Manuscript version: Author's Accepted Manuscript
The version presented in WRAP is the author's accepted manuscript and may differ from the published version or Version of Record.

Persistent WRAP URL:
http://wrap.warwick.ac.uk/114372

How to cite:
Please refer to published version for the most recent bibliographic citation information.
If a published version is known of, the repository item page linked to above, will contain details on accessing it.

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions.

Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

Publisher's statement:
Please refer to the repository item page, publisher's statement section, for further information.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk.

Simultaneous Fault Models for the Generation and Location of Efficient Error Detection Mechanisms

Matthew Leeke

Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK
Email: matthew.leeke@warwick.ac.uk

The application of machine learning to software fault injection data has been shown to be an effective approach for the generation of efficient error detection mechanisms (EDMs). However, such approaches to the design of EDMs have invariably adopted a fault model with a single-fault assumption, limiting the relevance of the detectors and their evaluation. Software containing more than a single fault is commonplace, with safety standards recognising that critical failures are often the result of unlikely or unforeseen combinations of faults. This paper addresses this shortcoming, demonstrating that it is possible to generate efficient EDMs under simultaneous fault models. In particular, it is shown that (i) efficient EDMs can be designed using fault injection data collected under models accounting for the occurrence of simultaneous faults, (ii) exhaustive fault injection under a simultaneous bit flip model can yield improved EDM efficiency, (iii) exhaustive fault injection under a simultaneous bit flip model can be made non-exhaustive and (iv) EDMs can be relocated within a software system using program slicing, reducing the resource costs of experimentation to practicable levels without sacrificing EDM efficiency.

Keywords: Detection; Error; Model; Location; Machine Learning

Received August 20, 2018; revised December 16, 2018; accepted February 21, 2019

1. INTRODUCTION

The design of error detection mechanisms (EDMs) is integral to the development of dependable software systems [1]. EDMs are fundamentally concerned with the detection of erroneous software states. Once detected by an EDM, erroneous software states can be handled by error recovery mechanisms (ERMs) to maintain proper function. A failure to contain the propagation of erroneous state is known to make recovery more difficult, leading to a focus on the efficiency of EDMs through measures such as coverage and latency [2].

The effectiveness of an EDM has been shown to depend on two factors. These factors are (i) the error detection predicate that it implements and (ii) its location in a software system [3]. This gives rise to two related problems. Firstly, the EDM design problem, which is concerned with the derivation of an error detection predicate over program variables that can be used for the detection of erroneous system states. Secondly, the EDM location problem, which is concerned with the identification of those software locations at which an EDM will be most effective. Though often treated as orthogonal for simplicity, the interaction of the implemented error detection predicate and the software location are demonstrably critical to the efficiency of an EDM [4].

The efficiency of an EDM can be characterised by completeness and accuracy [3]. Completeness is the capability of an EDM to detect erroneous states, i.e., its associated true positive rate. In contrast, accuracy is the capability of an EDM to avoid incorrectly detecting erroneous states, i.e., its associated false positive rate. An erroneous state is one that will lead to system failure if the error is not handled, where a failure is characterised as a violation of a system specification. An EDM that is complete and accurate is commonly known as a perfect detector. Due to implementation constraints, it is not generally possible to generate or guarantee the existence of a perfect detector for a particular software location [5].

The role of a fault model is to provide a means for analysing the response of software system to the presence a well defined set of faults, such that appropriate EDMs and ERMs can be designed to impart dependability. The assumption that faults do not occur simultaneously or interact is a limitation of many fault models and the software fault injection frameworks that implement them, not least because software systems containing more than a single fault are commonplace. Indeed, numerous existing safety standards recognise that critical system failures are often the result of unlikely or unforeseen interactions combinations of faults [6].

It has been shown that efficient error detection predicates for EDMs can be designed through the application of machine learning algorithms to data sets generated during software fault injection [7]. This approach demonstrated, under a transient data value fault model, that it was possible to generate error detection predicates for specified locations with a true positive rate of nearly 100% and a false positive rate close to 0% for the detection of failure-inducing states. As is consistent with the overwhelming majority of software fault injection frameworks, these results were achieved under a single-fault assumption, calling into question their relevance in the context of real-world software systems.

1.1. Problem Statement

In generating error detection predicates for EDMs through the application of machine learning to fault injection data, an implication of the single-fault assumption is that the efficiency of the EDMs generated is relevant in the context of a single fault. This implication limits the application of these EDMs in practical software systems.

This paper addresses this problem by demonstrating that practicable simultaneous fault injection can be used to generate efficient EDMs for a specified location in a software system. In doing this it is shown that the adoption of a simultaneous fault model enables a larger set of faults to be captured than existing models that make the single-fault assumption. The viability of simultaneous fault models is based on results demonstrating that exhaustive fault injection under a simultaneous bit flip model can be made non-exhaustive and that EDMs can be relocated without regeneration.

1.2. Contributions

This paper makes several specific contributions to the design of efficient EDMs. In particular, the research presented demonstrates that:

- Efficient EDMs can be designed using fault injection data collected under models accounting for the occurrence of simultaneous faults;
- Exhaustive fault injection under a simultaneous bit flip model can yield better EDM efficiency than under a non-simultaneous fault model;
- Exhaustive fault injection under a simultaneous bit flip model can be made non-exhaustive without sacrificing the efficiency of the resultant EDMs, thus reducing the resource costs of experimentation to a practicable level.
- Efficient EDMs can be relocated within a software system using program slicing without sacrificing the efficiency of those EDMs.

1.3. Paper Structure

The remainder of this paper is structured as follows: Section 2 provides an overview of research relating to fault models for detector design. Section 3 details the adopted system and fault models. Section 4 provides an overview of how machine learning algorithms can be used to generate detection predicates for specified software locations. Section 5 provides details of the experiments conducted in this paper, including the target software systems and applied machine learning algorithms. Section 6 presents the results associated with the experiments conducted, alongside a discussion of their significance in the context of efficient error detector design. Section 7 concludes the paper with a summary of findings and a brief discussion of future work in EDM design.

2. RELATED WORK

A fault model has been shown to be composed of two parts, a local model and a global model [8]. The local fault model states the type of faults that are assumed to occur, whilst the global model dictates the extent to which the local fault model can occur. Ideally software should be examined under a representative workload and fault model. However, analyses under such circumstances can be impractical due to numbers of test cases or the fault space, particularly in the case of simultaneous fault models that demand consideration of fault combinations. Research has addressed these problems through leveraging parallel architectures in the execution of the large number of fault injection experiments [9], though more scalable techniques have focused on sampling strategies for the test cases and error spaces [10, 11].

The origin of the established bit flip fault model is in the diagnosis of hardware faults. In the context of software fault injection, bit flip and stuck-at fault models are often used to mimic transient and permanent hardware faults [12]. There has been much research on the representativeness of the faults captured by fault models used in software fault injection, motivated by results showing the issue can impact the validity of fault injection analysis. The results presented in [11] demonstrated that representativeness and resource efficiency in fault injection can be improved through the use of machine learning techniques and software metrics, a shift in emphasis from fault-analysis focused approaches.

The simultaneous fault models evaluated in this paper were proposed in [13] in response to a proliferation of software fault injection frameworks making the single-fault assumption, despite this being known to be unrealistic [6]. The models were developed on notions of coincidence and impact before being evaluated with regard to utility using metrics such as coverage and failure induction. Note that this focus is distinct from research exploring the simultaneous execution of fault injection experiments [14].

The focus of this paper is on the generation and location of efficient error detection predicates for EDMs under simultaneous fault models. The application of machine learning to EDM design is appealing because it does not assume the availability of a formal system specification or rely on the experience of software engineers, since the application of each has been shown to provide undesirable levels of detection efficiency [15]. The approach is also applicable to real-world software systems, as opposed to smaller finite-state constructions. Given that simultaneous fault models are considered to be more representative of practical software than models used to-date, the efficiencies presented in this paper provide a more representative commentary on the efficacy of machine learning for EDM design.

3. MODELS

In this section the system, fault and data models used in this paper are described.

3.1. System Model

A software system S is taken to be a tuple, consisting of a set of software modules, $M_1 \ldots M_n$, and a set of connections. A software module M_k consists of an import interface I_k, an export interface E_k, a set of non-composite program variables V_k and a sequence of actions $A_{k1} \ldots A_{ki}$. Each program variable in V_k has a domain of values. Each action in $A_{k1} \ldots A_{ki}$ may read or write to a subset of V_k. Two software modules M_k and M_l are connected if the export interface of M_k is matched with the import interface of M_l, i.e., a connection exists if E_k is matched with I_l. Thus, a software system $S = (MOD, CON)$, where $MOD = \{M_1 \ldots M_n\}$, and $CON = \{(M_k^a, M_l^a)\}$, where M_k exports to the import interface of M_l over connection a. The adopted system model is consistent with [7] and compatible with the simultaneous fault models developed in [13].

3.2. Fault Models

The simultaneous fault models described were developed to improve software fault injection by overcoming the single-fault assumption, thus permitting more meaningful analyses [13]. As the fault injection conducted in [13] focused on the point of entry to modules, the fault injection experiments in this paper focus on generating EDMs at the entry points to modules. This approach is supported by existing research findings [16], noting that injecting faults at the interface of a software module is not equivalent to injecting faults in the body of the same module [17]. To maintain compatibility with existing research, software state was characterised by all variables in scope at the point of fault injection. The described fault models were systematically applied to exhaustive combinations of all variables in scope.

Fault models were used only for the collection of software fault injection data that served as input to the machine learning algorithms, i.e., the machine learning algorithms took no account of the fault model or data generation process.

3.2.1. Bit Flip (BF)

The BF fault model injects a single bit flip fault into the representation of a single variable in each fault injection experiment, thus incorporating the single-fault assumption and providing a broad basis for comparison with simultaneous fault models. This is a well established model for software fault injection, being consistent with fault models used in previous work on the application of machine learning for the generation of efficient EDMs [7, 12].

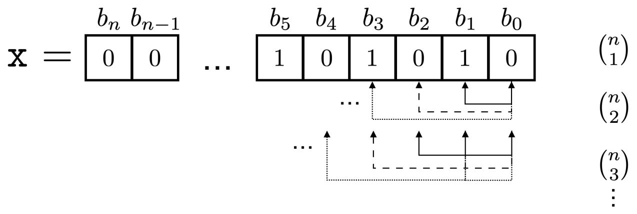
3.2.2. Simultaneous Fuzzing (FuzzFuzz)

A single fuzzing injection involves the modification of a variable value to a random value of the same bit length. If fuzzing were simultaneously applied to a single variable, the result would not differ from a single fuzzing injection. This means there is no need to consider injection in the same variable. Rather, under the FuzzFuzz model, the values of more than one target variable are subject to fuzzing. Exhaustive software fault injection using fuzzing is impractical, since the number of possible injection values for an n-bit variable v is 2^n. For this reason we restrict experimentation to a fixed number of fault injections for each variable combination simultaneously targeted under FuzzFuzz, adhering to the experimental guidance derived from results presented in [13] and [18].

3.2.3. Simultaneous Bit Flip (SimBF)

The SimBF fault model performs fault injections at the resolution of a single variable. In the original formulation of this fault model, only combinations of two bit flips were considered [13]. To ensure that initial experiments under this model could be said to exhaustive, bit flip fault injection was systemically applied to exhaustive k-combinations of bits in each variable representation. Coupled with the exhaustive consideration of combinations of variables in which to inject, this fault model requires a large number of experiments to consider exhaustively. For example, just one of the 16-bit variables used by the function shown in Figure 1 would require $\binom{n}{1} + \binom{n}{2} + ... + \binom{n}{n} = 65,535$ experiments. This means that at least 196,605 experiment would be required to analyse each variable in the function individually, even before the

```
short sum (short x, short y, short z) {
    return (x + y + z);
}
```



An n-bit variable requires $\binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n}$ experiments to be exhaustively considered, i.e., x requires 65,535 experiments.

FIGURE 1: The experimental cost of the SimBF fault model for a single variable in a one-statement function.

combinations of variables are considered. It should be noted that this number of experiments is not practical at scale. An exhaustive approach is initially used in this paper to set a standard for EDM efficiencies, such that the efficiencies of EDMs generated by non-exhaustive approaches can be better understood.

4. EDM GENERATION USING MACHINE LEARNING

Recall that the effectiveness of an EDM has been shown to depend the error detection predicate it implements and its location in software. If the location is known, EDM generation is tantamount to the generation of an error detection predicate for implementation. The premise of applying machine learning to fault injection data is that an error detection predicate can the derived because the data generated during fault injection captures aspects of the relationships between software states and future system failure. As data collected during fault injection analysis provides an indication of whether a sampled software state resulted in a failure, the generation of an error detection predicate is a supervised learning problem.

Data is assumed to exist as a single relation consisting of a set of n input attributes that define an n-dimensional space, I. Every point in I is a potential state of the process being modelled. In supervised learning an algorithm is tasked with learning a good approximation, $\hat{f}$, of an unknown function, f, given a training data set, $T \subseteq I$, consisting of the N pairs $\langle x_i, f(x_i)\rangle$. In the case of learning from fault injection data the function known to be discrete and binary, since a software state either leads to a system failure or a successful execution. The task of learning a binary function is is referred to as concept learning, a special case of classification. Within a data set, instances of the class of interest, known as the concept, are referred to as positive instances. Instances not belonging to the concept are referred to as negative instances.

A process for EDM generation using fault injection data is described in [7]. The process consists of five stages, where these stages include the evaluation of EDM efficiency. The stages of the process, as reflected in Sections 4.1-4.5, are: Data Collection (4.1), Data Preprocessing (4.2), Model Generation (4.3), Model Refinement (4.4), and Model Evaluation (4.5).

4.1. Data Collection

The fault injection process is a means for the acquisition of data that captures the functional relationship between software state and system failure. The fault model applied in fault injection dictates the nature and extent of the exploration of software states, making the selection and robust application of a representative fault model imperative. The exploration of simultaneous fault models is the fundamental concern of this paper, focusing fault injection analysis on the fault models described in Section 3.

4.2. Data Preprocessing

Preprocessing transforms fault injection data into a suitable relational data format for learning. This transformation provides an opportunity to address issues such as class imbalance, which can prevent the development of reliable predictive models in concept learning problems [19]. The data sets resulting from fault injection analysis often contain significantly fewer positive instances than negative instances, i.e., there are significantly fewer examples of system failure than successful execution. This feature of the fault injection data must be accounted for before predictive models are generated. It is appropriate to tackle this problem in preprocessing because most approaches to address class imbalance require the generation of derivative data sets, a task made simpler if these are produced during data transformation.

4.3. Model Generation

Symbolic pattern learning algorithms are an effective class of algorithm for the generation of error detection predicates, not least because their output can easily be interpreted as first-order predicates. This paper applies decision tree induction and rule induction as machine learning algorithms, since these have been shown to be capable of generating efficient, in some cases near-perfect, predicates for EDMs [7]. The function approximation learnt, referred to as the model, by a classification algorithm needs to be evaluated, in order to obtain a measure of the expected accuracy of the model on unseen data. Typically the accuracy of a model is measured by the percentage of test data instances correctly classified, hence most algorithms seek to learn hypotheses that minimise the number of errors. Conveniently this is consistent with the notions of accuracy and completeness used in the measurement of EDM efficiency. However, this implicitly assumes that all types of misclassification incur an equal cost,

TABLE 1: General concept learning confusion matrix.

		Predicted Class		
		Pos.	Neg.	Marginal Sum
Actual	Pos.	TP	FN	n_{pos}
	Neg.	FP	TN	n_{neg}
	Marginal Sums	$\hat{n}_{pos}$	$\hat{n}_{neg}$	n

which is rarely the case. For example, in safety-critical context, a model misclassifying a failure-inducing state will typically result in a more significant cost than a non-failure-inducing state being misclassified.

4.4. Model Refinement

Models are refined by varying the parameters associated with the applied machine learning algorithms. This is achieved by repeating the execution of the algorithms under different configurations on the undersampled and oversampled data sets generated during preprocessing to establish an algorithm configuration and data set that yields the most efficient EDM. Achieving a perfect detector may not be possible for a given location. This is not a direct limitation of the machine learning approach, rather it is a theoretical constraint of the EDM design problem [5].

4.5. Model Evaluation

The predictions made by a model for a given data set can be cross-tabulated with the classes assigned to the instances by the target function to produce a confusion matrix. Table 1 shows the general form of a confusion matrix for concept learning. TP is the number of positives instances labelled positive by $\hat{f}$, known as true positives, whilst FN is the number of positive instances labelled negative, known as false negatives. FP is the number of negative instances labelled positive, known as false positives, whilst TN is the number of negative instances labelled negative, known as true negatives. Finally, n_{pos}/n_{neg} are the number of positive/negative instances in the test data and $\hat{n}_{pos}/\hat{n}_{neg}$ are the number of instances predicted as positive / negative.

4.5.1. The Area Under Curve (AUC) Metric

Aggregate measures of model quality seek to balance the concerns of the confusion matrix shown in Table 1, such that favourable performance in one area is not achieved through the neglect of performance in another. The most basic of these measures are true negative rate ($TNR = \frac{TN}{TN+FP}$) and true positive rate ($TPR = \frac{TP}{TP+FN}$). These measures give rise to ROC analysis, which is based on a plot in two dimensions where each model is a point defined by the coordinates (1-TNR, TPR). Note that (1-TNR) is also referred to as the false positive rate ($FPR = \frac{FP}{TN+FP}$).

Under different configurations, the same classifier will produce multiple points on such a plot. The Area Under the ROC Curve (AUC) obtained by joining these points to (0,0) and (1,1), as in Equation 1, is one of the most common metrics for measuring model performance.

An AUC of 0.5 implies random model performance, i.e., a coin toss in the case of concept learning. An AUC of 1.0 indicates that a model is near-perfect in its discrimination. As the focus of this paper is on evaluating the impact of more practical fault models on EDM efficiency, where this is understood as their accuracy and completeness, the AUC metric is a natural choice for use in model evaluation.

$$AUC = \frac{TPR - FPR + 1}{2} \tag{1}$$

Misclassification costs are likely to vary in the context of dependable software systems, hence steps must be taken to ensure that favourable AUC values are not achieved through the neglect of accuracy or completeness. With this in mind, TPR and FPR are also considered by the results presented in Section 6.

4.6. Model Relocation

The cost of generating a model for a specified location makes it desirable for existing model to be relocatable without model regeneration or the loss of detection efficiency. As the model takes the form of a first-order predicate over program variables, program slicing can be used to relocate a model once it has been generated. The concept of program slicing was first proposed by Weiser [20]. The majority of program slicing approaches rely on the construction of a program dependence graph (PDG) [21], though approaches can also be categorised according to whether they provide static or dynamic, inter-procedural or intra-procedural, executable or non-executable, and forward or backward slicing.

In the context of relocating a generated model for efficient error detection it is necessary to adopt a static slicing approach, since the cost of applying dynamic slicing approaches is typically commensurate with the model regeneration that program slicing seeks to avoid. As models exist at the entry-point of software modules it is desirable that the slicing approach be capable of accounting for inter-procedural dependences and producing forward program slices. Figure 2 shows the body of a sample program and its associated PDG, where data dependencies between program statements are shown as solid lines and control dependencies are shown as dotted lines [22].

EDM relocation involves the abstract interpretation of the target software module, whereby the generated error detection predicate serves as the starting point for interpretation and the conjunctive rules of the error detection predicate are updated as the dependencies captured by the PDG are encountered during interpretation. Where a data dependency relating to a set of variables exists between successive statements, all conjunctive rules involving any variable

```
S1:  int a, b, x = 7;
S2:  if(x>0) {
S3:    a = x*x;
S4:    b = x-3;
     } else {
S5:    if (x<0) {
S6:      a = x*x*x;
S7:      b = x-6;
       } else {
S8:      a = x*x*x*x;
S9:      b = x-9;
       }
     }
S10: print(a);
S11: print(b);
```

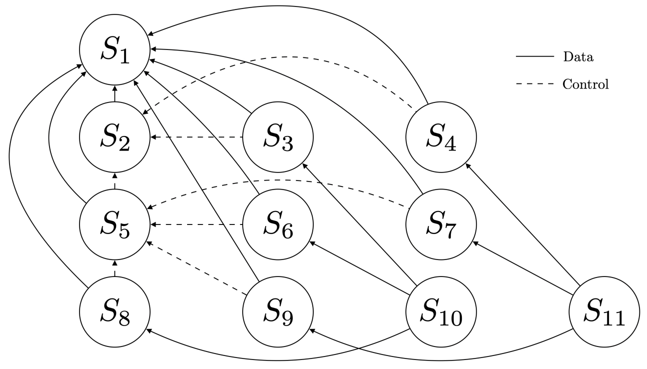


FIGURE 2: Example program body and its associated program dependency graph for model relocation [22].

in that set are necessarily subject to modification. The modification of conjunctive rules is performed with respect to the conjunctive rules that exists for the statement that is the target of the data dependency. Control dependencies encountered are considered to result in the duplication of the error detection predicate according to the number of control paths the dependency could introduce, with a modification of the conjunctive rules being performed for each path as dictated by data dependencies. Where it is not possible to resolve a data or control dependency with regard to how all conjunctive rules involving a variable should be updated, every rule involving that variable must be pruned. This process of abstract interpretation, guided by the dependencies identified in program slicing, is akin to is an approach that has been shown to provide an effective EDM design mechanism [23]. The development, application and evaluation of this approach to demonstrate the efficacy and experimental practicality of relocating efficient EDMs represents a technical contribution over [24].

5. EXPERIMENTAL SETUP

In this section the experimental approach employed in this paper is explained, including coverage of the target software systems.

5.1. Data Collection

Four software systems were subject to experimentation. Five randomly chosen modules in each of these software systems were selected for experimentation. System failures were identified through comparison with a fault-free execution, where any discrepancy in output or the completion of the test case was deemed a failure.

5.1.1. 7-Zip Archiving Utility (7Z)

7-Zip is a compression utility that supports archiving and encryption [25]. 7-Zip is widely-used, modular, written in C/C++ and has been designed, developed and maintained by a community of software engineers. Most source code and resources for 7-Zip are available under the GNU Lesser General Public License. A single file archiving procedure was executed as a test case.

5.1.2. FlightGear Flight Simulator (FG)

FlightGear is an open source flight simulator [26]. The software is modular, contains more than 250,000 lines of C/C++ and simulates a safety-critical situation. All source code and resources for FlightGear are available under the GNU General Public License. An aircraft takeoff procedure was executed as a test case.

5.1.3. MP3Gain (MG)

MP3Gain is an open source volume normaliser [27]. MP3Gain is modular, written in C/C++ and has been predominantly developed by a single software engineer. All source code for MP3Gain is available under the GNU General Public License. A single file volume normalisation procedure was executed as a test case.

5.1.4. ImageMagick (IM)

ImageMagick is an open source image editing suite that can be utilised from the command line [28]. ImageMagick is modular, written in C/C++ and has been designed, developed and maintained by a small team of software engineers. All source code and resources for ImageMagick are available under the Apache 2.0 license. A colour balancing, crop and scaling image procedure was executed as a test case.

5.2. Data Preprocessing

The limited simultaneous fault model support in fault injection tools meant a bespoke framework was used for the injection of faults. This meant a format transformation between the fault injection logging and the Attribute Relation File Format (ARFF) used by the Weka Data Mining suite for model generation [29]. In this format, each variable in scope is an attribute and the class label identifies system failures.

There are two general approaches for addressing class imbalanced data sets. Either the data distribution can be implicitly changed or the data set can be resampled to make the class distribution more uniform. As there is

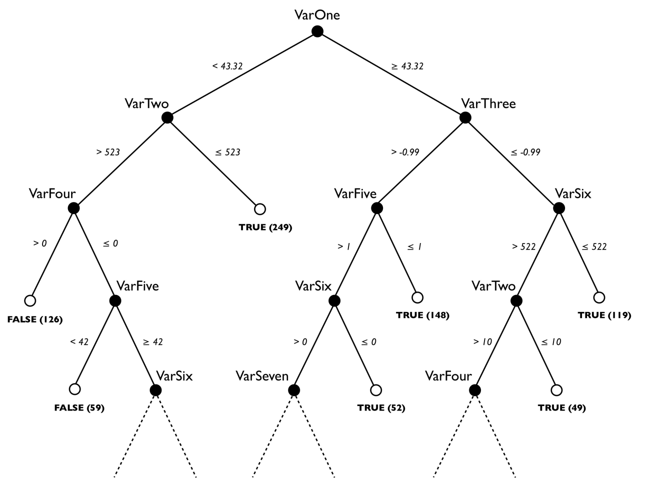


FIGURE 3: Example C4.5 generated decision tree [7].

no robust way to identify generally appropriate weights to associate with training examples, resampling was used in preprocessing. In particular, Synthetic Minority Oversampling Technique (SMOTE) was used to address class imbalance. SMOTE generates synthetic samples for minority classes along the line segment joining an example to k-minority class nearest neighbours, with cross validation being used to set the level of oversampling and undersampling of each class [30].

5.3. EDM Generation

The Weka Data Mining suite provided the baseline algorithms used in model generation [29]. In particular, the Weka Data Mining suite provided implementation of the C4.5 decision tree induction and Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithms [31, 32].

5.3.1. Decision Tree Induction

Decision tree induction learns a disjunction of conjunctive rules describing a concept. As shown in Figure 3, a decision tree consists of two node types; decision nodes and leaf nodes. A decision node contains an input attribute value. Each edge emanating from a decision node is labelled with one of the unique values in the domain of the attribute labelling the decision node. A leaf node is labelled using one of the classification labels. Each path of the tree from the root node to a leaf node is interpreted as a set of conjunctive expressions that lead to the classification label at the leaf node. The algorithm performs a greedy search of the space of all possible trees, choosing decision node attributes that maximise the reduction in entropy of the class label.

5.3.2. Rule Induction

Rule Induction operates in distinct phases. Specifically, beginning with with the least represented class label, the algorithm repeatedly grows and prunes rules until there are no positive examples left or the error rate is greater than 0.5. A rule is grown by incorporating greedy conditions until the rule is perfectly accurate. This is done by attempting to incorporate every possible value of each attribute and selecting the condition providing most information gain. A rule is pruned by removing any final sequences of antecedents according to a fixed pruning metric, providing some facility to incorporate domain knowledge and combat overfitting.

5.4. EDM Refinement

Refinement used 20 undersampling and 15 oversampling levels, distributed uniformly over [5,100] and [100,1500] for undersampling and oversampling respectively. The number of nearest neighbours considered in sampling process was uniformly distributed over [1,15]. These ranges were chosen to provide insight into achievable improvement, as opposed to a comprehensive parameter search pursuing optimal model performance.

5.5. EDM Evaluation

Following the application of each machine learning algorithm to each fault injection data set, 10-fold cross validation was used to generate a confusion matrix. The use of 10-fold cross validation involved the entries in each data set being partitioned into 10 stratified samples. For each of the ten cross validation runs, just one of these partitions is used as a test sample, whilst the other nine partitions are used as the training set for the machine learning algorithm.

5.6. EDM Relocation

After evaluating each EDM the most efficient EDMs in each software module were relocated to the exit-point of the module using forward slicing. Forward slicing was implemented using the Frama-C source code analysis platform for code transformation [33]. Forward slicing was based on the construction of the PDG from the location of the EDM to be relocated [21]. EDM relocation could then take place by updating the values in the conjunctive rules of the predicate according to the statements captured by the PDG. This update guided by identified dependencies is akin to abstract interpretation, an approach that has been shown to provide an effective EDM design mechanism [23]. Although computationally expensive, applying this approach to forward slicing ensures that the results are representative of what comprehensive program slicing can afford. Where there is a desire to relocate EDMs without such comprehensive analysis, it is possible to apply lightweight forward slicing approaches or to restrict the PDG to variables incorporated by an existing EDM, though it should be noted that the former could impair the relocated EDM.

6. RESULTS

Sections 6.1-6.5 present results for the error detection predicates learnt under varying fault models. For each of the machine learning algorithms applied, results are presented for the error detection predicates generated following optimisation. All optimisation performed in this paper was achieved through varying parameters that are independent of any data mining algorithm, i.e., the data set sampling levels applied prior to model generation. This ensures that the same refinement process can be applied regardless of the machine learning algorithm applied. In the tables presented in Sections 6.1-6.6, the FPR and TPR columns give the mean false positive and true positive rates taken across ten cross validations. A true positive corresponds to a model correctly identifying a failure-inducing state, whereas a false positive corresponds to a model incorrectly detecting a state as being failure-inducing. The AUC column shows the area under the ROC curve, the aggregate measure of efficiency that balances the consideration of TRP against FPR. The Var column gives the AUC variance across all ten cross validations, providing an indication of how consistently efficient models were generated.

6.1. BF Fault Model

Evaluating error detection predicates generated under the BF fault model provides a meaningful benchmark for the consideration of simultaneous fault models. It is desirable for error detection predicates generated under simultaneous fault models to maintain the high efficiency and low variance that are associated with error detection predicates generated by a commensurate mechanism under a non-simultaneous fault model.

Table 2 demonstrates levels of efficiency that are commensurate with those observed when using decision tree induction and rule induction to generate error detection predicates [7]. The hallmarks of these algorithms for predicate generation can be seen in the consistently high AUC values and low variance model generation, the former ranging from 0.89161 to 0.99991 for decision tree induction and from 0.88873 to 0.99780 for rule induction. Decision tree induction is the better performing of the two model generation algorithms, with markedly higher TPR in most cases, another observation that is commensurate with existing work in machine learning for error detection predicate generation. An AUC of 0.90411 or higher can be found for every module subject to analysis, an indication that the predicates generated are effective classifiers for failure inducing states. It can also be observed that, whilst some generated detectors were perfect with respect to accuracy ($TPR = 1$) and some perfect with respect to completeness ($FPR = 0$), no perfect detector ($TRP = 1$, $FPR = 0$) was generated.

6.2. FuzzFuzz Fault Model

The FuzzFuzz model is the first simultaneous fault model to be analysed. The space of possible fault injections under the FuzzFuzz fault model makes exhaustive injections impractical, since the number of possible injection values for a single n-bit variable v is 2^n. As such, experimentation was restricted to a fixed number of fault injections for each combination of variables simultaneously targeted. Tables 3 and 4 present results where the number of fault injections for each combination of variables simultaneously targeted is 30 and 100 respectively.

The error detection predicate efficiencies shown in Table 3 would be inappropriate for implementation in an EDM, since the highest AUC value across all models is 0.81939. However, Table 4 demonstrates that it is possible to generate efficient error detection predicates under a simultaneous fuzzing model. It is notable that the efficiencies of these error detection predicates are below those observed under the BF fault model, both in this paper and in existing research, with an aggregate mean AUC of 0.90493.

The efficiencies shown in Tables 3 and 4 should not be considered to reflect poorly on the efficacy of simultaneous fault models in general, since the set of injected faults associated with FuzzFuzz will result in greater perturbation of software state than under other models. Intuitively, the impact of fuzzing for a fixed number of repeats is to incur a less structured and thorough exploration of erroneous software state, thus making it more difficult for any machine learning algorithm to discern the relationship between erroneous software state and system failure. This intuition is substantiated by the improvement that can be seen in the efficiencies of the error detection predicates generated using a larger number of fault injection experiments for each combination of target variables.

6.3. SimBF Fault Model

The SimBF fault model is the most computationally expensive set of experiments presented in this paper, since bit flip fault injection was applied to exhaustive combinations of bits in each variable representation across exhaustive combinations of variables. Whilst polynomial in bit representation and the number of variables, conducting this number of experiments in the development of most software systems is impractical. Despite this, the associated efficiencies are the strongest presented and warrant consideration regardless of the cost incurred.

Table 5 shows that a perfect error detection predicate was generated under the SimBF fault model. The perfect detector is associated with module FG-2, though the error detection predicated associated with several other modules, most notably IM-3 and MG-1, are near perfect. The aggregate mean AUC for decision tree induction is 0.98861, meaning that it is, once again,

TABLE 2: The efficiency of error detection predicates generated and evaluated under the BF fault model.

		Decision Tree Induction				Rule Induction			
Software	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.99849	0.00100	0.99875	6E-07	0.96456	0.00157	0.98150	7E-04
	2	0.99914	0.00009	0.99953	8E-08	0.98554	0.01241	0.98657	1E-05
	3	0.99826	0.00002	0.99912	2E-09	0.93912	0.07671	0.93120	6E-06
	4	0.95422	0.00210	0.97606	5E-04	0.94685	0.06631	0.94027	5E-05
	5	0.96010	0.00090	0.97960	5E-07	0.93022	0.06467	0.93278	1E-04
FG	1	0.79633	0.01311	0.89161	2E-05	0.94151	0.09568	0.92291	5E-04
	2	0.99982	0.00000	0.99991	2E-10	0.98244	0.00420	0.98912	5E-05
	3	0.99662	0.00111	0.99776	8E-08	0.98786	0.00033	0.99376	8E-05
	4	0.93889	0.00235	0.96827	4E-06	0.87776	0.00677	0.93550	4E-02
	5	0.94427	0.04322	0.94350	4E-04	0.92419	0.01097	0.95661	8E-04
IM	1	0.83867	0.00633	0.91617	7E-04	0.81423	0.00766	0.90329	9E-03
	2	0.86937	0.02012	0.92463	9E-05	0.82677	0.02657	0.90010	4E-03
	3	0.94789	0.00091	0.97349	1E-04	0.86754	0.00675	0.93040	5E-02
	4	0.93159	0.00459	0.96350	1E-03	0.82377	0.00950	0.90714	1E-05
	5	0.91831	0.00842	0.95495	5E-03	0.84434	0.00905	0.91765	4E-02
MG	1	1.00000	0.00990	0.99505	1E-12	0.97130	0.00001	0.98565	4E-05
	2	0.97403	0.00000	0.98702	1E-32	0.99559	0.00000	0.99780	9E-06
	3	0.99380	0.00000	0.99690	1E-32	0.90587	0.04206	0.93190	7E-07
	4	0.82290	0.01469	0.90411	3E-07	0.81036	0.00177	0.90430	2E-05
	5	0.85073	0.00349	0.92362	1E-04	0.79360	0.01614	0.88873	7E-02
		0.93667	**0.00662**	**0.96468**	**4E-04**	**0.90667**	**0.02296**	**0.94186**	**1E-02**

TABLE 3: The efficiency of error detection predicates generated and evaluated under the FuzzFuzz fault model with 30 fault injection experiments for each combination of target variables.

		Decision Tree Induction				Rule Induction			
System	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.64430	0.17731	0.73350	5E-02	0.55872	0.28880	0.63496	7E-03
	2	0.59887	0.23541	0.68173	7E-02	0.51426	0.28823	0.61302	2E-03
	3	0.54452	0.04431	0.75011	9E-04	0.52676	0.04493	0.74092	7E-03
	4	0.63089	0.08624	0.77233	9E-02	0.50364	0.08907	0.70729	9E-03
	5	0.56538	0.00793	0.77873	1E-02	0.50114	0.00960	0.74577	9E-03
FG	1	0.67200	0.04847	0.81177	3E-02	0.53404	0.03964	0.74720	6E-03
	2	0.61528	0.00626	0.80451	5E-03	0.53584	0.01239	0.76173	6E-03
	3	0.52155	0.06680	0.72738	1E-02	0.51829	0.06821	0.72504	9E-04
	4	0.69746	0.08621	0.80563	6E-02	0.58829	0.08780	0.75025	7E-03
	5	0.53417	0.00917	0.76250	1E-03	0.51322	0.04446	0.73438	4E-03
IM	1	0.61181	0.00258	0.80462	5E-02	0.51907	0.04399	0.73754	6E-02
	2	0.61151	0.04591	0.78280	8E-03	0.60607	0.04972	0.77818	4E-03
	3	0.64896	0.01207	0.81845	2E-02	0.54398	0.06983	0.73708	5E-03
	4	0.66317	0.02439	0.81939	3E-03	0.53377	0.04608	0.74385	5E-03
	5	0.62518	0.00411	0.81054	8E-04	0.50813	0.00966	0.74924	1E-03
MG	1	0.53590	0.00987	0.76302	1E-03	0.50873	0.03244	0.73815	9E-03
	2	0.63777	0.00879	0.81449	5E-04	0.54761	0.00956	0.76903	8E-02
	3	0.55767	0.01651	0.77058	1E-03	0.51580	0.12764	0.69408	4E-03
	4	0.57714	0.03535	0.77090	8E-03	0.52553	0.08434	0.72060	4E-02
	5	0.52629	0.00890	0.75870	1E-02	0.50150	0.01485	0.74333	5E-02
		0.60099	**0.04683**	**0.77708**	**2E-02**	**0.53022**	**0.07306**	**0.72858**	**2E-02**

the better performing of the two model generation algorithms. This is higher than the 0.93667 recorded under the BF fault model, despite SimBF being the stronger of the two fault models in terms of the set of faults imposed / perturbation of software state. Supporting preliminary findings in [7], this is an indication that a comprehensive and systematic exploration of erroneous software states is fundamental to the generation of efficient error detection predicates using machine learning.

6.4. Simultaneous Fault Model Efficacy

Although simultaneous fault models are proposed to be more representative and cross validation allows the efficiency of the generated error detection predicates

TABLE 4: The efficiency of error detection predicates generated and evaluated under the FuzzFuzz fault model with 100 fault injection experiments for each combination of target variables.

		Decision Tree Induction				**Rule Induction**			
System	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.92298	0.08920	0.91689	7E-05	0.85316	0.00538	0.92389	9E-02
	2	0.89798	0.00161	0.94819	3E-04	0.81497	0.00887	0.90305	2E-02
	3	0.84798	0.00944	0.91927	2E-05	0.79370	0.01118	0.89126	6E-01
	4	0.89089	0.06959	0.91065	4E-03	0.79718	0.00204	0.89757	9E-03
	5	0.83002	0.00933	0.91035	3E-03	0.78101	0.00145	0.88978	7E-02
FG	1	0.75158	0.02956	0.86101	5E-06	0.77264	0.01997	0.87634	9E-02
	2	0.88952	0.00260	0.94346	1E-06	0.75132	0.00987	0.87073	1E-02
	3	0.91492	0.06680	0.92406	9E-07	0.83180	0.06126	0.88527	2E-03
	4	0.89191	0.00876	0.94158	5E-05	0.88171	0.03874	0.92149	1E-02
	5	0.83837	0.00714	0.91562	7E-05	0.80233	0.02890	0.88672	1E-02
IM	1	0.78233	0.00126	0.89054	7E-02	0.81423	0.04303	0.88560	2E-01
	2	0.71151	0.00638	0.85257	2E-03	0.82677	0.02987	0.89845	9E-02
	3	0.83389	0.00342	0.91524	2E-03	0.86754	0.06319	0.90218	6E-02
	4	0.73310	0.02439	0.85436	8E-04	0.82377	0.04415	0.88981	2E-02
	5	0.81016	0.00330	0.90343	2E-02	0.84434	0.00750	0.91842	9E-04
MG	1	0.82051	0.00583	0.90734	1E-05	0.97130	0.00761	0.98185	7E-02
	2	0.84187	0.00035	0.92076	1E-03	0.99559	0.00319	0.99620	5E-03
	3	0.77253	0.00716	0.88269	4E-03	0.90587	0.09489	0.90549	2E-02
	4	0.76022	0.01853	0.87085	1E-03	0.81036	0.03768	0.88634	1E-02
	5	0.82237	0.00299	0.90969	2E-03	0.79360	0.01193	0.89084	4E-02
		0.82823	**0.01838**	**0.90493**	**6E-03**	**0.83666**	**0.02654**	**0.90506**	**7E-02**

TABLE 5: The efficiency of error detection predicates generated and evaluated under the SimBF fault model.

		Decision Tree Induction				**Rule Induction**			
System	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.99988	0.00005	0.99992	2E-14	0.96484	0.00063	0.98211	2E-02
	2	0.99974	0.00003	0.99986	9E-08	0.99744	0.00670	0.99537	9E-03
	3	0.99964	0.00001	0.99982	5E-08	0.94111	0.00948	0.96582	3E-02
	4	0.98831	0.00031	0.99400	1E-05	0.95326	0.00311	0.97508	2E-03
	5	0.97990	0.00078	0.98956	3E-03	0.95828	0.03180	0.96324	1E-03
FG	1	0.89734	0.00433	0.94651	1E-06	0.94151	0.00136	0.97008	7E-03
	2	1.00000	0.00000	1.00000	1E-19	0.98655	0.00360	0.99148	5E-02
	3	0.99710	0.00057	0.99827	7E-09	0.99121	0.00020	0.99551	3E-02
	4	0.94179	0.00167	0.97006	5E-06	0.91728	0.00310	0.95709	1E-03
	5	0.96588	0.04672	0.95958	7E-05	0.93679	0.00224	0.96728	8E-02
IM	1	0.97914	0.00633	0.98641	4E-06	0.90968	0.00243	0.95363	9E-02
	2	0.97927	0.00040	0.98944	7E-05	0.89180	0.00241	0.94470	6E-02
	3	1.00000	0.00006	0.99997	1E-32	0.92153	0.00547	0.95803	3E-02
	4	0.98926	0.00459	0.99234	3E-03	0.86372	0.08034	0.89169	9E-03
	5	0.99956	0.00031	0.99963	1E-11	0.93834	0.00635	0.96600	2E-02
MG	1	1.00000	0.00004	0.99998	1E-32	0.98766	0.00043	0.99362	2E-03
	2	0.99855	0.00000	0.99928	1E-32	0.99598	0.00035	0.99782	9E-04
	3	0.99964	0.00000	0.99982	1E-32	0.91953	0.00101	0.95926	3E-02
	4	0.94480	0.00235	0.97123	4E-03	0.80133	0.00338	0.89898	8E-02
	5	0.95439	0.00126	0.97657	2E-03	0.79408	0.01217	0.89096	3E-02
		0.98071	**0.00349**	**0.98861**	**6E-04**	**0.93060**	**0.00883**	**0.96089**	**3E-02**

to be evaluated, gaining insight into the efficacy of simultaneous fault models is challenging. It is natural to consider the extent to which the error detection predicates generated under a simultaneous fault model can account for the set of faults injected under a non-simultaneous model, since the latter is commonly used to inform EDM design. This is achieved by determining whether the error detection predicates generated under the SimBF fault model account for the faults injected under the BF fault model. These models are related, in that the set of faults injected under the BF fault model is a strict subset of the set of faults injected under the SimBF fault model.

The unsampled BF data set was evaluated against the best performing error detection predicates generated under the SimBF fault model. Table 6 shows the efficiency of error detection predicates generated under the SimBF fault model and evaluated under the BF fault model. The efficiencies of the error detection predicates generated under the SimBF fault model when confronted by the set of faults associated with the BF fault model demonstrate the utility of simultaneous fault models. Once again, the more comprehensive exploration of erroneous state under the SimBF fault model enables the resultant error detection predicates to be more accurate and complete than those generated under the BF fault model. Most notably, at least one perfect detector has been generated for at least one module in each software system.

Simultaneous fault models aim to be more representative of faults that occur in real-world software. Whilst it can not be argued that results presented in Table 6 further the argument of representativeness beyond what is shown in [13], the results are a strong indication that simultaneous fault model provides considerations over and above the widely used BF model.

6.5. Restricted SimBF Fault Model

Having demonstrated the set of fault injections associated with BF is accounted for under SimBF, it is reasonable that SimBF could be used in fault injection analysis, not least with regard to the transient and permanent hardware faults that BF is commonly used to emulate. However, the computational expense of performing bit flip fault injection on exhaustive combinations of bits in each variable representation makes the model impractical for many software validation processes.

The problem of impractical experimental cost can be solved by restricting the number of fault injections performed or developing a strategy to intelligently sample the error space. To this point simultaneous bit flip fault injections have been exhaustive. That is, if n variables were in scope, the fault model was exhaustively applied to the representation of every k-combination of variables for $1 \leq k \leq n$. By restricting the faults injected to being in every k-combination of bits in the representation for $1 \leq k \leq 2$ and, similarly only k-combinations of variables for $1 \leq k \leq 2$, the number of experiments is dramatically reduced whilst preserving the essence of simultaneous fault injection.

Table 7 shows the performance of the generated error detection predicates when simultaneous bit flip fault injection is restricted to every k-combination of bits in representation for $1 \leq k \leq 2$ and k-combinations of variables for $1 \leq k \leq 2$. This reduces the number of fault injection experiments on a single variable from $\binom{n}{1} + \binom{n}{2} + ... + \binom{n}{k}$ to $\binom{n}{1} + \binom{n}{2}$, where n is the number of bits in the variable. Similarly reducing the number of variable combinations to $\binom{n}{1} + \binom{n}{2} + \binom{n}{3} + ... + \binom{n}{k}$ to $\binom{n}{1} + \binom{n}{2}$, where n is the number of variables in scope.

The performance of the error detection predicates generated under the SimBF model with 2 or fewer simultaneous faults, shown in Table 7, is identical to the performance of the exhaustive SimBF model for all but five software modules. In each of these cases the associated error detection predicates have decreased in TPR and FPR, hence a commensurate reduction in AUC, though the impact is less severe where the error detection predicates are generated using rule induction.

The TPR, FPR and AUC values of the five impacted software modules do not revert to the efficiencies of the BF model, despite the restricted model retaining near-perfect detection capability with regard to single fault injections. This represents a further indication that the consideration of the simultaneous fault model is providing greater depth of analysis, now at a more reasonable experimental cost. It is similarly interesting to note the consistency of the AUC variance across the exhaustive and restricted models, suggesting near identical error detection predicates are being generated, despite the cross validation process excluding informative instances for evaluation. This is consistent with the view that capturing the correctness of a basis set of program variables is sufficient to provide efficient error detection [34].

Table 8 shows the performance of the generated error detection predicates when simultaneous bit flip fault injection is restricted to every k-combination for $1 \leq k \leq 3$. This increases the number of fault injection experiments conducted for a single variable from $\binom{n}{1} + \binom{n}{2}$ to $\binom{n}{1} + \binom{n}{2} + \binom{n}{3}$, where n is the number of bits in the representation. Similarly increasing the number of variables combinations to $\binom{n}{1} + \binom{n}{2}$ to $\binom{n}{1} + \binom{n}{2} + \binom{n}{3}$, where n is the number of variables in scope.

The performance of the error detection predicates generated under SimBF with three or fewer simultaneous faults, shown in Table 8, builds on the performance of the model with two or fewer simultaneous faults. The restricted model with three or fewer simultaneous faults yields identical performance to the exhaustive SimBF fault model for all but two software modules, where these are limited to a single software system.

6.6. Relocated EDM Efficiency

As in Section 5, the most efficient EDMs in each software module generated were relocated to the exit-point of the module using forward slicing. Table 9 shows the efficiency of error detection predicates generated, relocated and evaluated under the SimBF fault model.

Table 9 shows one case of decreased TPR and five cases where FPR has increased, explained by the fact that abstract interpretation over-approximates system behaviour [23]. The efficiency properties of most EDMs remain unchanged and the maximum impact of any impairment is a 0.00601 decrease in AUC. There is no apparent difference between the magnitude of the impact of relocation across the decisions tree induction and rule induction algorithms. Table 9 demonstrates that relocating efficient EDMs is a viable alternative

TABLE 6: The efficiency of error detection predicates generated under the SimBF fault model and evaluated against BF data.

		Decision Tree Induction				Rule Induction			
System	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.99996	0.00000	0.99992	2E-15	0.99981	0.00007	0.99987	4E-08
	2	1.00000	0.00000	1.00000	1E-32	1.00000	0.00110	0.99945	2E-09
	3	1.00000	0.00000	1.00000	1E-32	1.00000	0.00000	1.00000	1E-32
	4	0.99940	0.00002	0.99969	1E-32	0.99944	0.00050	0.99947	6E-15
	5	0.99992	0.00048	0.99972	4E-29	0.98199	0.00104	0.99048	1E-04
FG	1	0.99866	0.00700	0.99583	4E-31	0.99711	0.00484	0.99614	2E-04
	2	1.00000	0.00000	1.00000	1E-32	1.00000	0.00000	1.00000	1E-32
	3	1.00000	0.00000	1.00000	1E-32	1.00000	0.00048	0.99976	2E-17
	4	0.99990	0.00092	0.99949	8E-16	0.99990	0.00110	0.99940	4E-09
	5	1.00000	0.00112	0.99944	9E-06	1.00000	0.00099	0.99951	3E-16
IM	1	0.99931	0.00031	0.99950	4E-06	0.99811	0.00031	0.99890	7E-02
	2	0.99902	0.00000	0.99951	7E-05	0.99703	0.00000	0.99852	4E-09
	3	1.00000	0.00000	1.00000	1E-32	1.00000	0.00060	0.99970	9E-15
	4	1.00000	0.00192	0.99904	3E-03	1.00000	0.00422	0.99789	9E-18
	5	1.00000	0.00001	1.00000	1E-11	1.00000	0.00080	0.99960	2E-29
MG	1	1.00000	0.00000	1.00000	1E-32	1.00000	0.00000	1.00000	1E-32
	2	0.99906	0.00000	0.99953	1E-32	0.99900	0.00000	0.99950	9E-08
	3	0.99999	0.00000	0.99999	2E-32	0.99999	0.00012	0.99994	3E-06
	4	0.99917	0.00200	0.99859	8E-08	0.99907	0.00280	0.99814	2E-31
	5	0.99909	0.00062	0.99924	5E-16	0.99890	0.00082	0.99904	1E-04
		0.99967	**0.00072**	**0.99947**	**2E-04**	**0.99852**	**0.00099**	**0.99877**	**4E-03**

TABLE 7: The efficiency of error detection predicates generated and evaluated under the SimBF fault model with simultaneous fault injections restricted to k-combinations of bits in representation and variables for $1 \leq k \leq 2$.

		Decision Tree Induction				Rule Induction			
System	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.99988	0.00005	0.99992	2E-14	0.96484	0.00063	0.98211	2E-02
	2	1.00000	0.00000	1.00000	1E-19	0.98655	0.00360	0.99148	5E-02
	3	0.99710	0.00057	0.99827	7E-09	0.99121	0.00020	0.99551	3E-02
	4	0.96205	0.00170	0.98018	8E-06	0.94901	0.00311	0.97295	7E-03
	5	0.97990	0.00078	0.98956	3E-03	0.95828	0.03180	0.96324	1E-03
FG	1	0.88218	0.00855	0.93682	1E-06	0.94151	0.00136	0.97008	7E-03
	2	1.00000	0.00000	1.00000	1E-19	0.98655	0.00360	0.99148	5E-02
	3	0.99710	0.00057	0.99827	7E-09	0.99121	0.00020	0.99551	3E-02
	4	0.94004	0.00218	0.97006	4E-03	0.90948	0.00610	0.95709	8E-03
	5	0.96588	0.04672	0.95958	7E-05	0.93679	0.00224	0.96728	8E-02
IM	1	0.97914	0.00633	0.98641	4E-06	0.90968	0.00243	0.95363	9E-02
	2	0.97927	0.00040	0.98944	7E-05	0.89180	0.00241	0.94470	6E-02
	3	1.00000	0.00080	0.99997	4E-32	0.92153	0.00547	0.95803	3E-02
	4	0.98926	0.00459	0.99234	3E-03	0.86372	0.08034	0.89169	9E-03
	5	0.99956	0.00031	0.99963	1E-11	0.93834	0.00635	0.96600	2E-02
MG	1	1.00000	0.00004	0.99998	1E-32	0.98766	0.00043	0.99362	2E-03
	2	0.99855	0.00000	0.99928	1E-32	0.99598	0.00035	0.99782	9E-04
	3	0.99964	0.00000	0.99982	1E-32	0.91953	0.00101	0.95926	3E-02
	4	0.92249	0.00238	0.96006	5E-02	0.80100	0.00348	0.89876	8E-02
	5	0.90100	0.00157	0.97657	2E-03	0.79408	0.01217	0.89096	3E-02
		0.97465	**0.00388**	**0.98681**	**3E-03**	**0.93194**	**0.00836**	**0.96206**	**3E-02**

to regeneration, since the efficiencies of EDMs are not substantially impaired following relocation. This result reduces the resource costs of the experimentation associated with EDM design because it obviates the need to generate individual EDMs for new locations, a process that has been shown to be computationally expensive. Instead, EDMs that are known to be efficient can be relocated. Combined with the results presented in Section 6.5, it may be desirable to design a single efficient EDM under a restricted fault model, thereby reducing its design cost, and proceeding to relocate to EDM to multiple locations across a software system, removing the need for further design cost.

It is noteworthy that all relocated EDMs with

TABLE 8: The efficiency of error detection predicates generated and evaluated under the SimBF fault model with simultaneous fault injections restricted to k-combinations of bits in representation and variables for $1 \leq k \leq 3$.

		Decision Tree Induction				Rule Induction			
System	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.99988	0.00005	0.99992	4E-12	0.96484	0.00063	0.98211	2E-06
	2	0.99974	0.00003	0.99986	6E-07	0.99744	0.00670	0.99537	6E-04
	3	0.99964	0.00001	0.99982	5E-08	0.94111	0.00948	0.96582	3E-02
	4	0.98831	0.00031	0.99400	1E-05	0.95326	0.00311	0.97508	2E-03
	5	0.97990	0.00078	0.98956	3E-03	0.95828	0.03180	0.96324	1E-03
FG	1	0.89699	0.00655	0.94522	1E-06	0.94151	0.00136	0.97008	7E-03
	2	1.00000	0.00000	1.00000	1E-19	0.98655	0.00360	0.99148	5E-02
	3	0.99710	0.00057	0.99827	7E-09	0.99121	0.00020	0.99551	3E-02
	4	0.94004	0.00218	0.97006	5E-06	0.90948	0.00610	0.95709	1E-03
	5	0.96588	0.04672	0.95958	7E-05	0.93679	0.00224	0.96728	8E-02
IM	1	0.97914	0.00633	0.98641	4E-06	0.90968	0.00243	0.95363	9E-02
	2	0.97927	0.00040	0.98944	7E-05	0.89180	0.00241	0.94470	6E-02
	3	1.00000	0.00080	0.99997	4E-32	0.92153	0.00547	0.95803	3E-02
	4	0.98926	0.00459	0.99234	3E-03	0.86372	0.08034	0.89169	9E-03
	5	0.99956	0.00031	0.99963	1E-11	0.93834	0.00635	0.96600	2E-02
MG	1	1.00000	0.00004	0.99998	1E-32	0.98766	0.00043	0.99362	2E-03
	2	0.99855	0.00000	0.99928	1E-32	0.99598	0.00035	0.99782	9E-04
	3	0.99964	0.00000	0.99982	1E-32	0.91953	0.00101	0.95926	3E-02
	4	0.94480	0.00235	0.97123	4E-03	0.80133	0.00338	0.89898	8E-02
	5	0.95439	0.00126	0.97657	2E-03	0.79408	0.01217	0.89096	3E-02
		0.98060	**0.00366**	**0.98855**	**6E-04**	**0.93021**	**0.00897**	**0.96089**	**3E-02**

TABLE 9: The efficiency of error detection predicates generated, relocated and evaluated under the SimBF fault model.

		Decision Tree Induction				Rule Induction			
System	**Module**	**TPR**	**FPR**	**AUC**	**Var**	**TPR**	**FPR**	**AUC**	**Var**
7Z	1	0.99988	0.00005	0.99992	4E-12	0.96484	0.00063	0.98211	2E-06
	2	0.99974	0.00003	0.99986	6E-07	0.99744	0.00670	0.99537	6E-04
	3	0.99964	0.00001	0.99982	5E-08	0.94111	0.00948	0.96582	3E-02
	4	0.98831	0.00031	0.99400	1E-05	0.95326	0.00311	0.97508	2E-03
	5	0.97990	0.00078	0.98956	3E-03	0.95828	0.03180	0.96324	1E-03
FG	1	0.88849	0.00750	0.94050	1E-04	0.94151	0.00136	0.97008	7E-03
	2	1.00000	0.00826	0.99587	2E-16	0.98655	0.00360	0.99148	5E-02
	3	0.99710	0.00057	0.99827	7E-09	0.99121	0.00020	0.99551	3E-02
	4	0.94179	0.00284	0.96948	1E-04	0.90948	0.00722	0.95113	3E-03
	5	0.96588	0.04672	0.95958	7E-05	0.93679	0.00224	0.96728	8E-02
IM	1	0.97914	0.00633	0.98641	4E-06	0.90968	0.00243	0.95363	9E-02
	2	0.97927	0.00040	0.98944	7E-05	0.89180	0.00241	0.94470	6E-02
	3	1.00000	0.00207	0.99897	1E-32	0.92153	0.00667	0.95743	2E-02
	4	0.98926	0.00459	0.99234	3E-03	0.86372	0.08034	0.89169	9E-03
	5	0.99956	0.00031	0.99963	1E-11	0.93834	0.00635	0.96600	2E-02
MG	1	1.00000	0.00009	0.99996	1E-32	0.98766	0.00188	0.99289	1E-03
	2	0.99855	0.00000	0.99928	1E-32	0.99598	0.00035	0.99782	9E-04
	3	0.99964	0.00000	0.99982	1E-32	0.91953	0.00101	0.95926	3E-02
	4	0.94480	0.00235	0.97123	4E-03	0.80133	0.00338	0.89898	8E-02
	5	0.95439	0.00126	0.97657	2E-03	0.79408	0.01217	0.89096	3E-02
		0.98027	**0.00422**	**0.98803**	**6E-04**	**0.93021**	**0.00916**	**0.96052**	**3E-02**

reduced efficiency are associated with modules where EDM performance was also impaired when a restricted set of variables was used in EDM generation, suggesting that there exist identifiable software modules for which it is inherently more difficult to generate efficient EDMs.

The results presented demonstrate that the Restricted SimBF model provides a practical simultaneous fault model for the generation of efficient EDMs based on the application of machine learning to software fault injection data sets. The efficiencies of the generated error detection predicates surpass those of predicates generated under the BF model when evaluated against non-simultaneous or simultaneous faults. It should be noted that the results presented were derived under synthetic workloads and simultaneous fault models developed by existing research, a common limitation of work in the design of error and anomaly detection approaches.

7. CONCLUSION

In this section the contributions made in this paper are summarised and future work discussed.

7.1. Summary

The application of machine learning to fault injection data has been shown to be amongst the most effective approaches for the generation of efficient EDMs. However, such approaches to design of EDMs have invariably adopted a fault model with a single-fault assumption. Although simultaneous faults do not necessarily exist in all scenarios, the single-fault assumption limits the practical relevance of research in fault injection and fails to recognise the demands of established safety standards. This paper addresses this problem, demonstrating that efficient EDMs can be generated using fault injection data collected under fault models accounting for simultaneous faults. In particular, it is shown that (i) efficient EDMs can be designed using fault data collected under models accounting for the occurrence of simultaneous faults, (ii) exhaustive fault injection under a simultaneous bit flip model can yield improvements to EDM efficiency, and (iii) exhaustive fault injection under a simultaneous bit flip model can be made non-exhaustive and (iv) EDMs can be relocated within a software system using program slicing, reducing the resource costs of experimentation to practicable levels without sacrificing EDM efficiency.

7.2. Future Work

The results presented motivate further consideration of simultaneous fault model representativeness. The examination of fault injection data and efficient error detection predicates each serve as a means for gaining insight into fault model representativeness. In contrast, the task of sampling error states and test cases to reduce experimental cost is well explored. Despite this, existing methods for error state and test case sampling invariably require domain knowledge. Examining the software states captured by efficient error detection predicates could provide insight into how to better sample error spaces and test cases, such that designed EDMs are inherently efficient and amenable to relocation within a software system.

8. ACKNOWLEDGEMENTS

This work was supported by The Alan Turing Institute under the EPSRC grant EP/N510129/1.

REFERENCES

[1] Arora, A. and Kulkarni, S. S. (1998) Detectors and correctors: A theory of fault-tolerance components. *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, Amsterdam, Netherlands, May, pp. 436–443. IEEE Computer Society.

[2] Powell, D., Martins, E., Arlat, J., and Crouzet, Y. (1995) Estimators for fault tolerance coverage evaluation. *IEEE Transactions on Computers*, **44**, 261–274.

[3] Jhumka, A., Freiling, F., Fetzer, C., and Suri, N. (2006) An approach to synthesise safe systems. *International Journal of Security and Networks*, **1**, 62–74.

[4] Thomas, A. and Pattabiraman, K. (2013) Error detector placement for soft computation. *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, Budapest, Hungary, June, pp. 1–12. IEEE.

[5] Jhumka, A. and Leeke, M. (2009) Issues on the design of efficient fail-safe fault tolerance. *Proceedings of the 20th IEEE International Symposium on Software Reliability Engineering*, Mysuru, India, November, pp. 155–164. IEEE Computer Society.

[6] Candea, G., Delgado, M., Chen, M., and Fox, A. (2003) Automatic failure-path inference: A generic introspection technique for internet applications. *Proceedings of the 3rd IEEE Workshop on Internet Applications (WIAPP 2003)*, California, USA, June, pp. 132–141. IEEE.

[7] Leeke, M., Jhumka, A., and Anand, S. S. (2013) Towards the design of efficient error detection mechanisms for transient data errors. *The Computer Journal*, **56**, 674–692.

[8] Volzer, H. (1998) Verifying fault tolerance of distributed algorithms formally - an example. *Proceedings of the 1st International Conference on the Application of Concurrency to System Design*, Fukushima, Japan, March, pp. 187–197. IEEE Computer Society.

[9] Duarte, A., Cirne, W., Brasileiro, F., and Machado, P. (2006) Gridunit: Software testing on the grid. *Proceedings of the 28th ACM/IEEE International Conference on Software Engineering*, Shanghai, China, May, pp. 779–782. ACM.

[10] Leeke, M. and Jhumka, A. (2009) Evaluating the use of reference run models in fault injection analsyis. *Proceedings of the 15th Pacific Rim International Symposium on Dependable Computing*, Shanghai, China, November, pp. 121–124. IEEE.

[11] Natella, R., Cotroneo, D., Duraes, J. A., and Madeira, H. S. (2013) On fault representativeness of software fault injection. *IEEE Transactions on Software Engineering*, **39**, 80–96.

[12] Powell, D. (1992) Failure model assumptions and assumption coverage. *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, Wisconsin, USA, July, pp. 386–395. IEEE Computer Society.

[13] Winter, S., Tretter, M., Sattler, B., and Suri, N. (2013) simFI: From single to simultaneous software fault injections. *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks*, Budapest, Hungary, June, pp. 1–12. IEEE.

[14] Winter, S., Schwahn, O., Natella, R., Suri, N., and Cotroneo, D. (2015) No pain, no gain? the utility of parallel fault injections. *Proceedings of the 37th IEEE/ACM International Conference on Software Engineering*, Florence, Italy, May, pp. 494–505. IEEE.

[15] Jhumka, A., Hiller, M., and Suri, N. (2004) An approach for designing and assessing detectors for dependable component-based systems. *Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering*, Florida, USA, March, pp. 69–78. IEEE Computer Society.

[16] Lanzaro, A., Natella, R., Winter, S., Cotroneo, D., and Suri, N. (2014) An empirical study of injected versus actual interface errors. *Proceedings of the 23rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, California, USA, July, pp. 397–408. ACM.

[17] Moraes, R., Barbosa, R., Duraes, J., Mendes, N., Martins, E., and Madeira, H. (2006) Injection of faults at component interfaces and inside the component code: are they equivalent? *Proceedings of the 6th European Dependable Computing Conference*, Coimbra, Portugal, October, pp. 53–64. IEEE.

[18] Johansson, A., Suri, N., and Murphy, N. (2007) On the selection of error model(s) for os robustness evaluation. *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks*, Edinburgh, UK, June, pp. 502–511. IEEE.

[19] Japkowicz, N. (2000) The class imbalance problem: Significance and strategies. *Proceedings of the 2nd International Conference on Artificial Intelligence*, Nevada, USA, June, pp. 111–117. IEEE Computer Society.

[20] Weiser, M. (1981) Program slicing. *Proceedings of the 5th International Conference on Software Engineering*, California, USA, March, pp. 439–449. ACM.

[21] Ferrante, J., Ottenstein, K. J., and Warren, J. D. (1987) The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, **9**, 319–349.

[22] Agrawal, H. and Horgan, J. R. (1990) Dynamic program slicing. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementationonference on Programming language design and implementation*, New York, USA, June, pp. 246–256. ACM.

[23] Jhumka, A., Hiller, M., Claesson, V., and Suri, N. (2002) On systematic design of globally consistent executable assertions in embedded software. *ACM SIGPLAN Notices*, **37**, 75–84.

[24] Leeke, M. (2017) Simultaneous fault models for the generation of efficient error detection mechanisms. *Proceedings of the 28th IEEE International Symposium on Software Reliability Engineering*, Toulouse, France, November, pp. 112 – 123. IEEE.

[25] 7-Zip (2018). http://www.7-zip.org/.

[26] FlightGear (2018). http://www.flightgear.org/.

[27] MP3Gain (2018). http://mp3gain.sourceforge.net/.

[28] ImageMagick (2018). https://www.imagemagick.org.

[29] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009) The weka data mining software: An update. *SIGKDD Explorations*, **11**, 10–18.

[30] Chawla, N. V., Cieslak, D. A., Hall, L. O., and Joshi, A. (2008) Automatically countering imbalance and its empirical relationship to cost. *Journal of Data Minining and Knowledge Discovery*, **17**, 225–252.

[31] Cohen, W. W. (1995) Fast effective rule induction. *Proceedings of the 12th International Conference on Machine Learning*, California, USA, July, pp. 115–123. Morgan Kaufmann.

[32] Quinlan, J. R. (1992) *C4.5: Programs for Machine Learning*. Morgan Kaufmann, California, USA.

[33] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B. (2015) Frama-c: A software analysis perspective. *Formal Aspects of Computing*, **27**, 573–609.

[34] Fairbrother, J. and Leeke, M. (2017) On basis variables for efficient error detection. *Proceedings of the 15th IEEE International Conference on Dependable, Autonomic and Secure Computing*, Florida, USA, November, pp. 399 – 406. IEEE.