
        return value;
    } else {
        return super.map.get(key);
    }
}
```

```
public Process exec(String command) throws IOException {
        return exec(command, null, null);
}
```

Figure 1: A classic gadget chain in ACC. *LazyMap.get()* could be a gadget core.

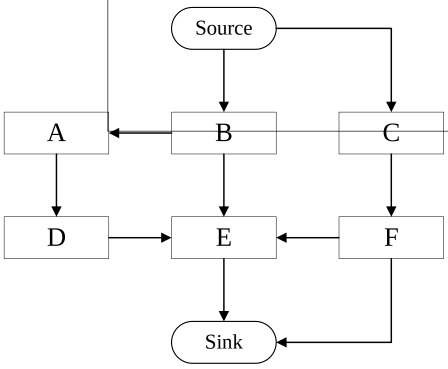


Figure 2: A typical call graph.

verification is to determine whether the corresponding byte stream of this object can trigger the sink method during the deserialization process. We design a unique verification method based on the serialization protocol in this work. This work is based on the highly structured nature of serialized byte stream.

Figure 3 shows the comparison of two serialized byte streams. The left of the image shows the byte stream corresponding to a gadget chain, and the right shows the byte stream corresponding to the corresponding *edge chain*. The difference is the byte stream corresponding to the *core chain*. This allows us to modify the byte stream to add incomplete *edge chain* to the gadget chain.

Figure 4 illustrates our workflow, which has the following steps.

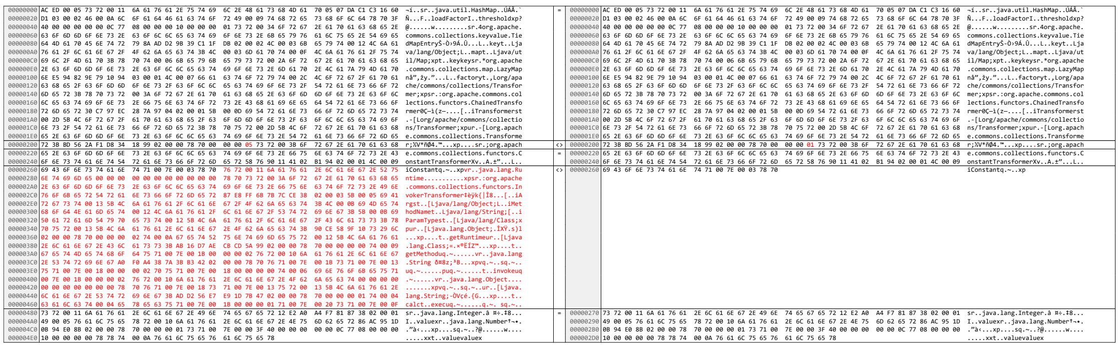
1. Get the results of static analysis, preprocessing each *edge chain*.
2. Pass the *edge chain* into GCGM as the parameter to generate the corresponding *edge object*.

Algorithm 1 Gadget Core Growth Mehtod

Input: x_0: an *edge chain* from static analysis; $Core$: a core object of gadget chain parser, which contains information about a gadget core.

Return: an object generated by GCGM;

```
 1: set $CurrentObject$ = Core.getCoreObject($x_0$);
 2: set $ClassList$ = Core.getClassList($x_0$);
 3: for $i$ = 0; $i$<$ClassList$.length();i++ do
 4:     $CurrentClassName$ = $ClassList[i]$;
 5:     $Constructor$ = $Reflection$.getConstrctor($CurrentClassName$);
 6:     $Fields$ = $Reflection$.getFilds($CurrentClassName$);
 7:     $Types$ = Reflection.getClass($CurrentObject$) + Reflection.getInterface($CurrentObject$)
 8:     for $Field$ in $Filds$ do
 9:         for $Type$ in $Types$ do
10:             if $Filed$=$Type$ then
11:                 $NewObject$ = $Constructor$.newInstance();
12:                 set $NewObject.Field$ = $CurrentObject$;
13:                 $CurrentObject$ = $NewObject$;
14:             end if
15:         end for
16:     end for
17: end forreturn $CurrentObject$;
```



```
00000000 AC ED 00 05 73 72 00 11  6A 61 76 61 2E 75 74 69  6C 2E 48 61 73 68 4D 61  70 05 07 DA C1 C3 16 60  ¬í..sr..java.util.HashMap..ÚÁÃ.`
00000020 D1 03 00 02 46 00 0A 6C  6F 61 64 46 61 63 74 6F  72 49 00 09 74 68 72 65  73 68 6F 6C 64 78 70 3F  Ñ...F..loadFactorI..thresholdxp?
00000040 40 00 00 00 00 00 0C 77  08 00 00 00 10 00 00 00  01 73 72 00 34 6F 72 67  2E 61 70 61 63 68 65 2E  @......w.........sr.4org.apache.
00000060 63 6F 6D 6D 6F 6E 73 2E  63 6F 6C 6C 65 63 74 69  6F 6E 73 2E 6B 65 79 76  61 6C 75 65 2E 54 69 65  commons.collections.keyvalue.Tie
00000080 64 4D 61 70 45 6E 74 72  79 8A AD D2 9B 39 C1 1F  DB 02 00 02 4C 00 03 6B  65 79 74 00 12 4C 6A 61  dMapEntryŠ-Ò›9Á.Û...L..keyt..Lja
000000A0 76 61 2F 6C 61 6E 67 2F  4F 62 6A 65 63 74 3B 4C  00 03 6D 61 70 74 00 0F  4C 6A 61 76 61 2F 75 74  va/lang/Object;L..mapt..Ljava/ut
000000C0 69 6C 2F 4D 61 70 3B 78  70 74 00 06 6B 65 79 6B  65 79 73 72 00 2A 6F 72  67 2E 61 70 61 63 68 65  il/Map;xpt..keykeysr.*org.apache
000000E0 2E 63 6F 6D 6D 6F 6E 73  2E 63 6F 6C 6C 65 63 74  69 6F 6E 73 2E 6D 61 70  2E 4C 61 7A 79 4D 61 70  .commons.collections.map.LazyMap
00000100 6E E5 94 82 9E 79 10 94  03 00 01 4C 00 07 66 61  63 74 6F 72 79 74 00 2C  4C 6F 72 67 2F 61 70 61  nå”‚žy.”...L..factoryt.,Lorg/apa
00000120 63 68 65 2F 63 6F 6D 6D  6F 6E 73 2F 63 6F 6C 6C  65 63 74 69 6F 6E 73 2F  54 72 61 6E 73 66 6F 72  che/commons/collections/Transfor
00000140 6D 65 72 3B 78 70 73 72  00 3A 6F 72 67 2E 61 70  61 63 68 65 2E 63 6F 6D  6D 6F 6E 73 2E 63 6F 6C  mer;xpsr.:org.apache.commons.col
00000160 6C 65 63 74 69 6F 6E 73  2E 66 75 6E 63 74 6F 72  73 2E 43 68 61 69 6E 65  64 54 72 61 6E 73 66 6F  lections.functors.ChainedTransfo
00000180 72 6D 65 72 30 C7 97 EC  28 7A 97 04 02 00 01 5B  00 0D 69 54 72 61 6E 73  66 6F 72 6D 65 72 73 74  rmer0Ç—ì(z—....[..iTransformerst
000001A0 00 2D 5B 4C 6F 72 67 2F  61 70 61 63 68 65 2F 63  6F 6D 6D 6F 6E 73 2F 63  6F 6C 6C 65 63 74 69 6F  .-[Lorg/apache/commons/collectio
000001C0 6E 73 2F 54 72 61 6E 73  66 6F 72 6D 65 72 3B 78  70 75 72 00 2D 5B 4C 6F  72 67 2E 61 70 61 63 68  ns/Transformer;xpur.-[Lorg.apach
000001E0 65 2E 63 6F 6D 6D 6F 6E  73 2E 63 6F 6C 6C 65 63  74 69 6F 6E 73 2E 54 72  61 6E 73 66 6F 72 6D 65  e.commons.collections.Transforme
00000200 72 3B BD 56 2A F1 D8 34  18 99 02 00 00 78 70 00  00 00 05 73 72 00 3B 6F  72 67 2E 61 70 61 63 68  r;½V*ñØ4.™...xp....sr.;org.apach
00000220 65 2E 63 6F 6D 6D 6F 6E  73 2E 63 6F 6C 6C 65 63  74 69 6F 6E 73 2E 66 75  6E 63 74 6F 72 73 2E 43  e.commons.collections.functors.C
00000240 6F 6E 73 74 61 6E 74 54  72 61 6E 73 66 6F 72 6D  65 72 58 76 90 11 41 02  B1 94 02 00 01 4C 00 09  onstantTransformerXv..A.±”...L..
00000260 69 43 6F 6E 73 74 61 6E  74 71 00 7E 00 03 78 70  76 72 00 11 6A 61 76 61  2E 6C 61 6E 67 2E 52 75  iConstantq.~..xpvr..java.lang.Ru
00000280 6E 74 69 6D 65 00 00 00  00 00 00 00 00 00 00 00  78 70 73 72 00 3A 6F 72  67 2E 61 70 61 63 68 65  ntime...........xpsr.:org.apache
000002A0 2E 63 6F 6D 6D 6F 6E 73  2E 63 6F 6C 6C 65 63 74  69 6F 6E 73 2E 66 75 6E  63 74 6F 72 73 2E 49 6E  .commons.collections.functors.In
000002C0 76 6F 6B 65 72 54 72 61  6E 73 66 6F 72 6D 65 72  87 E8 FF 6B 7B 7C CE 38  02 00 03 5B 00 05 69 41  vokerTransformer‡èÿk{|Î8...[..iA
000002E0 72 67 73 74 00 13 5B 4C  6A 61 76 61 2F 6C 61 6E  67 2F 4F 62 6A 65 63 74  3B 4C 00 0B 69 4D 65 74  rgst..[Ljava/lang/Object;L..iMet
00000300 68 6F 64 4E 61 6D 65 74  00 12 4C 6A 61 76 61 2F  6C 61 6E 67 2F 53 74 72  69 6E 67 3B 5B 00 0B 69  hodNamet..Ljava/lang/String;[..i
00000320 50 61 72 61 6D 54 79 70  65 73 74 00 12 5B 4C 6A  61 76 61 2F 6C 61 6E 67  2F 43 6C 61 73 73 3B 78  ParamTypest..[Ljava/lang/Class;x
00000340 70 75 72 00 13 5B 4C 6A  61 76 61 2E 6C 61 6E 67  2E 4F 62 6A 65 63 74 3B  90 CE 58 9F 10 73 29 6C  pur..[Ljava.lang.Object;.ÎXŸ.s)l
00000360 02 00 00 78 70 00 00 00  02 74 00 0A 67 65 74 52  75 6E 74 69 6D 65 75 72  00 12 5B 4C 6A 61 76 61  ...xp....t..getRuntimeur..[Ljava
00000380 2E 6C 61 6E 67 2E 43 6C  61 73 73 3B AB 16 D7 AE  CB CD 5A 99 02 00 00 78  70 00 00 00 00 74 00 09  .lang.Class;«.×®ËÍZ™...xp....t..
000003A0 67 65 74 4D 65 74 68 6F  64 75 71 00 7E 00 1B 00  00 00 02 76 72 00 10 6A  61 76 61 2E 6C 61 6E 67  getMethoduq.~......vr..java.lang
000003C0 2E 53 74 72 69 6E 67 A0  F0 A4 38 7A 3B B3 42 02  00 00 78 70 76 71 00 7E  00 1B 73 71 00 7E 00 13  .String ð¤8z;³B...xpvq.~..sq.~..
000003E0 75 71 00 7E 00 18 00 00  00 02 70 75 71 00 7E 00  18 00 00 00 00 74 00 06  69 6E 76 6F 6B 65 75 71  uq.~......puq.~......t..invokeuq
00000400 00 7E 00 1B 00 00 00 02  76 72 00 10 6A 61 76 61  2E 6C 61 6E 67 2E 4F 62  6A 65 63 74 00 00 00 00  .~......vr..java.lang.Object....
00000420 00 00 00 00 00 00 00 78  70 76 71 00 7E 00 18 73  71 00 7E 00 13 75 72 00  13 5B 4C 6A 61 76 61 2E  .......xpvq.~..sq.~..ur..[Ljava.
00000440 6C 61 6E 67 2E 53 74 72  69 6E 67 3B AD D2 56 E7  E9 1D 7B 47 02 00 00 78  70 00 00 00 01 74 00 04  lang.String;-ÒVçé.{G...xp....t..
00000460 63 61 6C 63 74 00 04 65  78 65 63 75 71 00 7E 00  1B 00 00 00 01 71 00 7E  00 20 73 71 00 7E 00 0F  calct..execuq.~......q.~. sq.~..
00000480 73 72 00 11 6A 61 76 61  2E 6C 61 6E 67 2E 49 6E  74 65 67 65 72 12 E2 A0  A4 F7 81 87 38 02 00 01  sr..java.lang.Integer.â ¤÷.‡8...
000004A0 49 00 05 76 61 6C 75 65  78 72 00 10 6A 61 76 61  2E 6C 61 6E 67 2E 4E 75  6D 62 65 72 86 AC 95 1D  I..valuexr..java.lang.Number†¬•.
000004C0 0B 94 E0 8B 02 00 00 78  70 00 00 00 01 73 71 00  7E 00 00 3F 40 00 00 00  00 00 0C 77 08 00 00 00  .”à‹...xp....sq.~..?@......w....
000004E0 10 00 00 00 00 78 78 74  00 0A 76 61 6C 75 65 76  61 6C 75 65 78                                      .....xxt..valuevaluex
```

```
=  00000000 AC ED 00 05 73 72 00 11  6A 61 76 61 2E 75 74 69  6C 2E 48 61 73 68 4D 61  70 05 07 DA C1 C3 16 60  ¬í..sr..java.util.HashMap..ÚÁÃ.`
   00000020 D1 03 00 02 46 00 0A 6C  6F 61 64 46 61 63 74 6F  72 49 00 09 74 68 72 65  73 68 6F 6C 64 78 70 3F  Ñ...F..loadFactorI..thresholdxp?
   00000040 40 00 00 00 00 00 0C 77  08 00 00 00 10 00 00 00  01 73 72 00 34 6F 72 67  2E 61 70 61 63 68 65 2E  @......w.........sr.4org.apache.
   00000060 63 6F 6D 6D 6F 6E 73 2E  63 6F 6C 6C 65 63 74 69  6F 6E 73 2E 6B 65 79 76  61 6C 75 65 2E 54 69 65  commons.collections.keyvalue.Tie
   00000080 64 4D 61 70 45 6E 74 72  79 8A AD D2 9B 39 C1 1F  DB 02 00 02 4C 00 03 6B  65 79 74 00 12 4C 6A 61  dMapEntryŠ-Ò›9Á.Û...L..keyt..Lja
   000000A0 76 61 2F 6C 61 6E 67 2F  4F 62 6A 65 63 74 3B 4C  00 03 6D 61 70 74 00 0F  4C 6A 61 76 61 2F 75 74  va/lang/Object;L..mapt..Ljava/ut
   000000C0 69 6C 2F 4D 61 70 3B 78  70 74 00 06 6B 65 79 6B  65 79 73 72 00 2A 6F 72  67 2E 61 70 61 63 68 65  il/Map;xpt..keykeysr.*org.apache
   000000E0 2E 63 6F 6D 6D 6F 6E 73  2E 63 6F 6C 6C 65 63 74  69 6F 6E 73 2E 6D 61 70  2E 4C 61 7A 79 4D 61 70  .commons.collections.map.LazyMap
   00000100 6E E5 94 82 9E 79 10 94  03 00 01 4C 00 07 66 61  63 74 6F 72 79 74 00 2C  4C 6F 72 67 2F 61 70 61  nå”‚žy.”...L..factoryt.,Lorg/apa
   00000120 63 68 65 2F 63 6F 6D 6D  6F 6E 73 2F 63 6F 6C 6C  65 63 74 69 6F 6E 73 2F  54 72 61 6E 73 66 6F 72  che/commons/collections/Transfor
   00000140 6D 65 72 3B 78 70 73 72  00 3A 6F 72 67 2E 61 70  61 63 68 65 2E 63 6F 6D  6D 6F 6E 73 2E 63 6F 6C  mer;xpsr.:org.apache.commons.col
   00000160 6C 65 63 74 69 6F 6E 73  2E 66 75 6E 63 74 6F 72  73 2E 43 68 61 69 6E 65  64 54 72 61 6E 73 66 6F  lections.functors.ChainedTransfo
   00000180 72 6D 65 72 30 C7 97 EC  28 7A 97 04 02 00 01 5B  00 0D 69 54 72 61 6E 73  66 6F 72 6D 65 72 73 74  rmer0Ç—ì(z—....[..iTransformerst
   000001A0 00 2D 5B 4C 6F 72 67 2F  61 70 61 63 68 65 2F 63  6F 6D 6D 6F 6E 73 2F 63  6F 6C 6C 65 63 74 69 6F  .-[Lorg/apache/commons/collectio
   000001C0 6E 73 2F 54 72 61 6E 73  66 6F 72 6D 65 72 3B 78  70 75 72 00 2D 5B 4C 6F  72 67 2E 61 70 61 63 68  ns/Transformer;xpur.-[Lorg.apach
   000001E0 65 2E 63 6F 6D 6D 6F 6E  73 2E 63 6F 6C 6C 65 63  74 69 6F 6E 73 2E 54 72  61 6E 73 66 6F 72 6D 65  e.commons.collections.Transforme
<> 00000200 72 3B BD 56 2A F1 D8 34  18 99 02 00 00 78 70 00  00 00 01 73 72 00 3B 6F  72 67 2E 61 70 61 63 68  r;½V*ñØ4.™...xp....sr.;org.apach
=  00000220 65 2E 63 6F 6D 6D 6F 6E  73 2E 63 6F 6C 6C 65 63  74 69 6F 6E 73 2E 66 75  6E 63 74 6F 72 73 2E 43  e.commons.collections.functors.C
   00000240 6F 6E 73 74 61 6E 74 54  72 61 6E 73 66 6F 72 6D  65 72 58 76 90 11 41 02  B1 94 02 00 01 4C 00 09  onstantTransformerXv..A.±”...L..
<> 00000260 69 43 6F 6E 73 74 61 6E  74 71 00 7E 00 03 78 70                                                    iConstantq.~..xp
=  00000270 73 72 00 11 6A 61 76 61  2E 6C 61 6E 67 2E 49 6E  74 65 67 65 72 12 E2 A0  A4 F7 81 87 38 02 00 01  sr..java.lang.Integer.â ¤÷.‡8...
   00000290 49 00 05 76 61 6C 75 65  78 72 00 10 6A 61 76 61  2E 6C 61 6E 67 2E 4E 75  6D 62 65 72 86 AC 95 1D  I..valuexr..java.lang.Number†¬•.
   000002B0 0B 94 E0 8B 02 00 00 78  70 00 00 00 01 73 71 00  7E 00 00 3F 40 00 00 00  00 00 0C 77 08 00 00 00  .”à‹...xp....sq.~..?@......w....
   000002D0 10 00 00 00 00 78 78 74  00 0A 76 61 6C 75 65 76  61 6C 75 65 78                                      .....xxt..valuevaluex
```

Figure 3: A comparison between the serialized byte stream of two similar objects.

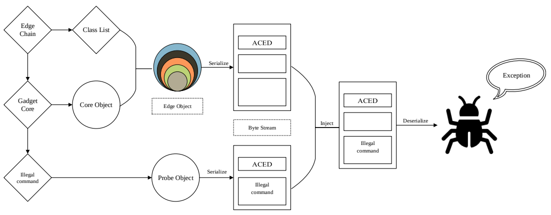


Figure 4: The Process in Dynamic Validation.

3. Generate the *probe object* by passing the *core chain* as an argument. This *probe object* contains an **illegal command** that cannot be executed by Runtime.exec().
4. Serialize the *edge object* and the *probe object* into byte stream.
5. Inject the **illegal command** of the *probe object* into the byte stream of the *edge object* by modifying the byte stream.
6. The acquired byte stream is deserialized, the **illegal command** will be triggered, and the detector will catch a specific exception.

4. Evaluate

To evaluate the effectiveness of our method, we designed feasibility, inefficiency, comparison, and versatility ex-

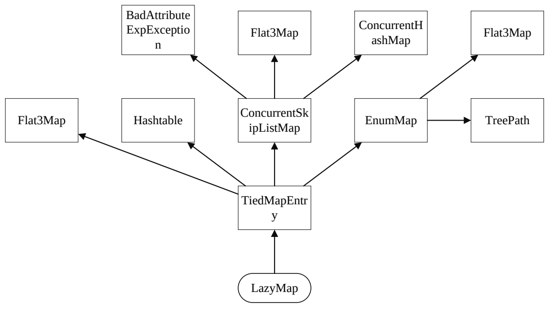


Figure 5: A Growth Tree in ACC,including 7 gadget chains

periments.

Experimental Environment: The experiments were implemented on an Intel(R) Core(TM) i3-10100 CPU @ 3.60GHz with 16GB of RAM on Windows 10.21H1. Gadget Inspector(DFS) and Gadget Catcher were run in Java 8 (release JDK 8u302).

Experimental Setup: In the feasibility experiment, we will execute the method in the test set and determine the availability of the technique based on the results. In the inefficiency experiment, we will observe the impact of two critical parameters of the method on the efficiency and results by modifying these two parameters. In the comparison experiment, we will compare the discovery of our tool with some other methods mentioned so far. In the versatility experiment, we will make discoveries on some other libraries.

Test Set: The following libraries will be used for the test set in this section.

- Apache Commons Collections 3.1
- Commons Beanutils 1.92
- Apache Commons Collections4 4.0
- Jython Standalone 2.5.2.

4.1. Results and Discussion

Feasibility Experiment: LazyMap will be set as a gadget core in ACC. With default settings, a total of 52 gadget chains were discovered. Our results not only hit all three chains in *ysoserial* that are suitable for the JDK version of *CommonsCollections5*, *CommonsCollections6*, and *CommonsCollections7* but also found many other gadget chains, which fully verified the correctness of our strategy. A growth tree is made with some discovered results in Figure 5, the root node is the gadget core, and the page node is the outer class capable of triggering the *readObject()* methods.

Efficiency Experiment: In this experiment, we verify MTV(the maximum times of visits to a node) to optimize default settings.

Table 1
The Result of Efficiency Evaluation Experiment

MTV	Static Analysis		Dynamic Verification	
	Time Cost(s)	Result	Time Cost(s)	Result
50000	862	365724	3673	52
10000	386	74844	125	28
5000	183	37626	58	24
1000	87	7567	8	13
500	51	3816	<1	11
Null	> 3d	/	/	/

Table 2
The Result of Camparion Experiment

Discovery Strategy	Results	Valid Results	Time Cost(min)
Gadget Inspector	4	0	<2
Hybrid Analysis	1	1	<20
Composite Discover	52	52	>60

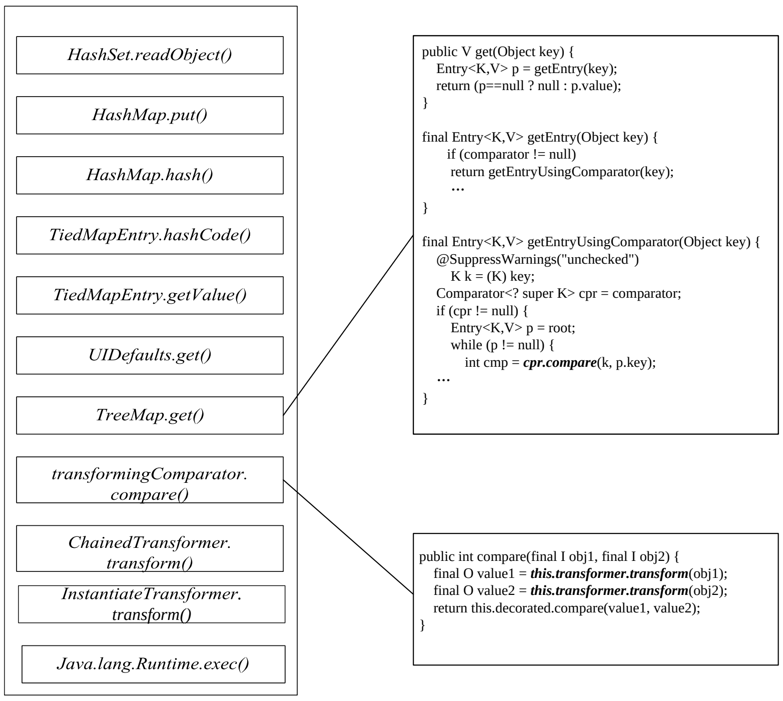
Table 1 shows the results of the efficiency experiments. This result shows that the number of experimental results and the time cost are positively related to MTV. At an MTV of 50,000, it is possible to obtain over 360,000 results in static analysis and 52 gadget chains after dynamic calibration, when the time spent is about 1 hour. The rule that can be summarized is that when MTV is set more significant, more results can be obtained, but the time overhead is also more; when MTV is infinite, the results cannot be obtained in the expected time.

Comparison Experiment:

In this experiment, we use the ACC library as the test set. Table 2 shows the three strategies' search results and the valid results, Where the results of Gadget Inspector are from our experiments. The experimental results of the hybrid analysis strategy are from the original article. It shows the effectiveness of our discovery strategy over the other two discovery strategies.

In addition, we also performed a simple runtime comparison. Gadget Inspector, a static analysis tool, was able to produce results in two minutes, the hybrid analysis approach was able to get results in 19 minutes, while our method took more than 1h when the MTV was set to 50,000.

Versatility Experiments: We are also trying to use our strategy for gadget chain discovery for some other libraries. In this experiment, we found a new gadget chain in ACC4 by combining the static analysis results shown in Figure 6. This new gadget chain is not currently included in *ysesorial* and can be judged as a new gadget chain. In addition, the *TransformingComparator()* method in this gadget chain can be used as a new gadget core to implement the discovery of other types of gadget chains. In our experiments, we initially verified the feasibility



```
public V get(Object key) {
   Entry<K,V> p = getEntry(key);
   return (p==null ? null : p.value);
}

final Entry<K,V> getEntry(Object key) {
     if (comparator != null)
      return getEntryUsingComparator(key);
      ...
}

final Entry<K,V> getEntryUsingComparator(Object key) {
   @SuppressWarnings("unchecked")
      K k = (K) key;
   Comparator<? super K> cpr = comparator;
   if (cpr != null) {
      Entry<K,V> p = root;
      while (p != null) {
         int cmp = cpr.compare(k, p.key);
   ...
}
```

```
public int compare(final I obj1, final I obj2) {
   final O value1 = this.transformer.transform(obj1);
   final O value2 = this.transformer.transform(obj2);
   return this.decorated.compare(value1, value2);
}
```

Figure 6: A new gadget chain with a new path to trigger gadget core.

Table 3
General experiments

library	release	gadget core
commons-beanutils	1.9.2	BeanComparator
commons-collections4	4.0	TransformingComparator
jython-standalone	2.5.2	Comparator

experiments of the three libraries in Table 3 and proved that this gadget core could be applied to discover gadget chains.

5. Conclusion

Based on the previous work, we have completed our analysis strategy. This strategy overcomes the common false positives and negatives in gadget chain discovery. The experimental results also prove the correctness and efficiency of our design. We also have a massive advantage in comparing with other strategies.

On the other hand, this strategy also has limitations that rely on manual analysis. Finding a suitable gadget core and building its validation strategy is necessary before analyzing a new library.

References

[1] T. Greanier, Discover the secrets of the java serialization api, 2021. URL: https://www.oracle.com/technical-resources/articles/java/serializationapi.html.
[2] Docs.Oracle.com, Trail: Rmi (the java™ tutorials), 2020. URL: https://docs.oracle.com/javase/tutorial/rmi/.
[3] Docs.Oracle.com, Objectinputstream (java platform se 7), 2011. URL: https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html.
[4] M. Daconta, When runtime.exec() won't, 2000. URL: https://www.ikkisoft.com/stuff/Defending_against_Java_Deserialization_Vulnerabilities.pdf.
[5] J. Forshaw, Are you my type?, 2012. URL: https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf.
[6] B. Stephen, What do weblogic, websphere, jboss, jenkins, opennms, and your application have in common? this vulnerability., 2015. URL: https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability.
[7] C. D. 2021, Cve-2021-36981(vulnerability in sernet

verinice 1.22.2), 2021. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-36981.

[8] C. D. 2021, Cve-2021-35464(vulnerability in forgerock am server 7.0), 2021. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3546.

[9] L. Carettoni, Defending against java deserialization vulnerabilities, 2016. URL: https://www.ikkisoft.com/stuff/Defending_against_Java_Deserialization_Vulnerabilities.pdf.

[10] S. Cristalli, E. Vignati, D. Bruschi, A. Lanzi, Trusted execution path for protecting java applications against deserialization of untrusted data, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2018, pp. 445–464.

[11] N. Koutroumpouchos, G. Lavdanis, E. Veroni, C. Ntantogian, C. Xenakis, Objectmap: Detecting insecure object deserialization, in: Proceedings of the 23rd Pan-Hellenic Conference on Informatics, 2019, pp. 67–72.

[12] I. Haken, Automated discovery of deserialization gadget chains, Proceedings of the Black Hat USA (2018).

[13] S. Rasheed, J. Dietrich, A hybrid analysis to detect java serialisation vulnerabilities, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1209–1213.

[14] C. Frohoff, G. Lawrence, ysoserial (a proof-of-concept tool for generating payloads that exploit unsafe java object deserialization.), 2015. URL: https://github.com/frohoff/ysoserial.

[15] M. Bechler, marshalsec, 2017. URL: https://github.com/mbechler/marshalsec.

[16] F. Dotta, Reliable discovery and exploitation of java deserialization vulnerabilities, 2017. URL: https://techblog.mediaservice.net/2017/05/reliable-discovery-and-exploitation-of-java-deserialization-vulnerabilities.

[17] Y. Li, T. Tan, Y. Zhang, J. Xue, Program tailoring: Slicing by sequential criteria, in: 30th European Conference on Object-Oriented Programming (ECOOP 2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[18] E. Bruneton, R. Lenglet, T. Coupaye, Asm: a code manipulation tool to implement adaptable systems, Adaptable and extensible component systems 30 (2002).

[19] Docs.oracle.com, Java serialization protocol, 2014. URL: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html.

[20] Docs.oracle.com, Java reflection api, 2014. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.

Exploring the Impact of Code Style in Identifying Good Programmers

Rafed Muhammad Yasir[1], Dr. Ahmedul Kabir[1]

[1]*Institute of Information Technology (IIT), University of Dhaka, Dhaka, Bangladesh*

Abstract
Code style is an aesthetic choice exhibited in source code that reflects programmers' individual coding habits. This study is the first to investigate whether code style can be used as an indicator to identify good programmers. Data from Google Code Jam was chosen for conducting the study. A cluster analysis was performed to find whether a particular coding style could be associated with good programmers. Furthermore, supervised machine learning models were trained using stylistic features and evaluated using recall, macro-F1, AUC-ROC and balanced accuracy to predict good programmers. The results demonstrate that good programmers may be identified using supervised machine learning models, despite that no particular style groups could be attributed as a good style.

Keywords
code style, identify good programmer, stylistic features

1. Introduction

Code style represents the physical layout of code (e.g., indentation, bracket placement), which reflects an individual's personal programming habits that do not affect its functionality [1]. Figure 1 shows two code snippets that are functionally similar but written in two different styles. Code style has an impact on various aspects of software engineering, including software maintenance [2] and speed of software development [3]. However, no prior studies have been conducted to see whether good programmers can be detected by looking at their coding style. This paper investigates the potential for using code style to identify good programmers.

```
int sum(int a, int b){
    return a+b;
}
```

(a) Style 1

```
int sum(int a, int b)
{
  return a+b;
}
```

(b) Style 2

Figure 1: Two functionally same code snippets written in different styles

Establishing a link between code style and good programmers can have several implications. Many software repositories contain style guidelines that are used to enforce a specific code style in order to maintain software quality [4]. However, these style guides are often opinionated and arbitrary [5, 6]. If a specific code style exhibited by programmers can be identified as a good style, it can be used to create non-arbitrary style guidelines for better software maintenance.

In the software industry, the developers hired by a company directly affect the quality of the codebase that they maintain. During recruitment, the candidates who apply for jobs often have to solve a set of programming problems. However, existing hiring practices do not account for the possibility that a skilled programmer could have a bad day and fail to answer a question correctly. Thus, in some circumstances a judgment may be unfair. If positive stylistic features can be identified in a programmer's code, they can be used as an additional criterion to enhance recruitment processes. This study is an initial attempt to determine whether such relationships between competent programmers and their code style can be established.

To conduct the study, the solutions collected from Google Code Jam (GCJ) [7] were used as the dataset. 30 stylistic metrics were extracted from the codes and used as features for analysis. Two methods of analysis were used. At first, clustering algorithms were applied to the data to discover style groups and check whether good programmers belonged to a particular style group. Secondly, supervised machine learning models were trained using stylistic features to predict good programmers. The models were evaluated using recall, macro-F1, area under curve of ROC (AUC-ROC), and balanced accuracy.

Results show that, although style groupings were found, there were no specific groups with which good programmers could be associated. However, supervised machine learning models showed that good programmers can be predicted to some extent. Based on the evaluated metrics, a Balanced Random Forest achieved the best re-

QuASoQ'22: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual
bsse0733@iit.du.ac.bd (R. M. Yasir); kabir@iit.du.ac.bd (Dr. A. Kabir)
https://rafed.github.io/ (R. M. Yasir)

sult with an average of 0.65 recall, 0.51 macro-F1, and 0.69 AUC-ROC.

2. Related Work

To the best of our knowledge, this is the first time code style has been used to identify good programmers. Early research conducted by Oman and Cook proposed a taxonomy for code styles to help people grasp a coherent view on the basis and application of code styles [8]. The four major categories of their taxonomy are general practices, typographic style, control structure style and information structure style. They also concluded in further research that code style is more than cosmetic and that it can affect areas such as code comprehension [9].

Caliskan et al. proposed a Code Stylometry Feature Set (CSFS) with which they performed source code authorship attribution [10]. Their feature set is language agnostic and can be used for other programming languages as well. With their method, they achieved 94% accuracy in classifying 1600 authors and 98% accuracy in classifying 250 authors. They concluded that this method can help in the identification of authors of malicious programs, ghostwriting detection, software forensics and copyright investigation.

Mirza and Cosma explored the suitability of using code style in detecting plagiarism in the BlackBox dataset [11]. BlackBox is a project that collects data from users of the BlueJ which is a Java IDE [11]. Their study showed that code style is suitable for detecting plagiarism.

For evaluating software projects, Zou et al. explored how code style inconsistency can affect pull request integration in projects on Github [3]. By analyzing 117 public repositories, they concluded that code styles with specific criteria can influence both the acceptance of pull requests and the time required to merge a pull request.

Mi and Yu conducted a study on stylistic inconsistency in software projects [2]. They proposed a collection of stylistic metrics for C++ projects and used these metrics to analyze small-scale Github projects. By using hierarchical agglomerative clustering they showed that stylistic differences exist between source files in a project. They concluded that, using the degree of stylistic inconsistency as a basis, code comprehensibility and software maintainability could be improved in the future.

Several tools have been developed that can check stylistic inconsistencies and help programmers improve code style. Ala-Mutka et al. developed *style++* that helps students learn good C++ programming conventions [12]. Mäkelä et al. developed *Japroach* that checks whether Java programs have a particular style and if style related issues exist in them [13]. Ogura et al. developed *style-coordinator* to decrease inconsistency and improve code readability [1].

Table 1
Number of Participants in a Round

	2015	2016	2017
Qualification round	10744	11401	11342
Round 2	1650	1641	1824
Round 3	266	296	286
World Finals	22	20	21

3. Methodology

3.1. Dataset Description

The dataset for the study was made up of the solutions gathered from the Google Code Jam (GCJ) website [7]. GCJ is an annual programming contest held by Google. GCJ is selected because its data is publicly available and it can somewhat resemble programming exams in recruitment processes. Professional programmers, students and amateurs from all around the world participate in GCJ. Therefore, not only does the dataset consist of source code from varying sources, but they also solve the same problem which makes comparative study possible. The contest consists of seven rounds, each progressively harder than the previous. The rounds are: Qualification round, Round 1A, Round 1B, Round 1C, Round 2, Round 3 and World Finals. We consider the programmers who reached at least Round 3 as "good" programmers because participating in this round requires passing the previous rounds with a large number of accepted solutions.

Although GCJ accepts solutions in many programming languages, C++ was selected as the preferred language for evaluation as it is more prevalent among participants and has the highest number of submissions. Each problem of the contest has two validation sets: a small input set and a large input set. A solution for the large validation set is a valid solution for the small input set, but not vice versa. For our analysis, the solutions from the small input were taken as it had more submissions and it would also be redundant if both solutions were taken. A small number of solutions were rejected as the language encoding consisted of non-standard characters.

The solutions to the contests held in 2015, 2016, and 2017 are chosen for experimentation. However, we only include solutions from Qualification Round, Round 2 and Round 3 for our dataset. Round 1A, Round 1B, and Round 1C are excluded because participating in these rounds are optional and thus lacks submissions from all programmers. Solutions from the World Finals are excluded because the number of finalists is too small to take into consideration for analysis. Table 1 shows the number of participants in each round of the contests.

From the collected data, layout and lexical stylistic features were extracted based on [10]. Abstract syntax tree based features were omitted as these features are

Table 2
Feature Description of Dataset

Feature	Definition
numTabs/length	Number of tabs divided by file length in characters
numSpaces/length	Number of space characters divided by file length in characters
numEmptyLines/length	Number of empty lines divided by file length in characters
whiteSpaceRatio	Ratio between the number of whitespace characters (spaces, tabs, and newlines) and non-whitespace characters
newLineBeforeOpenBrace	Ratio between the number of code blocks preceded by a newline character and not preceded by a newline character
tabsLeadLines	Ratio between the number of lines preceded by a tab and not preceded by a tab
avgLineLength	Average length of each line
stdDevLineLength	Standard deviation of the lengths of each line
numkeyword/length	Number of occurrences of keyword divided by file length in characters, where keyword is one of if, else, else-if, for, while, do, break, continue, switch, case (10 different features)
numTernary/length	Number of ternary operators divided by file length in characters
numTokens/length	Number of word tokens divided by file length in characters
numUniqueTokens/length	Number of unique keywords used divided by file length in characters
numComments/length	Number of comments divided by file length in characters
numLineComments/length	Number of line comments divided by file length in characters
numBlockComments/length	Number of comments divided by file length in characters
numLiterals/length	Number of string, character, and numeric literals divided by file length in characters
numMacros/length	Number of preprocessor directives divided by file length in characters
nestingDepth	Highest depth of control statements and loops
numFunctions/length	Number of functions divided by file length in characters
avgParams	Average number of parameters of functions
stdDevNumParams	Standard deviation of the number of parameters of functions

not within the control of a programmer. Furthermore, term frequency based features were also excluded as they largely depend on the corpus being used. Following these criteria, 30 stylistic features were extracted. The features are listed in Table 2.

3.2. Approach

This section discusses the setups for exploring the effects of code style on classifying good programmers. Two methods were used for this purpose: (1) clustering techniques and (2) supervised machine learning algorithms.

3.2.1. Analyzing Using Clustering Techniques

Clustering is a method of partitioning objects into homogeneous groups on the basis of similarity among those objects [14]. t-SNE is one such algorithm that can discover the potential number of clusters in a dataset with high dimensions [15]. For each problem in the contests, t-SNE graphs were plotted with the intent of finding groups that conform to a particular style. Each data point in the plots represents a solution submitted by a programmer. The data points are labeled as:

- Red: reached World Finals
- Green: reached Round 3
- Light blue: other programmers

The plots provide an estimate for the number of clusters and the distribution of good programmers in the clusters.

To further validate the clustering provided by t-SNE, Hierarchical Agglomerative Clustering (HAC) with Ward linkage was performed and dendrograms were plotted. HAC is a clustering algorithm that treats every data point as a cluster and they are gradually merged to form a single cluster [16]. The number of clusters indicated by the dendrograms was matched with the number of clusters indicated by t-SNE before further analysis was performed.

To analyze the properties of the t-SNE clusters, solutions to each problem in the dataset were clustered using K-Means With K=number of clusters estimated by t-SNE. The solutions in the data were then labeled based on the cluster they belonged to. This labeled data was fitted to a Random Forest Classifier to obtain the feature importance of the tree. Based on the tree's feature importance, it was determined what style groups exist and whether good programmers belong to a specific style group.

3.2.2. Analyzing using Supervised Machine Learning Algorithms

Supervised learning is a method of training a model that can make predictions based on labeled data [17]. For predicting good programmers, the following models were trained: Logistic Regression (LR), Support Vector Classifier (SVC), K-Nearest Neighbors (KNN), Decision Tree (DT) and Random Forest (RF). A Dummy classifier was also trained to act as a performance baseline for comparison [18]. The models were trained for each problem in the dataset. Table 1 shows that the number of participants in Round 3 is far less than the participants in the previous rounds. That is, the proportion of "good" programmers in the dataset is much lower in comparison to the other programmers. This makes the classification an imbalanced classification problem [19]. To balance the training data, the up-sampling technique SMOTE [20] was used prior to training the above mentioned models. Furthermore, Balanced Random Forest (BRF) and RUS Adaboost classifier (RUSAda) were also trained which performs under-sampling to balance training data [21]. For bias-free results, all trained models were K-fold cross-validated.

4. Experimental Analysis

4.1. Performance Evaluation

The clusters created for analysis were evaluated empirically. Although the analyzed dataset had labels and the results could be evaluated using a metric, this was not done, as evaluating clustering algorithms using labels is not recommended [22].

For the supervised algorithms recall, macro-F1 and Area Under Curve of ROC (AUC-ROC) were used to evaluate the models. Recall is the measure of the fraction of good programmers correctly identified as good programmers [23]. Recall is calculated as equation (1). F1 is an evaluation metric measured by combining precision and recall, and it is calculated as (3) [23]. Macro-F1 is the arithmetic mean of the per class F1 scores. It has been selected as an evaluation criterion because the training data was imbalanced, and macro-F1 is a good metric for imbalanced data [24]. AUC-ROC is the area under a ROC curve that allows comparison between models [23]. Apart from these evaluation metrics, balanced accuracy was also reported. Balanced accuracy is defined as the average of recall obtained on each class [25].

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative} \tag{1}$$

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive} \tag{2}$$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \tag{3}$$

Table 3
Feature importance of Clusters

Features	Importance
newLineBeforeOpenBrace	0.282
tabsLeadLines	0.163
numTabs/length	0.151
numSpaces/length	0.103

4.2. Results and Discussion

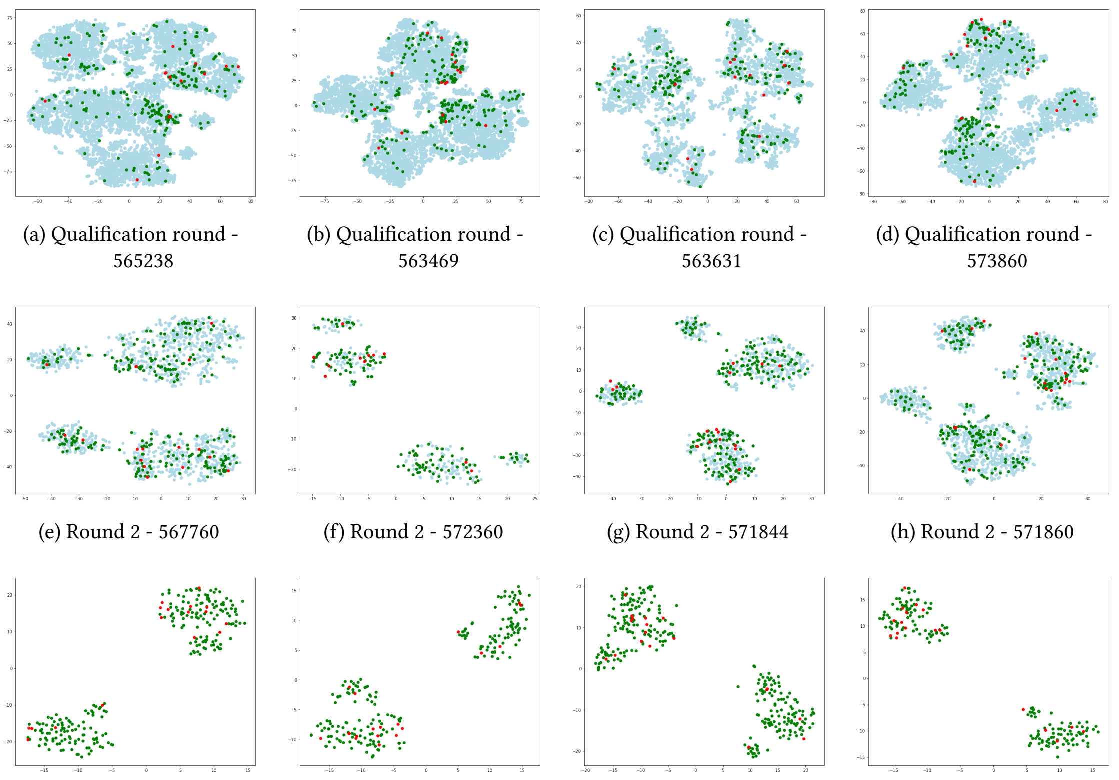
After analyzing the contest results from 2015, 2016, and 2017, it was discovered that they were similar. Therefore, only the results of one year (2016) are shown in this section.

Figure 2 shows the t-SNE clusters of all the problems in the dataset of the year 2016. The caption of each image shows the round and the problem number. From the graphs, it can be said that 4 stylistic clusters exist for each solution. The most important features of the clusters determined by a Random Forest Classifier are shown in Table 3. The importance of all other features was less than 0.05. It is seen that *newLineBeforeOpenBrace* and *tabsLeadLines* are the most prominent features in separating the clusters. A manual inspection of the codes also proved the findings to be true. The discovered clusters are formed around the following feature combinations:

- new line before opening braces, tabs lead lines
- no new line before opening braces, tabs lead lines
- new line before opening braces, whitespace lead lines
- no new line before opening braces, whitespace lead lines

Although style clusters were found, the good programmers were almost equally distributed among them. As a result, we cannot conclude that good programmers belong to a specific cluster.

The results of the supervised machine learning models are shown in Table 4. BRF, LR, SVC and RUSAda performed better than the dummy model, which indicates that some patterns can be identified by the models that can be used to predict good programmers. Also, BRF outperformed all models in terms of Recall, macro-F1 and AUC-ROC. While different studies have used code style for various aspects such as author identification and plagiarism detection, none of the studies have dealt with good programmer identification. Therefore, we cannot compare our results with those of existing studies. However, our results can inspire further research on the relationship between code style and the coding ability of programmers.



(a) Qualification round - 565238 (b) Qualification round - 563469 (c) Qualification round - 563631 (d) Qualification round - 573860

(e) Round 2 - 567760 (f) Round 2 - 572360 (g) Round 2 - 571844 (h) Round 2 - 571860

(i) Round 3 - 574081 (j) Round 3 - 563236 (k) Round 3 - 563461 (l) Round 3 - 512508

Figure 2: t-SNE Style Clusters of GCJ 2016

Table 4
Prediction Results of Supervised Learning Models

Model	Recall	macro-F1	AUC-ROC	Balanced Accuracy
BRF	0.650	0.511	0.695	0.645
LR	0.641	0.523	0.692	0.651
SVC	0.601	0.523	0.689	0.639
RUSAda	0.510	0.50	0.626	0.590
Dummy	0.485	0.412	0.499	0.489
KNN	0.469	0.494	0.593	0.565
DT	0.287	0.525	0.542	0.542
RF	0.185	0.539	0.664	0.537

5. Threats to Validity

This section presents aspects that may threaten the validity of the study:

- **Internal validity:** The result of our analysis largely depends on the stylistic features that were used. Using other stylistic features may affect the results. However, many existing studies [10, 26] have used these features for their analysis, so they can be relied upon.
- **External validity:** The analysis was done on the source files of the GCJ dataset. Therefore, the findings of this study may not be generally applicable to contests in other formats. Furthermore, as only C++ codes were selected for analysis, it cannot be said whether stylistic features of other programming languages will show similar results. Additionally, the criteria for defining a good programmer are subjective and could be defined in other ways depending on the context. In such contexts, our results can not be generalized.

To ensure the reliability of the study, the analysis results are made publicly available in Jupyter notebooks at github.com/rafed/GcjStyleAnalysis.

6. Conclusion

This paper explores whether code style can be used to identify good programmers. The study was conducted on C++ solutions from the Google Code Jam contest. Clustering techniques such as t-SNE and hierarchical agglomerative clustering were used to discover whether style clusters exist and if good programmers could be attributed to any of them. Although four style clusters were found, good programmers could not be associated with a particular cluster. However, supervised machine learning showed that stylistic attributes can be used to predict good programmers. Seven machine learning models were trained and evaluated using recall, macro-F1 and AUC-ROC. A Balanced Random Forest yielded the best results with 0.650 recall, 0.511 macro-F1 and 0.695 AUC-ROC. The results indicate that code style can be used as a measure to identify good programmers.

Future research will examine if defining style guidelines based on the coding style of skilled programmers enhances the quality of software. Additionally, it is possible to investigate how the current recruitment procedures might be efficiently linked with the prediction of good programmers utilizing code style. There is also potential for improving our results using other techniques.

References

[1] N. Ogura, S. Matsumoto, H. Hata, S. Kusumoto, Bring your own coding style, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, 2018, pp. 527–531.

[2] Q. Mi, J. Keung, Y. Yu, Measuring the stylistic inconsistency in software projects using hierarchical agglomerative clustering, in: Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering, ACM, 2016, p. 5.

[3] W. Zou, J. Xuan, X. Xie, Z. Chen, B. Xu, How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects, Empirical Software Engineering (2019) 1–33.

[4] T. Erkkinen, Model style guidelines for production code generation, Technical Report, SAE Technical Paper, 2005.

[5] Pullrequest.com, 2022. URL: https://www.pullrequest.com/blog/create-a-programming-style-guide/.

[6] Google style guides, 2022. URL: https://google.github.io/styleguide/.

[7] Google, Past contests, google code jam, 2022. URL: https://code.google.com/codejam/past-contests.

[8] P. W. Oman, C. R. Cook, A programming style taxonomy, Journal of Systems and Software 15 (1991) 287–301.

[9] P. W. Oman, C. R. Cook, Typographic style is more than cosmetic, Communications of the ACM 33 (1990) 506–520.

[10] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, R. Greenstadt, De-anonymizing programmers via code stylometry, in: 24th {USENIX} Security Symposium ({USENIX} Security 15), 2015, pp. 255–270.

[11] O. M. Mirza, M. Joy, G. Cosma, Style analysis for source code plagiarism detection—an analysis of a dataset of student coursework, in: 2017 IEEE 17th international conference on advanced learning technologies (ICALT), IEEE, 2017, pp. 296–297.

[12] K. Ala-Mutka, T. Uimonen, H.-M. Jarvinen, Supporting students in c++ programming courses with automatic program style assessment, Journal of Information Technology Education: Research 3 (2004) 245–262.

[13] S. Mäkelä, V. Leppänen, Japroch: A tool for checking programming style, Kolin Kolistelut—Koli Calling 2004 (2004) 151.

[14] S. C. Johnson, Hierarchical clustering schemes, Psychometrika 32 (1967) 241–254.

[15] G. C. Linderman, S. Steinerberger, Clustering with t-sne, provably, SIAM Journal on Mathematics of Data Science 1 (2019) 313–332.

[16] K. Sasirekha, P. Baby, Agglomerative hierarchical clustering algorithm-a, International Journal of Scientific and Research Publications 83 (2013) 83.

[17] S. J. Russell, P. Norvig, Artificial intelligence: a modern approach, Malaysia; Pearson Education Limited„ 2016.

[18] Scikit-learn, Dummy classsifier, 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.dummy.DummyClassifier.html.

[19] N. Japkowicz, S. Stephen, The class imbalance problem: A systematic study, Intelligent data analysis 6 (2002) 429–449.

[20] N. V. Chawla, K. W. Bowyer, L. O. Hall, W. P. Kegelmeyer, Smote: synthetic minority oversampling technique, Journal of artificial intelligence research 16 (2002) 321–357.

[21] Imbalanced-learn, Ensembled methods, 2022. URL: https://imbalanced-learn.readthedocs.io/en/stable/api.html#module-imblearn.ensemble.

[22] I. Färber, S. Günnemann, H.-P. Kriegel, P. Kröger, E. Müller, E. Schubert, T. Seidl, A. Zimek, On using class-labels in evaluation of clusterings, in: MultiClust: 1st international workshop on discovering, summarizing and using multiple clusterings held in conjunction with KDD, 2010, p. 1.

[23] D. M. Powers, Evaluation: from precision, recall and f-measure to roc, informedness, markedness

and correlation (2011).
[24] B. Wu, S. Lyu, B. Ghanem, Constrained submodular minimization for missing labels and class imbalance in multi-label learning, in: Thirtieth AAAI Conference on Artificial Intelligence, 2016.
[25] Balanced accuracy, 2022. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.balanced_accuracy_score.html.
[26] M. Tereszkowski-Kaminski, S. Pastrana, J. Blasco, G. Suarez-Tangil, et al., Towards improving code stylometry analysis in underground forums, in: Proceedings on Privacy Enhancing Technologies (PETS), 2022.

An Empirical Evaluation of Defect Prediction Models Using Project-Specific Measures

Umamaheswara Sharma B[1,*,†], Ravichandra Sadam[2,†]

[1]Research Scholar, Department of Computer Science and Engineering, National Institute of Technology, Warangal, Telangana, India
[2]Associate Professor, Department of Computer Science and Engineering, National Institute of Technology, Warangal, Telangana, India

Abstract
Due to the advantages of economizing the testing resources such as cost, time, and consequently the manpower on the developing software project, research on software defect prediction (SDP) has gained traction in academia. Though many works in the literature discuss constraints that are limiting the final prediction performance, finding the essential benefits in terms of the project objectives such as cost, service time, and failure is rarely explored. On the basis of these project objectives, the gap of finding the best performing SDP model is still present in the literature. In this regard, in this work, a detailed empirical analysis of With-in Project Defect Prediction (WPDP), Cross-Project Defect Prediction (CPDP), and *mixed*-Cross-Project Defect Prediction (M-CPDP) models is provided using the project-specific performance measures such as percent of perfect cleans (PPC), percent of non-perfect cleans (PNPC), false omission rate (FOR), and its additional derived performance measures, which are proposed by Sharma et al. in [1]. The empirical analysis is provided on 14 publicly available datasets collected from the PROMISE repository using the baselines such as support vector machines (SVM), decision trees (DT), and k-nearest neighbours (k-NN) classifiers. From the empirical results, we observe that the M-CPDP model is significantly better at providing maximum savings in the allocated budget, minimum service time, and minimum failure incidents on the majority of the target projects.

Keywords
With-in Project Defect Prediction, Mixed Cross-Project Defect Prediction, Cross-Project Defect Prediction, project-specific Performance Measures, Prediction Quality Assessment

1. Introduction

Software defect prediction (SDP) models reduce the work load on the tester by providing the status of defect-proneness of the newly developed software module in a short time [2, 3, 4, 5, 6, 7, 8, 9, 10]. Hence, it reduces the total cost, time, and manpower that are spent on the target project [11, 1]. Because of these advantages, there are the works in the literature that address various constraints in building prediction models [2, 7, 12, 13, 14, 15, 16, 17, 18, 19, 20].

The literature on SDP classifies works on its major categories such as within-project defect prediction (WPDP) and cross-project defect prediction (CPDP). WPDP models use available local data from the same software to train the prediction model, whereas CPDP models use defects data collected from multiple projects to train the prediction model[21]. The CPDP models are further classified into many types, such as *mixed* CPDP (M-CPDP), mixed project defect prediction (MPDP), and *pair-wise*-CPDP [21]. In *mixed* CPDP, old versions of the same project along with the data of the other source projects are used to train the prediction model. Hence, the M-CPDP models are considered as a mixture of with-in and cross-project defect prediction contexts. In MPDP, the prediction is trained using the data from the target project along with the old versions of the same project and the data of the other source projects. Whereas in the *pair-wise* CPDP, the prediction models will be built using each project's data. Then, the mean or median of the performances of these pair-wise predictions is then used to calculate the final performance of the *pair-wise* CPDP model. Among all the variants of SDP, most literature discusses works on WPDP, CPDP, and M-CPDP [1, 10, 21] models.

The common objective of developing SDP models is to reduce the project cost that is being spent on the testing team, reduce the work load on the tester, and minimise the risk of observing the failures [1, 22]. Owing to these objectives, recently, a work by Sharma et al. in [1] discusses various novel project-specific performance measures such as percent of perfect cleans (PPC), percent of non-perfect cleans (PNPC), false-omission rate (FOR), percent of saved budget (PSB), and percent of remaining edits (PRE), to capture real-benefits from the prediction model. These performance measures are essential in understanding the main objectives of the prediction model but are also used to evaluate the developed prediction model.

Further extending the work of Sharma et al. [1], in

QuASoQ 2022: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual
*Corresponding author
†These authors contributed equally in this research.
uma.phd@student.nitw.ac.in (U. S. B); ravic@nitw.ac.in (R. Sadam)
0000-0003-1676-3347 (U. S. B)

this work, we provide an empirical analysis of the performances of the widely used SDP variants such as WPDP, CPDP, and M-CPDP models using the project-specific performance measures. For the empirical analysis, we have used benchmark classification approaches such as support vector machines (SVM), decision trees (DT), and k-nearest neighbours (k-NN), for each problem context. From the empirical analysis, we observe that the M-CPDP model is performing significantly better over the others in terms of the project-specific attributes.

1.1. Contributions

Capturing the project benefits from the SDP model and obtaining the best performing model is essential for any research practitioner. Hence, this work is mainly targeted at providing an empirical analysis of the SDP models such as WPDP, CPDP, and M-CPDP with regard to the project-specific performance measures such as percent of perfect cleans (PPC), percent of non-perfect cleans (PNPC), false-omission rate (FOR), percent of saved budget (PSB), and percent of remaining edits (PRE), to capture real-benefits from the best prediction model. Since interpreting the obtained results of the SDP model in terms of the project objectives is a necessary task, any research practitioner should have to evaluate their models using the project-specific measures as in [1]. To the best of our knowledge, conducting an empirical analysis between the prediction models such as WPDP, CPDP, and M-CPDP in order to know the best performing model in terms of the SDP objectives is new to this field of research.

Paper Organisation: The rest of the paper is organised as follows: Section 2 discusses the related work on the usage of the performance measures in SDP studies. In Section 3, we provide a complete list of project-specific performance measures, proposed by Sharma et al. in [1]. Section 4 presents the essential requirements for conducting empirical analysis such as utilised datasets, significance test and base-line machine learning (ML) models. In Section 5, we provide detailed empirical results in terms of the proposed measures and discuss the important observations. Section 6 discusses potential threats to the obtained prediction performance. Section 7 concludes the work and provides possible future work.

2. Related Work

In this section, we provide the papers that discuss the key findings from the SDP models in terms of the traditional measures. In this regard, we present the abstract details of two relevant studies that discuss the suitable performance measures for the SDP models. In addition, we also discuss the project-specific performance measures in evaluating the prediction models as suggested in [1].

Since model performance comparison received more attention, in [23], Jiang et al. discussed various traditional performance measures were investigated to find the most suitable candidate for the defect prediction tasks. The study analyses the strengths and weaknesses of the wide variety of numeric evaluation measures such as overall accuracy, error rate, sensitivity, specificity, precision, G-mean, F-measure, J-coefficient, in addition to the graphical summarisation measures such as receiver operating characteristic (ROC) curve, precision and recall (PR) curve, cost curve, and lift chart. The empirical study was conducted using five base-lines on the NASA projects. Since optimising the cost that is being spent on the project and maximising the efficiency of the software verification are the main objectives in developing SDP models, the task for the research practitioners is to minimise the misclassification rate. In this regard, the study [23] does not qualify any best traditional performance measures (this is due to the fact that, in terms of selective performance measures, rarely will one or few models prove to be the best for all possible uses in software quality assessment). However, they concluded that the F-measure offers a balanced consideration of the observed results.

Morasca and Lavazza in [24] conducted a study on choosing the best and relevant portion of the ROC curves, obtained from the predictions of the SDP model. The study proposed a new measure called the ratio of relevant areas (RRA) for evaluating the SDP models by taking only the parts of the ROC curves corresponding to the various values of the threshold. Their work also addresses the shortcomings of the widely used performance measures such as Area Under the Curve (AUC) and the Gini coefficient. However, in summary, their approach provides a theoretical illustration for the use of traditional measures in reducing the misclassification costs of the defect proneness models.

Recently, Sharma et al. in [1] discussed the shortcomings of widely used traditional measures such as F-measure and AUC, and proposed five project-specific performance measures to capture the important observations from the prediction model in terms of the project objectives. The study suggests using an interpretable measure that provides the predictions from the SDP model in terms of cost, service time, and failure, as these are the essential objectives to be accomplished from the prediction model.

Providing an empirical analysis of the prediction models by illustrating their performances in terms of the project-specific attributes is the primary research gap in SDP research. Hence, by extending the study of Sharma et al. in [1], in this work, we provide an empirical evaluation of the widely used defect prediction variants such as WPDP, CPDP, and M-CPDP models in order to validate the use of the project-specific performance measures.

Table 1
The PROMISE projects

Project	Modules	LoC	Defects	%Defects	Project	Modules	LoC	Defects	%Defects
Ant-1.3	125	37,699	20	16.00	Camel-1.4	872	98,080	145	16.63
Ant-1.4	178	54,195	40	22.47	Camel-1.6	965	113,055	188	19.48
Ant-1.5	293	87,047	32	10.92	Jedit-3.2	272	128,883	90	33.09
Ant-1.6	351	113,246	92	26.21	Jedit-4.0	306	144,803	75	24.51
Ant-1.7	745	208,653	166	22.28	Jedit-4.1	312	153,087	79	25.32
Camel-1.0	339	33,721	13	03.83	Jedit-4.2	367	170,683	48	13.08
Camel-1.2	608	66,302	216	35.53	Jedit-4.3	492	202,363	11	02.24

Table 2
The confusion matrix

		Actual values	
		Defective	Clean
Predicted values	Defective	TP	FP
	Clean	FN	TN

3. Project-Specific Measures

In this section, we present the details of the project-specific performance measures, such as percent of perfect cleans, percent of non-perfect cleans, false-omission rate, percent of saved budget, and percent of remaining edits.

The measures PPC and PSB interpret the predictions of the SDP model in terms of cost units, while the PNPC and PRE interpret the predictions in terms of service time units. The measure FOR is used to measure the failure chances from the misclassification of the defective modules. All the measures use the information from the confusion matrix (given in Table 2) among which the measures PPC, PNPC, PSB, and PRE measures utilise an extra attribute called lines of code (LoC) to compute the prediction performances. A detailed explanation of these measures is presented below.

1. Percent of Perfect Cleans (PPC): Since the true negatives (TN) represent the reduced work load, the measure PPC helps in deriving the percentage of reduced work load on the tester. The PPC is derived as the ratio of total TNs to total test instances.

$$\text{PPC} = \frac{|TN|}{|n_t|} \tag{1}$$

Where $|n_t|$ is the number of test instances.

2. Percent of Saved Budget (PSB): Using the measure PPC and an additional attribute called LoC, we estimate the total amount of saved budget in the developing project. The PSB is calculated as:

$$\text{PSB} = \frac{\sum_{i \in TN} SB(LoC_i)}{\sum_{i \in n_t} SB(LoC_i)} \tag{2}$$

Here, we assign a unit cost for servicing each line of code.

3. Percent of Non-Perfect Cleans (PNPC): In contrast to the measure PPC, PNPC is used to represent the percent of work load on the tester from the prediction model. This is because, except for the modules in TN, the tester has to conduct a code walk for all the remaining modules. The measure PNPC is expressed as:

$$PNPC = \frac{|n_t| - |TN|}{|n_t|} \tag{3}$$

4. Percent of Remaining Edits (PRE): Using the measure PNPC and an additional attribute called LoC, we estimate the total service time which is remaining for the tester after utilising the prediction model. Here, for each line of code, we assign a unit time to calculate the service time. This measure is defined below:

$$\text{PRE} = \frac{\sum_{i \in n_t - TN} RE(LoC_i)}{\sum_{i \in n_t} RE(LoC_i)} \tag{4}$$

5. False-Omission Rate (FOR): Unlike other measures, using the measure FOR, we calculate the total failures in the project with the use of SDP models. The major cause of the failures is when the defective module is predicted as clean. Since the total clean modules represents the combination of the false negatives and true negatives, the software may experience failure when the end user triggers a false negative module. Assuming each false negative instance can cause a single failure in the system, the percent of failure instances is measured as:

$$\text{FOR} = \frac{|FN|}{|TN| + |FN|} \tag{5}$$

Note that, the definitions of all these measure are directly taken from the work [1].

4. Study Design

In this section, we provide the details of the utilised datasets (in Section 4.1) and the base-line classifiers (in

Section 4.2). The details of the non-parametric test called Cliff's delta effect size test is provided in Section 4.3. An abstract procedure for the empirical approach is given in Section 4.4.

4.1. Utilised Defects Data

For the empirical analysis, we use publicly available 14 datasets from the PROMISE repository [25]. Each project consist of 24 metrics to describe the software module. Here, the software module can be either a class, method, or a program. We use each software metric as a feature to build the prediction model. A description of the utilised datasets is presented in Table 1.

4.2. Baseline Machine Learners

We perform empirical analysis for three SDP variants using three widely used base-line ML models: SVM, k-NN, and DT. A short description of the utilised baseline ML classifiers is given below.

SVM Classifier: We used a linear kernel function in the training process to compute some extreme data transformations for obtaining the separable data.

k-NN Classifier: The value of k is selected for the k-nearest neighbour (k-NN) model based on 10-fold cross validation. Appropriately, we have chosen k to be 11 after testing the model with various values of k.

Decision Tree Classifier: We have used a general classification and regression trees approach to build the DT classifier.

4.3. Statistical Significance Test

To observe the deviation between the the SDP models, we conduct a non-parametric test called Cliff's delta. This measure provides four levels of effectiveness of the one model over the other models. These levels are given in table 3. The larger value of Cliff's delta indicates the greater effect between the models.

Table 3
Cliff's delta effect size levels [26]

S.No	$\|\delta\|$	Effectiveness Category
1	$0.000 \leq \|\delta\| < 0.147$	Negligible
2	$0.147 \leq \|\delta\| < 0.330$	Small
3	$0.330 \leq \|\delta\| < 0.474$	Medium
4	$0.474 \leq \|\delta\| \leq 1.000$	Strong

4.4. Empirical Approach

An empirical evaluation is carried out on each variant of the SDP model. This is because each variant of SDP has different implementation criteria. However, we provide an empirical comparison of the developed models since each variant provides predictions on the same target project dataset. A general training procedure for the three tasks such as WPDP, CPDP, and M-CPDP is given below.

4.4.1. Training and Testing

The WPDP Model: Assume each software project has the availability of local data. Now, we train each base-line ML model on the released versions of the software project [10]. The modules in the latest version (target version or the target project) of the same project are then given as input to the trained WPDP model, in order to observe the predictions.

The CPDP Model: Assume each software project does not have the availability of local data. Now, we train each base-line ML model on the released software projects' defects data [1]. The modules in the newly developed software project (or the target project) are then given as input to the trained CPDP model, in order to observe the predictions.

The M-CPDP Model: Assume each software project has the availability of local data. Also, we assume that defect data for the source projects is available. Now, we train each base-line ML model on the defects data created by augmenting the data from the software project's released versions and the data from the source projects [21]. The modules in the latest version (target version or target project) of the target project are then given as input to the trained M-CPDP model, in order to observe the predictions.

4.4.2. Comparative Approach

To understand the role of project-specific measures in interpreting the performance of the best defect prediction model, we followed the below approach:

Empirical Procedure:

1. First, we use base-line classifiers such as SVM, k-NN, and DT to train the variants of the SDP models such as WPDP, CPDP, and M-CPDP on the PROMISE projects. Each model is evaluated on 10-fold cross validation to observe the mean predictions.
2. Second, we observe the average performances of the trained WPDP, CPDP, and M-CPDP using the project-specific performance measures on each target project.
3. Third, in terms of each base-line classifier, using Cliff's delta effect-size test, we compared the performances of the WPDP, CPDP, and M-CPDP using the project-specific performance measures.

Table 4
Performances of the variants of the SDP models that uses SVM as base classifier

Target Project	PPC			PSC			PNPC			PRE		
	WPDP	CPDP	M-CPDP	WPDP	CPDP	M-CPDP	WPDP	CPDP	M-CPDP	WPDP	CPDP	M-CPDP
Ant-1.3	0.6912	0.8309	0.8223	0.5632	0.6645	0.6423	0.3088	0.1691	0.1777	0.4368	0.3355	0.3577
Ant-1.4	0.6031	0.7428	0.8310	0.6174	0.7187	0.7322	0.3969	0.2572	0.1690	0.3826	0.2813	0.2678
Ant-1.5	0.6931	0.8328	0.8881	0.5652	0.6665	0.6724	0.3069	0.1672	0.1119	0.4348	0.3335	0.3276
Ant-1.6	0.5481	0.6878	0.6592	0.3719	0.4732	0.4613	0.4519	0.3122	0.3408	0.6281	0.5268	0.5387
Ant-1.7	0.5799	0.7196	0.7081	0.3567	0.4580	0.4325	0.4201	0.2804	0.2919	0.6433	0.5420	0.5676
Camel-1.0	0.8082	0.9479	0.9581	0.8163	0.9176	0.9213	0.1918	0.0521	0.0419	0.1837	0.0824	0.0787
Camel-1.2	0.5021	0.6418	0.6614	0.4610	0.5623	0.5747	0.4979	0.3582	0.3386	0.5390	0.4377	0.4253
Camel-1.4	0.7009	0.8406	0.8526	0.6170	0.7183	0.7313	0.2991	0.1594	0.1474	0.3830	0.2817	0.2687
Camel-1.6	0.6589	0.7986	0.8114	0.5921	0.6934	0.7269	0.3411	0.2014	0.1886	0.4079	0.3066	0.2731
Jedit-3.2	0.3819	0.5216	0.4614	0.1756	0.2769	0.2614	0.6181	0.4784	0.5386	0.8244	0.7231	0.7386
Jedit-4.0	0.5339	0.6736	0.5582	0.4492	0.5505	0.5132	0.4661	0.3264	0.4418	0.5508	0.4495	0.4868
Jedit-4.1	0.4826	0.6223	0.4593	0.1877	0.2890	0.2233	0.5174	0.3777	0.5407	0.8123	0.7110	0.7767
Jedit-4.2	0.5620	0.7017	0.6064	0.4619	0.5632	0.4633	0.4380	0.2983	0.3936	0.5381	0.4368	0.5367
Jedit-4.3	0.6660	0.8057	0.9047	0.6877	0.7890	0.8027	0.3340	0.1943	0.0953	0.3123	0.2110	0.1973
Average	0.6008	**0.7405**	0.7273	0.4945	**0.5958**	0.5828	0.3992	**0.2595**	0.2727	0.5055	**0.4042**	0.4172
Cliff's Delta	**0.4692**	0	-	**0.2959**	0	-	**0.4692**	0	-	**0.2959**	0	-

Table 5
Performances of the variants of the SDP models that uses k-NN as base classifier

Target Project	PPC			PSC			PNPC			PRE		
	WPDP	CPDP	M-CPDP	WPDP	CPDP	M-CPDP	WPDP	CPDP	M-CPDP	WPDP	CPDP	M-CPDP
Ant-1.3	0.5826	0.7223	0.7615	0.5632	0.6645	0.6766	0.4174	0.2777	0.2385	0.4368	0.3355	0.3234
Ant-1.4	0.5559	0.6956	0.6526	0.5611	0.6624	0.6525	0.4441	0.3044	0.3474	0.4389	0.3376	0.3475
Ant-1.5	0.6817	0.8214	0.8724	0.6911	0.7924	0.8056	0.3183	0.1786	0.1276	0.3089	0.2076	0.1944
Ant-1.6	0.5195	0.6592	0.6767	0.4633	0.5646	0.5852	0.4805	0.3408	0.3233	0.5367	0.4354	0.4148
Ant-1.7	0.5567	0.6964	0.7209	0.5226	0.6239	0.6525	0.4433	0.3036	0.2791	0.4774	0.3761	0.3475
Camel-1.0	0.7703	0.9100	0.9534	0.7205	0.8218	0.8413	0.2297	0.0900	0.0466	0.2795	0.1782	0.1587
Camel-1.2	0.4826	0.6223	0.6102	0.4763	0.5776	0.5614	0.5174	0.3777	0.3898	0.5237	0.4224	0.4386
Camel-1.4	0.7028	0.8425	0.8728	0.6300	0.7313	0.7433	0.2972	0.1575	0.1272	0.3700	0.2687	0.2567
Camel-1.6	0.6127	0.7524	0.7728	0.6021	0.7034	0.7313	0.3873	0.2476	0.2272	0.3979	0.2966	0.2687
Jedit-3.2	0.4318	0.5715	0.5544	0.4230	0.5243	0.5162	0.5682	0.4285	0.4456	0.5770	0.4757	0.4838
Jedit-4.0	0.4686	0.6083	0.6223	0.4714	0.5727	0.5785	0.5314	0.3917	0.3777	0.5286	0.4273	0.4215
Jedit-4.1	0.4661	0.6058	0.5248	0.4719	0.5732	0.5304	0.5339	0.3942	0.4752	0.5281	0.4268	0.4696
Jedit-4.2	0.5220	0.6617	0.6728	0.5219	0.6232	0.6269	0.4780	0.3383	0.3272	0.4781	0.3768	0.3731
Jedit-4.3	0.6137	0.7534	0.8052	0.6052	0.7065	0.7491	0.3863	0.2466	0.1948	0.3948	0.2935	0.2509
Average	0.5691	0.7088	**0.7195**	0.5517	0.6530	**0.6608**	0.4309	0.2912	**0.2805**	0.4483	0.3470	**0.3392**
Cliff's Delta	**0.6429**	0.0561	-	**0.5816**	0.0664	-	**0.6429**	0.0561	-	**0.58**16	0.0664	-

5. Study Results

In this section, we report the observed results of the WPDP, CPDP, and M-CPDP models in terms of the project-specific performance measures.

Tables 4, 5, and 6 provide the performances of the defect prediction models such as WPDP, CPDP, and M-CPDP on the 14 target projects in terms of the measures such as PPC, PSB, PNPC, and PRE, respectively. These tables also provide the results of the Cliff's delta effect-size test. The models such as WPDP, CPDP, and M-CPDP in Table 4 utilised the SVM as a base classifier to observe the predictions on the target datasets, whereas the models in Table 5 utilised the k-NN as a base classifier to observe the predictions on the target datasets. And, the models in Table 6 utilised the DT as a base classifier to observe the predictions on the target datasets.

From Table 4, it is observed that, in the majority of cases, the SVM-based CPDP outperformed the other models in terms of all the performance measures. In particular, the average PPC of the CPDP model has achieved a better value when compared with the other models. Therefore, the testers do not need to visit 74.05% modules to find their defect-proneness. As a consequence, on an average, the savings in the total allocated budget is more using the CPDP model when compared with the other models. Assume a total of 100% allocated budget on the project. Using the CPDP model, the project manager can save up to 59.58% of the budget, whereas with the use of the other models such as WPDP and M-CPDP, the project manager can save only 49.45% and 58.28% of the budget, respectively. On the contrary, since the PNPC is the converse of the measure PPC, the resultant measure PRE also shows its benefits using the CPDP model. Using SVM-based CPDP model, on an average, the testers will have to conduct a code walk on the 40.42% of the total written code. If the testers utilise either WPDP or M-CPDP models, respectively, they have to spend 50.55% and 41.72% of the total written code to observe the defective content. However, the Cliff's delta effect-size test indicating that there is no greater effect between the models such as CPDP and M-CPDP.

Table 6
Performances of the variants of the SDP models that uses DT as base classifier

Target Project	PPC			PSC			PNPC			PRE		
	WPDP	**CPDP**	**M-CPDP**	**WPDP**	**CPDP**	**M-CPDP**	**WPDP**	**CPDP**	**M-CPDP**	**WPDP**	**CPDP**	**M-CPDP**
Ant-1.3	0.7037	0.8434	0.8545	0.6849	0.7862	0.7966	0.2963	0.1566	0.1455	0.3151	0.2138	0.2034
Ant-1.4	0.6218	0.7615	0.8314	0.6210	0.7223	0.7313	0.3782	0.2385	0.1686	0.3790	0.2777	0.2687
Ant-1.5	0.6998	0.8395	0.8615	0.7219	0.8232	0.7913	0.3002	0.1605	0.1385	0.2781	0.1768	0.2087
Ant-1.6	0.5817	0.7214	0.7323	0.6077	0.7090	0.6594	0.4183	0.2786	0.2677	0.3923	0.2910	0.3406
Ant-1.7	0.5774	0.7171	0.7089	0.6010	0.7023	0.6947	0.4226	0.2829	0.2911	0.3990	0.2977	0.3053
Camel-1.0	0.8129	0.9526	0.9723	0.7521	0.8534	0.8615	0.1871	0.0474	0.0277	0.2479	0.1466	0.1385
Camel-1.2	0.5130	0.6527	0.6617	0.5311	0.6324	0.6080	0.4870	0.3473	0.3383	0.4689	0.3676	0.3920
Camel-1.4	0.7129	0.8526	0.8633	0.7210	0.8223	0.7966	0.2871	0.1474	0.1367	0.2790	0.1777	0.2034
Camel-1.6	0.6684	0.8081	0.8216	0.6613	0.7626	0.7717	0.3316	0.1919	0.1784	0.3387	0.2374	0.2283
Jedit-3.2	0.3936	0.5333	0.5434	0.4113	0.5126	0.5056	0.6064	0.4667	0.4566	0.5887	0.4874	0.4944
Jedit-4.0	0.5659	0.7056	0.6767	0.5657	0.6670	0.6055	0.4341	0.2944	0.3233	0.4343	0.3330	0.3945
Jedit-4.1	0.4917	0.6314	0.6467	0.5014	0.6027	0.6115	0.5083	0.3686	0.3533	0.4986	0.3973	0.3885
Jedit-4.2	0.5826	0.7223	0.7314	0.6008	0.7021	0.6966	0.4174	0.2777	0.2686	0.3992	0.2979	0.3034
Jedit-4.3	0.6826	0.8223	0.8618	0.6910	0.7923	0.7622	0.3174	0.1777	0.1382	0.3090	0.2077	0.2378
Average	0.6149	0.7546	**0.7691**	0.6194	**0.7207**	0.7066	0.3851	0.2454	**0.2309**	0.3806	**0.2793**	0.2934
Cliff's Delta	**0.6735**	0.1429	-	**0.5205**	-0.1021	-	**0.6735**	0.1429	-	**0.5205**	-0.1021	-

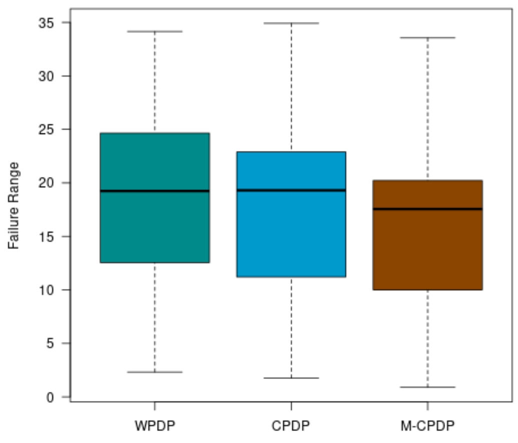


Figure 1: The box-plots representing the observed FOR values on the three models that uses SVM as base classifier

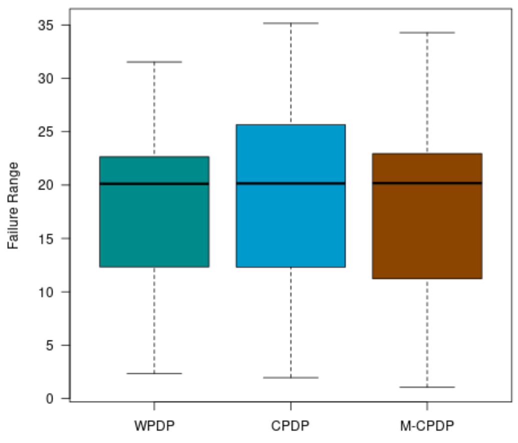


Figure 2: The box-plots representing the observed FOR values on the three models that uses k-NN as base classifier.

From Table 5, it is observed that, in the majority of cases, the k-NN-based M-CPDP outperformed the other models in terms of all the performance measures. In particular, the average PPC of the M-CPDP model has achieved a better value when compared with the other models. Hence, the testers do not need to conduct a code review on 71.95% modules to find their defect-proneness. As a consequence, on an average, the savings in the total allocated budget is more using the M-CPDP model when compared with the other models. For a total of 100% allocated budget on the project, using the M-CPDP model, the project manager can save up to 66.08% of the budget, whereas with the use of the other models such as WPDP and CPDP, the project manager can save 55.17% and 65.30% of the budget, respectively. On the other hand, using k-NN-based M-CPDP model, on an average, the testers will have to conduct a code walk only on the 33.92% of the total written code. If the testers utilise either WPDP or CPDP models, respectively, they have to spend 44.83% and 34.70% of the total written code to observe the defective content. The Cliff's delta effect-size test indicating that there is a negligible but positive effect from the M-CPDP model over the CPDP model.

From Table 6, it is observed that, in the majority of cases, the DT-based M-CPDP outperformed the other models in terms of the performance measures such as PPC and PNPC. While the model CPDP performed better than the other models in terms of measures such as

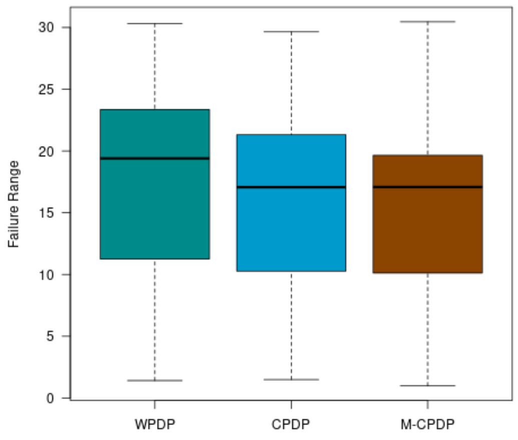


Figure 3: The box-plots representing the observed FOR values on the three models that uses DT as base classifier.

PSB and PRE. In particular, the average PPC of the M-CPDP model has achieved a better value when compared with the other models. Hence, the testers do not need to conduct a code review on 76.91% modules to find their defect-proneness. In contrast, on an average, the savings in the total allocated budget is more using CPDP model, when compared with the other models. On a total of 100% allocated budget on the project, using CPDP model, the project manager can save up to 72.07% of the budget, whereas with the use of the other models such as WPDP and M-CPDP, the project manager can save 61.94% and 70.66% of the budget, respectively. On the other hand, using the DT-based CPDP model, on an average, the testers will have to conduct a code walk only on the 27.93% of the total written code. If the testers utilise either WPDP or M-CPDP models, respectively, they have to spend 38.06% and 29.34% of the total written code to observe the defective content. The Cliff's delta effect-size test indicating that there is a negligible but positive effect from the M-CPDP model over the CPDP model in terms of PPC and PNPC measures. In terms of PSC and PRE, the Cliff's delta effect-size test indicating that there is a negligible but negative effect from the M-CPDP model over the CPDP model.

The box-plots in Figures 1, 2, and 3 represent the chances of failure incidents as a result of SVM, k-NN, and DT-based SDP models in the target projects. From Figures 1, 2, and 3, it is observed that, the median failure incidents are fewer using the M-CPDP model (which is trained using three classifiers) when compared with the other models. Since any software project should least expect a misclassifications from the prediction model, the M-CPDP model may suit well in real-time testing environments.

5.1. Discussion

Any project seeks benefits from the prediction model, hence, achieving its goal is of primary importance to the researcher. The major obstacle in selecting the model is obtaining a trade-off between the obtained performances from the various prediction models. In Section 5, we observe that, on the majority of target projects, the CPDP model has achieved its better performance in terms of PPC and PNPC. This shows that the CPDP model is good at predicting clean modules more accurately. However, surprisingly, the M-CPDP model has shown its strength in terms of budget savings, minimal service time, and more importantly, minimal failure incidents on the majority of the target projects. Hence, even though the CPDP model is better in terms of PPC and PNPC, the M-CPDP is better in terms of all the performance measures.

From Tables 4, 5, and 6, and Figures 1, 2, and 3, it is observed that, among all the baselines, after 10-fold cross validation, the decision tree-based M-CPDP model has achieved maximum budget savings, minimal service time, and minimal failure incidents on the majority of the target projects.

6. Threats to Validity

Variation in the observed performances at various working environments is common in the empirical research. In this section, we present the factors that may affect the observed performances.

Internal Validity:

The observed performances are based on the usage of a few base-line ML models, and the M-CPDP model has shown its strength using majority of the baselines. However, implementing the other baselines such as logistic regression, neural networks, ensemble models, etc. on the other widely used repositories such as NASA, AEEEM, ReLink, etc. is the major threat that may hinder the final performance of the M-CPDP model.

External Validity:

For the purpose of knowing the best model that is suitable for the real-time testing environments, we performed an empirical analysis on only three variants of SDP using the project-specific performance measures. The generalised conclusions can be made when conducting the empirical analysis on the other variants of the SDP such as MPDP, *pair-wise* CPDP, just-in time software defect prediction (JIT-SDP), and heterogeneous defect prediction (HDP).

7. Conclusion and Future Work

The research on proposing the software defect prediction (SDP) models is intended to diminish the workload on the tester by providing intelligent decisions on the defect-proneness of the newly developed software module. Hence, the objective of the SDP models is to decrease the time, cost, and manpower that are being spent on the software project. Inherently, the task of the SDP models is also to reduce the risk of misclassification (in particular, false negatives). In this regard, Sharma et al. in [1] proposed project-specific performance measures such as percent of perfect-cleans, percent of saved budget, percent of non-perfect cleans, percent of remaining edits, and false omission rate to interpret the obtained results in terms of the project objectives. Since it is important for the software engineering researcher to provide a better prediction model, it is necessary to interpret the results in terms of the project-specific objectives.

Extending the work of Sharma et al.[1], in this paper, we conducted an empirical analysis of the interpretation of the project-specific performance measures on the variants of SDP such as WPDP, CPDP, and *mixed*-CPDP. With the empirical analysis of the PROMISE projects, we conclude that the models such as CPDP and M-CPDP have achieved significantly better performances in terms of all the measures than the WPDP model. Among CPDP and M-CPDP, we observe that the number of failure incidents is lower with the use of the M-CPDP models. Also, on the majority of the target projects, the software managers may benefit from the use of M-CPDP models in terms of maximum savings in the allocated budget and minimal time required to service the code. Hence, we recommend using M-CPDP models in real-time testing environments.

Possible future research directions from this work include: (1) conducting a large-scale empirical analysis of all the variants of SDP on widely-used defect repositories such as PROMISE, NASA, AEEEM, ReLink, GitHub, etc. using the project-specific performance measures. (2) estimating the real-time feasibility of the M-CPDP model.

References

[1] U. S. B., R. Sadam, How far does the predictive decision impact the software project? the cost, service time, and failure analysis from a cross-project defect prediction model, Journal of Systems and Software 195 (2023) 111522. URL: https://www.sciencedirect.com/science/article/pii/S0164121222001984. doi:https://doi.org/10.1016/j.jss.2022.111522.

[2] V. R. Basili, L. C. Briand, W. L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE TRANSACTIONS ON SOFTWARE ENGINEERING 22 (1996).

[3] V. U. B. Challagulla, F. B. Bastani, R. Paul, Empirical Assessment of Machine Learning based Software Defect Prediction Techniques, 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (2005) 263–270.

[4] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: A proposed framework and novel findings, IEEE Transactions on Software Engineering 34 (2008) 485–496.

[5] C. Catal, B. Diri, A systematic review of software fault prediction studies, Expert Systems with Applications 36 (2009) 7346–7354.

[6] M. D'Ambros, M. Lanza, R. Robbes, Evaluating defect prediction approaches: a benchmark and an extensive comparison, Empirical Software Engineering 17 (2012) 531–577.

[7] B. Ghotra, S. McIntosh, A. E. Hassan, A large-scale study of the impact of feature selection techniques on defect classification models, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE, 2017, pp. 146–157.

[8] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, A. De Lucia, A Developer Centered Bug Prediction Model, IEEE Transactions on Software Engineering (2018).

[9] L. Kumar, S. K. Sripada, A. Sureka, S. K. Rath, Effective fault prediction model developed using least square support vector machine (lssvm), Journal of Systems and Software 137 (2018) 686–712.

[10] U. S. Bhutamapuram, R. Sadam, With-in-project defect prediction using bootstrap aggregation based diverse ensemble learning technique, Journal of King Saud University-Computer and Information Sciences (2021). URL: https://doi.org/10.1016/j.jksuci.2021.09.010.

[11] J. Xu, F. Wang, J. Ai, Defect prediction with semantics and context features of codes based on graph representation learning, IEEE Transactions on Reliability (2020).

[12] L. C. Briand, W. L. Melo, J. Wust, Assessing the applicability of fault-proneness models across object-oriented software projects, IEEE transactions on Software Engineering 28 (2002) 706–720.

[13] T. M. Khoshgoftaar, E. B. Allen, J. Deng, Using regression trees to classify fault-prone software modules, IEEE Transactions on Reliability (2002).

[14] T. Menzies, J. DiStefano, A. Orrego, R. Chapman, Assessing predictors of software defects, in: Proc. Workshop Predictive Software Models, 2004.

[15] K. O. Elish, M. O. Elish, Predicting defect-prone software modules using support vector machines, Journal of Systems and Software 81 (2008) 649–660.

[16] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, B. Murphy, Cross-project defect prediction: a large scale experiment on data vs. domain vs. process, in: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, 2009, pp. 91–100.
[17] I. H. Laradji, M. Alshayeb, L. Ghouti, Software defect prediction using ensemble learning on selected features, Information and Software Technology 58 (2015) 388–402.
[18] L. Kumar, S. Misra, S. K. Rath, An empirical analysis of the effectiveness of software metrics and fault prediction model for identifying faulty classes, Computer standards & interfaces 53 (2017) 1–32.
[19] S. Hosseini, B. Turhan, M. Mäntylä, A benchmark study on the effectiveness of search-based data selection and feature selection for cross project defect prediction, Information and Software Technology 95 (2018) 296–312.
[20] H. Chen, X.-Y. Jing, Z. Li, D. Wu, Y. Peng, Z. Huang, An empirical study on heterogeneous defect prediction approaches, IEEE Transactions on Software Engineering (2020).
[21] S. Herbold, A. Trautsch, J. Grabowski, A comparative study to benchmark cross-project defect prediction approaches, IEEE Transactions on Software Engineering 44 (2017) 811–833.
[22] C. Ni, X. Chen, F. Wu, Y. Shen, Q. Gu, An empirical study on pareto based multi-objective feature selection for software defect prediction, Journal of Systems and Software 152 (2019) 215–238.
[23] Y. Jiang, B. Cukic, Y. Ma, Techniques for evaluating fault prediction models, Empirical Software Engineering 13 (2008) 561–595.
[24] S. Morasca, L. Lavazza, On the assessment of software defect prediction models via roc curves, Empirical Software Engineering 25 (2020) 3977–4019.
[25] J. Sayyad Shirabad, T. Menzies, The PROMISE Repository of Software Engineering Databases, School of Information Technology and Engineering, University of Ottawa, Canada, 2005. URL: http://promise.site.uottawa.ca/SERepository.
[26] N. Cliff, Dominance statistics: Ordinal analyses to answer ordinal questions., Psychological bulletin 114 (1993) 494.

Towards Model Driven Safety and Security by Design

Miguel Campusano, Simon Hacks and Eun-Young Kang

SDU Software Engineering, Maersk Mc-Kinney Moller Institute, University of Southern Denmark, Odense

Abstract

Software is getting more and more complex, while it gets more and more important to make it safe and secure. At the same time, the expectations towards the software developers increase and it is unrealistic that they are able to cope properly with all safety and security requirements. To enable developers to focus on the important parts of the system, model driven software development got widely adopted. Within this work, we extend this approach by proposing an architecture, which allows to automatize the analysis of the safety and security properties of the system under design. After the analysis of the system, feedback will be provided to the developers so that they are able to reason about the design decisions that they recently made. To discuss our approach, we rely on a model driven approach for drone mission planning and envision how the different components of the architecture would need to interact.

Keywords

Model Driven Software Engineering, Automatized Analysis, Safety, Security, Model Checking

1. Introduction

The complexity of software is increasing as the problems that software solves are getting more and more difficult [1]. This is particularly true when software is used to interact with the real world, by physical interaction using cyber-physical systems, or by providing remote interfaces to enable interaction from all over the world [2]. Due to this interaction, there are new demands toward the safety and security of such safety-critical systems: systems should not harm the people using them nor neglect access to the system from an unauthorized source. However, because of the complexity of these systems, it is unreasonable to expect developers to produce bug-free, safe and secure code. For this, models play a central role, as high-level abstraction allows developers to focus on the fundamental complexity of the systems (i.e., what the system is supposed to do) instead of their incidental complexity (e.g., safety and security) [3]. This high-level abstraction allows developers to build safe and secure code by design by taking care of the incidental complexity. As it is challenging to include proper safety and security measures into a system subsequently, it is preferable to follow a safety/security by design approach while developing a software system [4]. In each development phase, the respective measures can be included instead of cumbersomely included at the end. This increases the safety and security of the system, while reducing the needed effort to introduce the needed measures.

There is a pressing need for methods and tools to verify and validate reliability of software systems, i.e., all software we build should be correct, robust, safe, and secure under certain circumstances. To prove safety and achieve error-free software systems, formal reasoning and methods are used by detecting when the system transitions into an unsafe state (i.e., one where it violates a critical safety requirement) [5, 6]. While testing can provide some reassurance that the systems being developed are bug-free, it is limited by the skills and expertise of the tester. It is not guaranteed that testing can find all errors or show their absence whereas formal verification can by employing exhaustive analysis [7]. Thus, the use of a combination of both approaches and software engineering techniques ensures potential errors are captured as early as possible. Our focus is on the use of formal methods alongside testing approaches, formal verification can be applied to establish functional correctness and can be combined with model-driven testing. This approach is integrated into a development workflow and provides correct configurations and practical considerations of design from an industrial perspective.

There are different approaches to achieve security by design for software systems [4]. One approach is to continuously perform penetration tests of the system under development [8]. However, this requires a large portion of resources to be permanently executed. Moreover, this requires already a system that can be tested. Another option is to perform attack simulations on a threat model that represents the system based on known vulnerabilities. This allows not only an easy security assessment of the actual system under development, but also it is possible to compare the security properties of different possible systems without having them already developed [9].

To enable software developers to assess the safety and security of their systems in almost real-time, it is nat-

QuASoQ 2022: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual

mica@mmmi.sdu.dk (M. Campusano); shacks@mmmi.sdu.dk (S. Hacks); eyk@mmmi.sdu.dk (E. Kang)
0000-0002-7894-6635 (M. Campusano); 0000-0003-0478-9347 (S. Hacks); 0000-0002-4589-2378 (E. Kang)

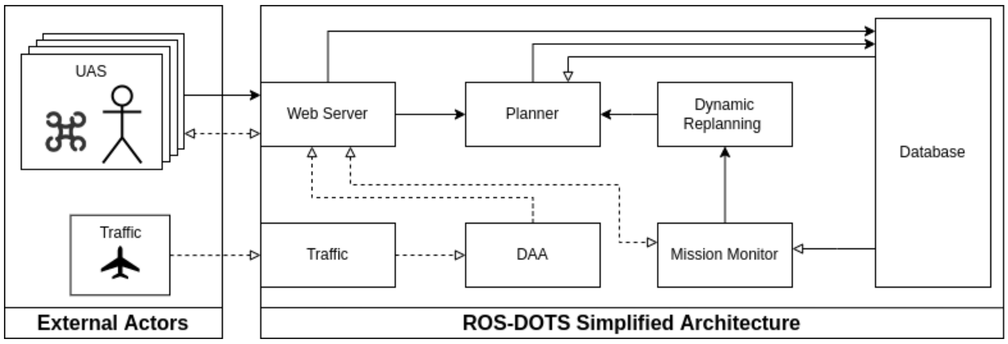


Figure 1: Simplified ROS-DOTS architecture. The different lines represent different types of interaction: black arrows are request/response calls, white arrows are callbacks, and dotted lines are constant data flow by publish/subscribe pattern.

ural to combine the beforehand presented approaches. Therefore, the models and design created by developers are not solely used for implementing and generating the system, but they are also transformed automatically into the respective representations that are needed for the assessment of the safety and security. In the background, the assessment will be executed and as soon as the results are available, the developer gets informed about potential safety or security issues in their actual software architecture thought these same models. In a second step, the developer also gets suggestions how to improve the safety and security based on the comparison of different possible evolutions of the actual architecture.

Within this work, we present the first step towards an automatized model driven safety and security assessment of software systems. To achieve this, we discuss first the background of Model Driven Software Development, Safety Assessment, and Security Assessment. Afterwards, we present our vision of achieving a model driven safety and security assessment before we discuss first insights and how we plan to continue our work.

2. Background and Related Work

2.1. Model Driven Software Development

Model Driven Software Development (MDSD) refers to using software abstractions to separate fundamental and incidental complexity of systems. These abstractions are done by models, which are the representation of the essential aspects of the system. By using them, developers can define structures and behaviors on these systems efficiently, considering the domain-specific aspects of the systems. Then, developers can use this high-level abstraction of the designed system to generate executable source code by a sequence of model transformations [10, 11].

Any modeling approach is described using metamodels [11]. Models are abstractions over similar programs while metamodels are abstractions over similar models from a particular domain. One specific way of designing systems using models is first designing a metamodel in the form of a Domain Specific Language (DSL). A DSL is a concise language that describes a solution of a particular domain. Developers can use the DSL to define programs that specify the behaviors of the different models of the system. Moreover, the domain abstractions allow the DSL to be used by developers and domain experts [12].

We have successfully used MDSD to develop Unmanned Aerial Systems (UAS), commonly refers as drones. We built a DSL for the specification of multi-UAS missions, called Drone Operation Template Specification (DOTS) [13], which uses specific languages constructions to coordinate the use of several UAS in the airspace. The DOTS programs are then loaded to a service-oriented architecture called ROS-DOTS [14] (Robot Operation System (ROS)). This architecture provides several services for UAS, in particular multi-UAS mission planning (i.e., specification of UAS flight paths to fulfill a goal), dynamic replanning of missions (i.e., flight paths modification of ongoing UAS missions), and a detect-and-avoid system (i.e., local collision alerting of UAS). Figure 1 shows a simplified version of ROS-DOTS.

2.2. Safety Assessment

Safety assessment methods include preliminary and system hazard and risk analysis, fault tree generation and analysis, failure mode and effects analysis. Despite such well-established methods provide an efficient support for safety engineers, the methods could benefit from an integration with system modeling, verification, and validation (V&V) environments. Efforts have been put into investigation of safety assessment through the MDSE based on general purpose System Modeling Language (SysML) [15], Similar studies are also conducted in other modeling language such as EAST-ADL [16, 17, 18, 19] or AADL [20] that support writing transformation rules towards formal languages to permit their analysis by formal tools. However, these languages are limited for robot design compared to our domain specific UAS and meta attack languages. Systems theory process analysis (STPA) [21, 22, 23] has been studied in the context of unifying both safety and security assessment. However, their approaches lack formalism that limits formal V&V. As far as we know, our approach is the first to combine both safety and security assessment based on MDSE, STPA, attack simulations, and formal V&V to guarantee trustworthiness in cyber-physical systems, e.g., robot, UAS, manufacturing, and IoT systems, etc.

To reasoning about and analysis of a design model it is essential that the modeling language has a well-defined (informal or formal) semantics. For cyber-physical systems (which heavily rely on the real-time aspect) a promising approach to provide analysis of models is to formally specify systems in a modeling language such as Timed Automata (TA) [24]. A TA is a finite state machine extended with clocks, where a *clock* is a variable over the positive real numbers. All clocks in a TA star at zero, grow continuously at the same rate, and can be tested and reset to zero. Clocks are tested using constraints on clocks, called *guards*. A TA over actions A is defined as a tuple $< N, l_0, E, V_C, I >$, where N and E are the locations and edges, $l_0 \in N$ is the initial location, V_C is the set of clock variables, and $I : N \longmapsto G$ with guards $g \in G$, actions $a \in A$, and a set of clocks $r \subseteq V_C$ to be reset (an alternative notation is $x := 0$ for the rest of a clock x). Figure 3 shows an example of a TA (model in right side), which is a formal representation of the *RiskStatus* state diagram (in left side). The actions used in the TA are `risky` and `no_risky`. The location *No Alert* is marked initial, as indicated by an extra circle inside it. The edge from `No Alert` to `Alert` is labelled with the action `risky` with a guard `status`. The mode has a clock `Time`, which is used to measure the time elapsed since the action `no_risk`. The location `After Alert` is labelled with a clock invariant to ensure that the delay is less than five time units between the actions `no_risky` and `risky/reset time`. Edges are labelled with guards to ensure that the delay is more than five time units and the current status is not alert, i.e., `status != alert`.

2.3. Security Assessment

Designing secure and reliable systems is challenging and attackers constantly find opportunities to compromise systems. There are different countermeasures at the disposal of organizations to cope with this challenge, such as applying best practices (e.g., OWASP [25]), penetration testing [26], established frameworks (e.g., Process for Attack Simulation and Threat Analysis (PASTA) [27]), or threat modeling [9].

Here, we facilitate threat modeling to analyze the security properties of the system under design. Via threat modeling one wants to reason about the complexity of a system, as well as identifying potential threats [28]. Usually, this is done by graphs, where each of the threats is modeled as a node and they are connected by edges [29]. Given a threat model, attack simulations allow to analyze attack scenarios on the described infrastructure [30, 31].

More concretely, we rely on the Meta Attack Language (MAL) as tool to perform our attack simulations. For a detailed overview of the MAL, we refer readers to the original paper [32]. First, a MAL DSL contains the main concepts of a domain under study, so called `assets`. An asset contains `attack steps`, which represent the actual attacks/threats that can be executed.

An attack step can be connected with *n* following steps creating an attack path, which is used for the attack simulation. Attack steps can be either OR or AND. Additionally, each attack step can be related with specific types of risks (i.e., confidentiality (C), integrity (I), and availability (A)). Furthermore, we have defenses at our disposal that do not allow connected attack steps to be performed. Finally, we can assign probability distributions to represent the effort to complete the related attack step. Assets have relations between them, so called `associations`. Moreover, we have inheritance between `assets` and each child asset inherits all the attack steps of the parent asset. Additionally, the assets can be organized into categories.

In List. 1, a short example of a MAL DSL is presented. In this example, we have four assets and their connections of attack steps from one asset to another. In the `Host` asset, the *connect* attack step is an OR attack step, while *access* is an AND attack step. The `->` symbol denotes the connected next attack step. For example, if an attacker performs *phish* on the `User`, it is possible to reach *obtain* on the associated `Password` and as a result finally perform *authenticate* on the associated `Host`. In the last lines of the example the `associations` between the assets are defined.

3. Automated Safety and Security Assessment

3.1. Architecture Framework

The MDSD approach allows developers to design systems using high-level abstractions (i.e., models). Then, these abstractions generate executable source code corresponding to the system's behavior. Our objective is to use an MDSD approach to model and build systems considering safety and security in their design. To do this, we plan to reuse the same abstractions that define programs to generate safety and security assessment artifacts. This conceptual model and the interaction between the sys-

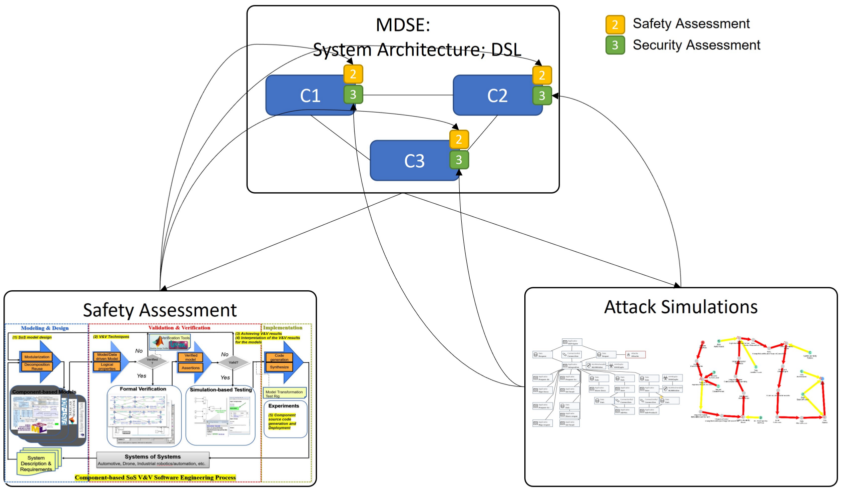


Figure 2: Methodology Roadmap

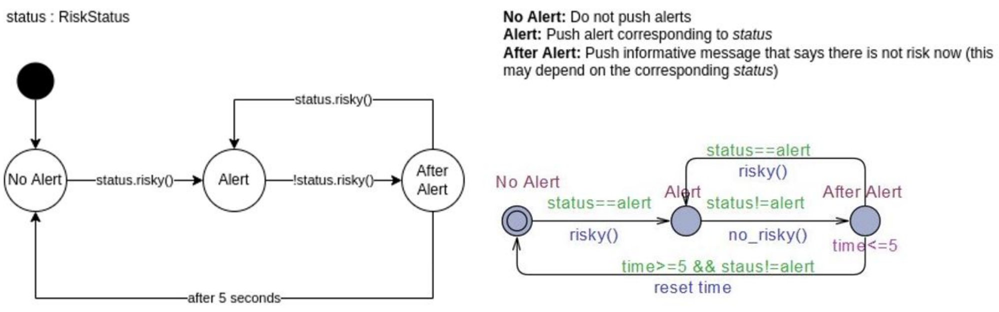
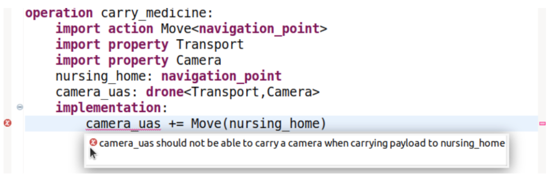


Figure 3: `ros-state diagram` for the risk status behavior of `Detect and Avoid (DAA)` service (left side) and its corresponding TA (right side)



```
operation carry_medicine:
    import action Move<navigation_point>
    import property Transport
    import property Camera
    nursing_home: navigation_point
    camera_uas: drone<Transport,Camera>
    implementation:
        camera_uas += Move(nursing_home)
```

camera_uas should not be able to carry a camera when carrying payload to nursing_home

Figure 4: Conceptual feedback of a privacy issue of a UAS moving to a location given directly in the DOTS DSL.

tem architecture, safety assessment, and security attack simulations can be seen in Figure 2. In the context of this work, we use the example of defining multi-UAS missions. In this system, we have two different high-level definitions: models derived over the DOTS DSL and models derived over the services and their interactions in the ROS-DOTS architecture.

First, the assessment of the safety and security properties of the models generated by the DSL can be explicitly shown to developers. To allow this level of feedback, a bi-directional connection between the models and the generated artifacts that validate security and safety properties should be available. In other words, models should generate artifacts in a way that the artifacts can relate to the models that generated them. Then, like in every modern Integrated Development Environment (IDE), the system can mark the problematic lines of code in the DSL program with a meaningful message for developers to fix the problems. For example, we can consider the case of a UAS transporting a package from point A to B. An operator can use any UAS capable of carrying a payload for this action. However, the system may restrict the use

```
category System {
  asset Network {
    | access
      -> hosts.connect
  }

  asset Host {
    | connect
      -> access
    | authenticate
      -> access
    | guessPwd
      -> guessedPwd
    | guessedPwd [Exp(0.02)]
      -> authenticate
    & access {C,I,A}
  }

  asset User {
    | attemptPhishing
      -> phish
    | phish [Exp(0.1)]
      -> passwords.obtain
  }

  asset Password {
    | obtain {C}
      -> host.authenticate
  }
}

associations {
  Network [networks] *
    <-- NetworkAccess --> * [hosts] Host
  Host [host] 1
    <-- Credentials --> * [passwords] Password
  User [user] 1
    <-- Credentials --> * [passwords] Password
}
```

Listing 1: Exemplary MAL Code

of a UAS with extra properties, which can entail security issues, for example, a UAS with a camera attached. An attacker can intercept the link between the UAS and the operator, accessing the camera images, which can bring privacy issues for the people living around the path of the UAS. While a UAS with a camera is essential for other use cases (e.g., monitoring a geographical area), we want to limit the use of the right UAS for the right job, to reduce security and safety issues. Figure 4 shows a concept of how this feedback can be displayed in DOTS.

Second, the safety and security properties of the architecture itself should also be checked and informed to developers. To do this, we can use the same idea of a bi-directional connection between the executable architecture and the generated artifacts that check security and safety properties. The architecture can generate artifacts to test properties over single services and the communication between multiple services. One example of a single service test is to check how the detect-and-avoid service works (DAA in Figure 1). This service alerts operators when a UAS is in a direct collision path with an external agent in the airspace. It is vital to check the safety properties of this service to ensure a safe interaction of agents in the airspace.

As a key example of a multi-service architecture tester, we present the dynamic replanning feature. This feature replans the path of every UAS involved in a mission when new constraints that affect the original plan are added into the airspace. This feature uses three services: Planner, Mission Monitor, and Dynamic Replanning. Suppose the dynamic replan service returns a plan with safety problems, such as 2 UAS in a direct path against each other. In that case, our safety checker can inform this issue directly to developers.

3.2. Formal framework for Safety and Security Assessment

To be trustworthy, systems need to remain safe and secure while being resilient to unpredictable changes, functional/operational failures and cyber-security threats. Rigorous V&V is essential to ensure trustworthiness of systems and clear definition of requirements is an important prerequisite for V&V.

Most engineering practice highly relies on V&V test-and-fix of system nature, which is time-consuming, expensive, and limiting the possibilities for exploration of alternatives in system design. Thus, we provide a correct-by-construction approach based on a combination of analysis techniques such as Systems Theoretic Process Analysis (STPA) [21] and formal verification such as model checking to generate critical requirements, remove ambiguities in the requirements, and specify formal safety and security properties that should be satisfied by the system. We also suggest a modularized/compositional approach for formal modeling to enhance re-usability and to reduce the complexity of formal modeling.

To facilitate the formal verification, the system architecture (illustrated in Figure 2) consisting of a set of function/service blocks and its operational behaviors are translated into a formal modeling description, UPPAAL [1] TA. To explicitly annotate and reason about the functional or operational behavior of each block (e.g., `DAA` in Figure 1) at the architecture level, we first adopt a state diagram (`RiskStatus ros-state diagram` in Figure 3) and extend it with a UML profile which integrates relevant concepts from our ROS-DOTS. Such a profiled model is then translated into a formal modeling description, UPPAAL TA. The translation process is supported in fully automatic by using our DSL. The communications between different blocks are also transformed into synchronization channels among other TAs in UPPAAL. Indeed, each `asset` and its behavioral logic in Listing 1 are visualized in a TA and the `associations` are represented in synchronization channels among different TAs.

[1]https://uppaal.org. An integrated tool environment for formal modeling, validation and verification of real-time systems modeled as networks of TA

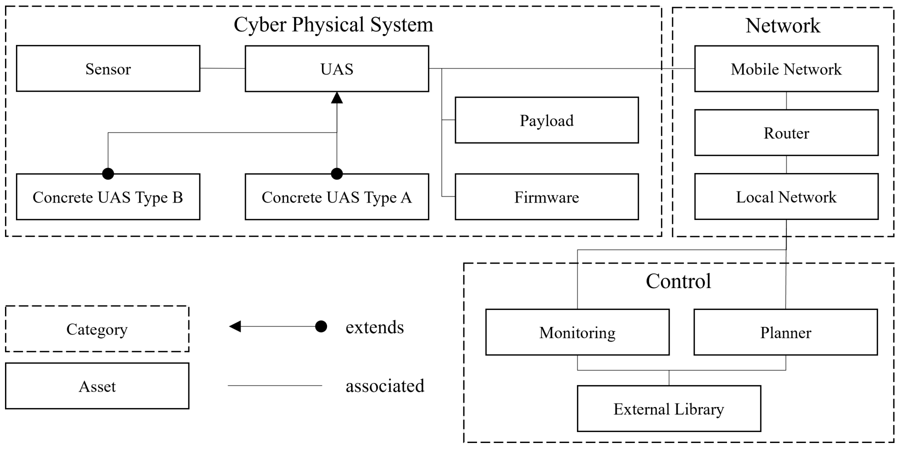


Figure 5: Sketch of a possible uasLang for security analysis

Similarly, a path-planning TA can be generated based on the `Planner` block and its `ros-state diagram`.

The auto-generated TA model is then amenable to formal verification using a UPPAAL model checker against safety and security requirements. Furthermore, additional safety and security constraints are identified based on STPA and feedback from model checking and attack simulation (which will be further explained in the following section). The system design can be refined by adding the new constraints which inhibit any hazards. Finally, we prove correctness and consistency of the safety and security constraints through formal verification and improve the system's design accordingly. In addition to formal verification, testing can be performed to validate the implementation (code as written/automatically generated by our DSL) actually runs correctly w.r.t. to the safe and secure design/specification.

The security of a system is affected by a broad variety of aspects. Formal verification of a system is an important step to assure to its security. However, to get a complete verification of the system, every detail about the system must be known. Moreover, depending on the system's size, a complete verification is time intensive and, thus, usually only conducted for extremely sensitive system like rockets transporting humans to space. Alternatively, a verification on an abstracted view of the system can be conducted, which is less time intensive, but accompanied by the cost of a not complete verification.

To address this shortcoming, we foresee in our architecture to not solely rely on verification, but also on threat modeling and attack simulations as they are also able to cope with incomplete information (i.e., "known unkowns" [33]). We facilitate MAL [32] as vehicle to perform out attack simulations. More concretely, we imagine a `uasLang` (cf. Figure 5) that might reuse certain parts of existing MAL languages [34], such as coreLang [35] for the fundamental IT parts or icsLang [36] for the operation technology (OT) similar parts (e.g., sensors or actuators).

In the following, we will shortly elaborate on the different assets that we envision in a possible `uasLang`. Firstly, we have the assets often referred to as cyber physical system. There is the *UAS* itself, which is the hardware platform carrying the *Payload.* Moreover, a *UAS* has *Sensors*, which help it to orientate itself in the real-world. The entire platform is controlled by the *Firmware*, which processes the incoming data from the *Sensors* and communicates with the controlling assets *Planner* and *Monitoring*.

The *Planner* is the central unit coordinating the different *UAS* and the routes they are taking to reach to their waypoints. The *Monitoring* receives the actual state of the *UAS* and provides an interface to external systems that might further process this data. Further, both assets can incorporate *External Libraries*, e.g., to perform better routing or providing additional reporting capabilities.

The communication between the *UAS* and the controlling units takes place in classical *Local Network*, like Ethernet, in which the IT parts are hosted and a *Mobile Network*, like 5G, that covers the operation area of the *UAS.* These two networks are usually separated by some kind of *Router* restricting the network access.

Given `uasLang` and the models created during the model driven software development, we are now able to create a threat model that represents the actual infrastructure and perform attack simulations in securiCAD [30]. The simulations will tell us potential threats in the architecture and in which time an attacker might be able to exploit them. This information is then played back to the developer, so that they are able to determine possible countermeasures in their system's architecture to improve the security.

4. Challenges & Future Directions

The distinctive features of cyber-physical systems with regard to model checking are their complexity, modularity and the need to comply with safety and security requirements, which means that every failure result should be thoroughly analyzed and fixed. Despite being one of the most reliable approaches for ensuring system correctness, model checking requires additional knowledge about a system as a whole and efforts aimed to localize an error in the model of the system. A tool or framework that supports user-friendly model checking which focuses on explanation of negative verification results and performs an analysis that the refined system design is still consistent would be highly beneficial. For example, in the system verification process, once a violation of a safety or security constraint/requirement is detected, a counterexample (failure trace) generated by a UPPAAL model checker that can be visualized in the architecture model in order to pinpoint on which time step/block has caused the property failure. Seemingly including such aspects into our framework and tool is crucial for making the formal verification techniques more approachable to engineers.

For the security assessment, we recognize that not all parts of the architecture (cf. Figure 1) are reflected one-to-one in `uasLang`. However, this is no issue due to two reasons. Firstly, the security assessment takes the point of view of an attacker. Thus, we are not solely interested in the system's architecture, but to all parts that are exposed to the environment. Consequently, `uasLang` contains further information (e.g., on concrete UAS deployed), that might be provided from outside of our presented solution. Other parts of the architecture might not be of greater relevance for the security assessment (e.g., the web server), as they do not change in our setting (and even do not have a representation in the used model driven software development approach) and thus do not have any influence on the outcome of the attack simulation results.

Secondly, it is recommended to base a newly developed MAL DSL on another existing MAL DSL [34, 37]. Consequently, we would base `uasLang` at least on `coreLang` [35]. In other words, we would have all assets available that are available in the base language (i.e., `coreLang`) and, thus, would be able to model all IT related assets presented in Figure 1.

Moreover, we have more consideration on the modeling of the architecture, when we relate this modeling to users and developers. For example, a DSL is built to be concise, to have the necessary features to describe the solution using models, and no more. However, developers may need to define extra properties for assessing security and safety properties, that are not needed to solve the problem itself. In other words, they may need to define aspects of the incidental complexity of the problem they are solving. While this may be an issue, we envision an architecture where developers should declare as few as possible incidental properties. In addition, even when they need to declare these properties, the system may allow them to define them in other parts of the system, such as configuration files, environment variables, directly into the architecture, etc.

Finally, we are aware that checking safety and security properties may take some time, making it inappropriate to give feedback to developers when they are writing their programs. Even when certain properties can be checked fast enough to give them while developers write their programs, we need to be aware of the properties that can take more time. For the properties that take a considerable amount of time, we envision a system that gives feedback to developers when the program is running (similar to debugging), or when the program finishes its execution (similar to the case of automatic testing or continuous integration/delivery systems).

References

[1] T. Mens, On the complexity of software systems, Computer 45 (2012) 79–81.

[2] K. Zhang, D. Han, H. Feng, Research on the complexity in internet of things, in: 2010 International Conference on Advanced Intelligence and Awarenss Internet (AIAI 2010), IET, 2010, pp. 395–398.

[3] O. Pastor, S. España, J. I. Panach, N. Aquino, Model-driven development, Informatik-Spektrum 31 (2008) 394–407.

[4] M. Waidner, M. Backes, J. Müller-Quade, Development of secure software with security by design, Fraunhofer-Verlag, 2014.

[5] E. M. Clarke, J. M. Wing, Formal methods: State of the art and future directions, ACM Comput. Surv. 28 (1996) 626–643.

[6] F. Benaben, M. Larnac, J. Pignon, J. Magnier, A process for improving multi-technology system high level design: Modeling, verification and validation of complex optronic systems, in: 2000 International Conference On Systems, Man & Cybernetics, volume 1–5, IEEE, 2000, pp. 1036–1040.

[7] C. Baier, J.-P. Katoen, Principles of Model Checking, MIT Press, Cambridge, 2007.

[8] B. Arkin, S. Stender, G. McGraw, Software penetration testing, IEEE Security & Privacy 3 (2005) 84–87.

[9] W. Xiong, R. Lagerström, Threat modeling–a systematic literature review, Computers & security 84 (2019) 53–69.

[10] S. Beydeda, M. Book, V. Gruhn, et al., Model-driven software development, volume 15, Springer, 2005.

[11] M. Völter, T. Stahl, J. Bettin, A. Haase, S. Helsen, Model-driven software development: technology, engineering, management, John Wiley & Sons, 2013.

[12] M. Fowler, R. Parsons, Domain-specific languages, Addison-Wesley Professional, 2010.

[13] M. Campusano, N. Heltner, N. Mølby, K. Jensen, U. P. Schultz, Towards declarative specification of multi-drone bvlos missions for utm, in: 2020 Fourth IEEE International Conference on Robotic Computing (IRC), IEEE, 2020, pp. 430–431.

[14] M. Campusano, K. Jensen, U. P. Schultz, Towards a service-oriented u-space architecture for autonomous drone operations, in: 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE), IEEE, 2021, pp. 63–66.

[15] P. David, V. Idasiak, F. Kratz, Reliability study of complex physical systems using sysml, Reliability Engineering & System Safety 95 (2010) 431–450.

[16] EAST-ADL, last accessed on 20.10.2022. http://maenad.eu/index.htm.

[17] E. Kang, P. Schobbens, Schedulability analysis support for automotive systems: from requirement to implementation, in: Symposium on Applied Computing, ACM, 2014, pp. 1080–1085.

[18] L. Huang, T. Liang, E. Kang, Tool-supported analysis of dynamic and stochastic behaviors in cyber-physical systems, in: 19th International Conference on Software Quality, Reliability and Security, IEEE, 2019, pp. 228–239.

[19] E. Kang, G. Perrouin, P. Schobbens, Model-based verification of energy-aware real-time automotive systems, in: 18th International Conference on Engineering of Complex Computer Systems, IEEE, 2013, pp. 135–144.

[20] P. Feiler, D. Gluch, Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language, Addison-Wesley Professional, 2012.

[21] N. Leveson, J. Thomas, STPA Handbook, Cambridge, 2018.

[22] I. Friedberg, K. McLaughlin, P. Smith, D. Laverty, S. Sezer, STPA-SafeSec: Safety and security analysis for cyber-physical systems, Journal of Information Security and Applications 34 (2017) 183–196.

[23] W. Young, N. G. Leveson, An integrated approach to safety and security based on systems theory, Commun. ACM 57 (2014) 31–35. doi:10.1145/2556938.

[24] R. Alur, D. L. Dill, A theory of timed automata, Theoretical Computer Science 126 (1994) 183–235.

[25] M. Bach-Nutman, Understanding the top 10 owasp vulnerabilities, arXiv preprint arXiv:2012.09960 (2020).

[26] M. Bishop, About penetration testing, IEEE Security & Privacy 5 (2007) 84–87.

[27] M. M. Morana, T. Uceda Vélez, Risk centric threat modeling: Process for attack simulation and threat analysis, John Wiley & Sons, Hoboken, New Jersey, 2015.

[28] A. Shostack, Threat modeling: Designing for security, Wiley, Indianapolis, IN, USA, 2014.

[29] S. Myagmar, A. J. Lee, W. Yurcik, Threat modeling as a basis for security requirements, in: SREIS, volume 2005, Citeseer, 2005, pp. 1–8.

[30] M. Ekstedt, P. Johnson, R. Lagerström, D. Gorton, J. Nydrén, K. Shahzad, securiCAD by foreseeti: A CAD tool for enterprise cyber security management, in: 19th International EDOC Workshop, IEEE, 2015, pp. 152–155.

[31] H. Holm, K. Shahzad, M. Buschle, M. Ekstedt, P^{2}CySeMoL: Predictive, probabilistic cyber security modeling language, IEEE Trans Dependable Secure Comput 12 (2015) 626–639.

[32] P. Johnson, R. Lagerström, M. Ekstedt, A meta language for threat modeling and attack simulations, in: 13th ARES Conference, 2018, pp. 1–8.

[33] S. Hacks, M. Kaczmarek-Heß, S. de Kinderen, D. Töpel, A multi-level cyber-security reference model in support of vulnerability analysis, in: International Conference on Enterprise Design, Operations, and Computing, Springer, 2022, pp. 19–35.

[34] S. Hacks, S. Katsikeas, Towards an ecosystem of domain specific languages for threat modeling, in: International Conference on Advanced Information Systems Engineering, Springer, 2021, pp. 3–18.

[35] S. Katsikeas, S. Hacks, P. Johnson, M. Ekstedt, R. Lagerström, J. Jacobsson, M. Wällstedt, P. Eliasson, An attack simulation language for the it domain, in: International Workshop on Graphical Models for Security, Springer, 2020, pp. 67–86.

[36] S. Hacks, S. Katsikeas, E. Ling, R. Lagerström, M. Ekstedt, powerlang: a probabilistic attack simulation language for the power domain, Energy Informatics 3 (2020) 1–17.

[37] S. Hacks, S. Katsikeas, E. Rencelj Ling, W. Xiong, J. Pfeiffer, A. Wortmann, Towards a systematic method for developing meta attack language instances, in: International Conference on Business Process Modeling, Development and Support, International Conference on Evaluation and Modeling Methods for Systems Analysis and Development, Springer, 2022, pp. 139–154.