Check for updates

Received: 28 February 2021 | Revised: 15 May 2021 | Accepted: 28 June 2021

DOI: 10.1049/sfw2.12038

IET Software

The Institution of Engineering and Technology

WILEY

ORIGINAL RESEARCH PAPER

Embedding and classifying test execution traces using neural networks

Foivos Tsimpourlas[1] | Gwenyth Rooijackers[1] | Ajitha Rajan[1] | Miltiadis Allamanis[2]

[1]School of Informatics, University of Edinburgh, Edinburgh, UK

[2]Microsoft Research, Cambridge, UK

Correspondence

Foivos Tsimpourlas, School of Informatics, University of Edinburgh, Edinburgh, UK

Email: F.Tsimpourlas@sms.ed.ac.uk

Funding information

Engineering and Physical Sciences Research Council, Grant/Award Number: EP/L01503X/1; Facebook, Grant/Award Number: Facebook Testing and Verification Award 2018 & 2019

Abstract

Classifying test executions automatically as pass or fail remains a key challenge in software testing and is referred to as the *test oracle problem*. It is being attempted to solve this problem with supervised learning over test execution traces. A programme is instrumented to gather execution traces as sequences of method invocations. A small fraction of the programme's execution traces is labelled with pass or fail verdicts. Execution traces are then embedded as fixed length vectors and a neural network (NN) component that uses the line-by-line information to classify traces as pass or fail is designed. The classification accuracy of this approach is evaluated using subject programs from different application domains—1. Module from Ethereum Blockchain, 2. Module from PyTorch deep learning framework, 3. Microsoft SEAL encryption library components, 4. Sed stream editor, 5. Nine network protocols from Linux packet identifier, L7-Filter and 6. Utilities library, commons-lang for Java. For all subject programs, it was found that test execution classification had high precision, recall and specificity, averaging to 93%, 94% and 96%, respectively, while only training with an average 14% of the total traces. Experiments show that the proposed NN-based approach is promising in classifying test executions from different application domains.

KEYWORDS

execution trace, neural networks, software testing, test oracle

CCS CONCEPTS

• Software and its engineering → Software testing and debugging; • Computing methodologies → *Supervised learning by classification.*

1 | INTRODUCTION

To make software testing faster, cheaper and more reliable, it is desirable to automate as much of the process as possible. Over the past decades, researchers have made remarkable progress in automatically generating effective test inputs [1, 2]. Automated test input generation tools, however, generate substantially more tests than manual approaches. This becomes an issue when determining the correctness of test executions, a procedure referred to as the *test oracle*, which is still largely manual and relies on developer expertise. Recent surveys on the test oracle problem [3–5] show that automated oracles based on formal specifications, metamorphic relations [6] and independent programme versions are not widely applicable and difficult to use in practice.

In our recent work [7], we sought to address the test oracle problem using supervised learning over execution traces of a given system. In particular, we used neural networks (NNs),

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by anyone other than the ACM must be honoured. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org. Conference'17, July 2017, Washington, DC, USA © 2021 Association for Computing Machinery. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00 https://doi.org/10.1145/nnnnnnn.nnnnnnn.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.

© 2021 The Authors. *IET Software* published by John Wiley & Sons Ltd on behalf of The Institution of Engineering and Technology.

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

well suited to learning complex functions and classifying patterns, to design the test oracles. We found this technique to be widely applicable and easy to use, as it only requires execution traces gathered from running test inputs through the programme under test (PUT) to design the oracle. This is shown in Figure 1 where a small fraction of the gathered execution traces labelled with pass/fail (shown in light grey) is used to train the NN model, which is then used to automatically classify the remaining unseen execution traces (coloured dark grey).

Previous work exploring the use of NNs for test oracles has been in a restricted context—applied to very small programs with primitive data types and only considering their inputs and outputs [8, 9]. Information in execution traces, which we believe is useful for test oracles, has not been considered by existing NN-based approaches. Other bodies of work in programme analysis have used NNs to predict method or variable names and detect name-based bug patterns [10, 11] relying on static programme information, namely, embeddings of the Abstract Syntax Tree or source code. Our approach in [7] is the first attempt at using *dynamic execution trace information in NN models for classifying test executions* and has the following steps:

1. Instrument a programme to gather execution traces as sequences of method invocations.
2. Label a small fraction of the traces with their classification decision.
3. Design a NN component that embeds the execution traces to fixed length vectors.
4. Design a NN component that uses the line-by-line trace information to classify traces as pass or fail.
5. Train a NN model that combines the above components and evaluate it on unseen execution traces for that programme.

1.1 | New contributions

The contributions of this study, different from our previous study, are summarised as follows:

1. **Support for Java programs.** Our work in [7] provided tool support in the low level virtual machine (LLVM) [12] framework to instrument the intermediate representation (LLVM-IR) of programmes to gather execution traces. LLVM, however, does not provide front-end support for Java programmes. In this study, we provide tool support to gather execution traces for Java programmes using the Soot framework [13].
2. **Extensive empirical evaluation.** We augment the experiments in [7] with 10 additional subject programmes—9 network protocols from L7-filter [14] and 1 Java utilities library from Defects4J [15], a database of real faults for open-source Java programmes. For these subject programmes, we evaluate the precision, recall and specificity of our approach in classifying execution traces. We also assess the size of the training set needed and compare accuracies against a hierarchical clustering technique for classifying execution traces proposed by Almaghairbe et al. [16].
3. **Generalisation.** We conduct an initial exploration into the ambitious possibility of using a model, trained using traces from one subject programme, to classify traces from other programmes in the same application domain. We use FSMs from the network protocol domain to evaluate this possibility. We found that our approach for designing a NN classification model was effective for all subject programmes. We achieved high accuracies in detecting both failing and passing traces, with an average precision of 93% and recall of 94%. Only a small fraction of the overall traces (average 14%) needed to be labelled for training the classification models. We found that generalisation of a classification model from one network protocol to others in the domain was feasible. Generalisation accuracies were not as high as the accuracy achieved using separate classification models, but we believe there is scope for improvement using fine-tuning in the future.

The study is organised as follows: Section 2 provides background on test oracles and related work in the use of machine learning for test oracles and more generally in software testing. Section 3 presents the algorithms and implementations of our approach including the new tool support for instrumenting Java programmes. Our experimental setup and subject programmes are described in Section 4. Performance of our approach over the different subject programmes and comparison to a state of the art approach is presented in Section 5.

2 | BACKGROUND

When a test oracle observes a test execution, it returns a test verdict, which is either pass or fail depending on whether the observations match expected behaviour. A test execution is

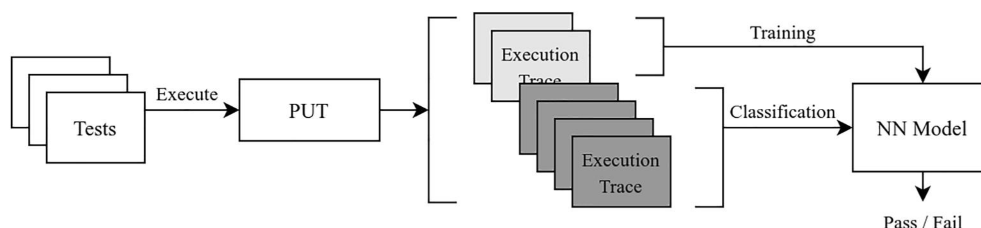

FIGURE 1 Key idea in our approach. NN, neural network; PUT, programme under test

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

the execution of the PUT with a test input. The importance of oracles as an integral part of the testing process has been a key topic of research for over three decades. We distinguish four different kinds of test oracles, based on the survey by Barr et al. in 2015 [3]. The most common form of test oracle is a *specified oracle*, one that judges behavioural aspects of the system under test with respect to formal specifications. Although formal specifications are effective in identifying failures, defining and maintaining such specifications is expensive and also relatively rare in practice. *Implicit* test oracles require no domain knowledge and are easy to obtain at no cost. However, they are limited in their scope as they are only able to reveal particular anomalies like buffer overflows, segmentation faults, and deadlocks. *Derived* test oracles use documentations or system executions, to judge a system's behaviour, when specified test oracles are unavailable. However, derived test oracles, like metamorphic relations and inferring invariants, are either not automated or are inaccurate and irrelevant, making it challenging to use them.

For many systems and much of testing as currently practised in industry, the tester does not have the luxury of formal specifications or assertions or even automated partial oracles [17, 18]. Statistical analysis and machine learning techniques provide a useful alternative for understanding software behaviour using data gathered from a large set of test executions.

2.1 | Machine learning for software testing

Briand et al. [19], in 2008, presented a comprehensive overview of existing techniques that apply machine learning for addressing testing challenges. Among these, the closest related work is that of Bowring et al. in 2004 [20]. They proposed an active learning approach to build a classifier of programme behaviours using a frequency profile of single events in the execution trace. Evaluation of their approach was conducted over one small programme whose specific structure was well suited to their technique. Machine learning techniques have also been used in fault detection. Brun and Ernst, in 2004 [21], explored the use of support vector machines and decision trees to rank programme properties, provided by the user, that are likely to indicate errors in the programme. Podgurski et al., in 2003 [22], used clustering over function call profiles to determine which failure reports are likely to be manifestations of an underlying error. A training step determines which features are of interest by evaluating those that enable a model to distinguish failures from non-failures. The technique does not consider passing runs. In their experiments, most clusters contain failures resulting from a single error.

More recently, Almaghairbe et al. [16] proposed an unsupervised learning technique to classify unlabelled execution traces of simple programmes. They gathered two kinds of execution traces, one with only inputs and outputs and another that includes the sequence of method entry and exit points, with only method names. Arguments and return values are not used. They used agglomerative hierarchical clustering algorithms to build an automated test oracle, assuming that passing traces are grouped into large, dense clusters and failing traces into many small clusters. They evaluated their technique on 3 programmes from the SIR repository [23]. The proposed approach has several limitations. They only support programmes with strings as inputs. They do not consider correct classification of passing traces. The accuracy achieved by the technique is not high, classifying approximately 60% of the failures. Additionally, the fraction of outputs that need to be examined by the developer is around 40% of the total tests, which is considerably higher than the labelled data used in our approach. We objectively compared the accuracy achieved by the hierarchical clustering technique against our approach using 15 PUTs, discussed in Section 5. We found that our approach achieves a significantly higher accuracy in classifying programme executions across all case studies.

Existing work using execution traces for bug detection has primarily been based on clustering techniques. Neural networks, especially with deep learning, have been very successful for complex classification problems in other domains like natural language processing, speech recognition, and computer vision. There is limited work exploring their benefits for software testing problems.

2.1.1 | Neural networks for test oracles

NNs were first used by Vanmali et al. [8] in 2002 to simulate the behaviour of simple programmes using their previous version, and applied this model to regression testing of unchanged functionalities. Aggarwal et al. [24] and Jin et al. [9] applied the same approach to test a triangle classification programme that computes the relationship among three edge inputs to determine the type of triangle. The few existing approaches using NNs have been applied to simple programmes having small I/O domains. The following challenges have not been addressed in existing work:

1. Training with test execution data and their vector representation—Existing work only considers programme inputs and outputs that are of primitive data types (integers, doubles, and characters). Test data for real programmes often use complex data structures and data types defined in libraries. There is a need for techniques that encode such data. In addition, existing work has not attempted to use programme execution information in NNs to classify tests. Achieving this will require novel techniques for encoding execution traces and designing a NN that can learn from them.
2. Test oracles for industrial case studies—Realistic programmes with complex behaviours and input data structures have not been previously explored.
3. Effort for generating labelled training data—Training data in existing work has been over simple programmes, like the

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

triangle classification programme, where labelling the tests was straightforward. Availability of labelled data that includes failing tests has not been previously discussed. Additionally, the proportion of labelled data needed for training and its effect on model prediction accuracy has not been systematically explored.

2.1.2 | Deep learning for software testing

The performance of NNs as classifiers was boosted with the birth of deep learning in 2006 [25]. Deep learning methods have *not* been explored extensively for software testing and, in particular, for the test oracle problem. Recently, a few techniques have been proposed for automatic pattern-based bug detection. For example, Pradel et al. [11] proposed a deep learning-based static analysis for automatic name-based bug detection and Allamanis et al. [26] used graph-based neural static analysis for detecting variable misuse bugs. In addition to these techniques, several other deep learning methods for statically representing the code have been developed [27, 28]. We do not discuss these further since we are interested in execution trace classification and in NNs that use dynamic trace information rather than a static view of the code.

2.1.3 | Embedding execution traces for neural networks

One of the main contributions of this study is an approach for embedding information in execution traces as a fixed length vector to be fed into the NN. There is limited work in using representations of execution traces. Wang et al. [29] proposed embeddings of execution traces in 2017. They used execution traces captured as a sequence of variable values at different programme points. A programme point is when a variable gets updated. Their approach uses recurrent NNs to summarise the information in the execution trace. Embedding of the traces is applied to an existing programme repair tool. The work presented by Wang et al. has several limitations: 1. Capturing execution traces as sequences of updates to every variable in the programme has an extremely high overhead and will not scale to large programmes. The study does not describe how the execution traces are captured, they simply assume they have them. 2. The approach does not discuss how variables of complex data types such as structs, arrays, pointers, and objects are encoded. It is not clear if the traces only capture updates to user-defined variables or if system variables are also taken into account. 3. The evaluation uses three simple, small programmes (e.g. counting parentheses in a string) from students in an introductory programing course. The complexity and scale of real programmes is not assessed in their experiments. Their technique for capturing and directly embedding traces as sequences of updates to every variable is infeasible in real programmes. Our approach captures and embeds traces as sequences of method invocations and updates to global variables, which scales better than tracking every programme variable. We have implemented our instrumentation in the LLVM compiler framework that is language-agnostic and scales to industry-sized programmes. We support all types of variables and objects, including system-defined variables.

Wang et al. [30] use a novel blended approach for learning programme representations with execution traces. In their study, they get a set of symbolic traces from programmes, one for each execution path. They also get concrete traces from programme executions, one for each test input. They create a blended trace by merging one symbolic trace with all concrete traces that exercise this corresponding execution path. They develop a NN architecture called LiGer, an attention-based recurrent NN (RNN). Their model consists of a vocabulary embedding layer, a fusion layer and a programmes embedding layer. The first encodes words to embedding vectors. In the fusion layer, one RNN embeds statements and a second RNN embeds all programme states of that statement within the same time step. Attention vectors are calculated and concatenated using these embeddings as input. Finally, all attention vectors are fed into an RNN sequentially and all time outputs are pooled. LiGer's embedding quality is evaluated with COSET [31] benchmark. Wang et al. also extend their model into an encoder-decoder architecture and evaluate their model for the purpose of method name prediction. LiGer outperforms three relevant code-embedding approaches across a set of benchmarks. However, their execution trace processing technique implies significant complexity. They only evaluate it on small functions with simple contexts. It is unknown whether this technique can be scaled across multiple functions of a real codebase. On the other hand, we show that our approach scales effectively over real and complex programmes from different domains.

3 | APPROACH

Our approach for building an automated test oracle for classifying execution traces has the following steps:

Step 1: Instrument the PUT to gather traces when executing the test inputs.
Step 2: Preprocess the traces to prune unnecessary information.
Step 3: Encode the pre-processed traces into vectors that can be accepted by the NN.
Step 4: Design a NN model that takes as input an encoded trace and outputs a verdict of pass or fail for that trace.

Figure 2a illustrates the steps in our approach, with the bottom half of the figure depicting steps 3 and 4 for any given pre-processed trace from step 2. We discuss each of the steps in the rest of this section.

3.1 | Instrument and gather traces

For every test input executed through the PUT, we aim to collect an execution trace as a sequence of method invocations,

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

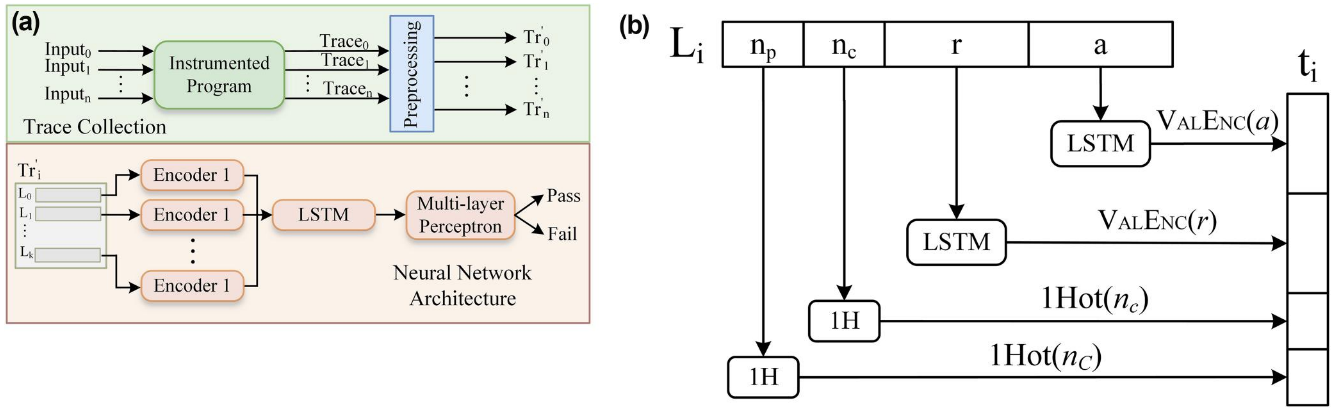

FIGURE 2 High-level architecture of our approach and Encoder 1 description. (a) Gathering traces, encoding them, and using neural networks to classify them; (b) Encoder 1 representing a single line in a trace as a vector containing function caller, callee names, arguments and return values

where we capture the name of the method being called, values and data types of parameters, return values and their types, and, finally, the name of the parent method in the call graph. We find gathering further information, for example updates to local variables within each method, incurs a significant overhead and is difficult to scale to large programmes. To gather this information we develop two different tools with support for different programing languages to make our framework widely applicable. Our first instrumentation tool is primarily aimed at C/C++ programmes and uses the middleware of LLVM [12] and instruments the intermediate representation (LLVM-IR) of programmes. This allows our implementation to be language-agnostic. LLVM provides front-end support for multiple programing languages in addition to C/C++ like CUDA, Haskell, Swift, and Rust among others, along with numerous libraries for optimisation and code generation.

Our second tool is aimed at Java programmes and uses Soot [13] to collect execution traces. Soot is a Java optimisation framework and provides libraries for users to analyse, instrument and optimise applications. We develop a pass with Soot to compile Java into bytecode and a second pass to convert bytecode to Jimple-IR. Jimple is a typed 3-address intermediate representation suitable for code transformations. Using Soot's abstract programming interface (API), we develop a second pass to instrument Jimple and finally compile into an executable. Our Soot framework is also compatible with any other programing language that can be compiled into Java bytecode, for example Scala.

To perform the instrumentation, we traverse the PUT, visiting each method. Every time a method invocation is identified, a code is injected to trace the caller–callee pair, the arguments and the return values. At the end of the programme, a code is inserted to write the trace information to the output.

Each trace contains a sequence of method invocations. This sequence comprises multiple lines, each line being a tuple (n_p, n_c, r, a) that represents a single method invocation within it having the following:

- The names of the caller (parent) n_p and called n_c functions.
- Return values rof the call, if any.
- Arguments passed a, if any.

The order of trace lines or method invocations is the order in which the methods complete and return to the calling point.

Our LLVM instrumentation supports all variable types including primitive types (such as int, float, char, and bool), composite data types (such as structs, classes, arrays) defined by a user or library, and pointers for return and argument values. Structs and classes are associated with a sequence of values for their internal fields. We instrument these data structures in a depth-first fashion, until all primitive types are traced. For pointers, we monitor the values they refer to.

Our instrumentation within Soot collects all Java primitive types, strings, primitive wrapper classes, atomic wrapper classes and arrays. We also support custom classes that are defined within the scope of a subject Java translation unit. Soot allows the instrumentation of public class members only; private methods and variables are not accessed.

3.2 | Training set

We execute the instrumented programme with each test input in the test suite to gather a set of traces. A subset of the traces is labelled and used in training the classification model. To label the traces as pass or fail, we compare actual outputs through the PUT with expected outputs provided by a reference programme or the specifications. Section 4.1 discusses how we label traces for the subject programmes in our experiment. It is worth noting that in our approach, the developer will only need to provide expected outputs for a *small proportion of tests rather than the whole test suite*. In the absence of expected output in tests, how tests will be labelled is a common question. Answering this question will depend on what is currently being done by the developer or organisation for classifying tests as pass or fail. Our approach will entail applying the same practice to labelling, albeit to a significantly smaller proportion of tests. To avoid data leakage in our

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

experiment in Section 4, we ensure that the expected output is removed from the traces. We also remove exceptions, assertions and any other information in the programme or test code that may act as a test oracle. This is further discussed in Section 4.2.

3.3 | Preprocessing

The execution traces gathered with our approach include information on methods declared in external libraries, called during the linking phase. To keep the length of the traces tractable and relevant, we preprocess the traces to only keep trace lines for methods that are defined within the module and remove trace lines for declared functions that are not defined but simply linked to it later.

For method invocations within loops, a new trace line is created for each invocation of the same method within the loop. For loops with large numbers of iterations, this can lead to redundancy when the method is invoked with similar arguments and return values. We address this potential redundancy issue by applying average pooling to trace lines with identical caller–callee methods within loops.

3.4 | Neural network model

In this step, we perform the crucial task of designing a NN that learns to classify the pre-processed traces as passing or failing. The shape and size of the input traces vary widely, and this presents a challenge when designing a NN that accepts fixed length vectors summarising the traces. To address this, our network comprises three components that are trained jointly and end-to-end: 1. a ValEnc that encodes values (such as the values of arguments and return values) into D_V-dimensional distributed vector representations, shown within Encoder 1 in Figure 2b, 2. a TrEnc that encodes variable-sized traces into a single D_T-dimensional vector, shown as Long Short Term Memory (LSTM) in Figure 2a, and finally, 3. a TraceClassifier that accepts the trace representation for state and predicts whether the trace is passing or failing. The multi-layer perceptron (MLP) in Figure 2a represents the TraceClassifier. We describe each component in detail in the rest of this section.

3.4.1 | Encoding values

Values within the trace provide useful indications about classifying a trace. However, values—such as ints, structs, and floats—vary widely in shape and size. We, therefore, design models that can summarise variable-sized sequences into fixed length representations. In the machine learning literature, we predominantly find three kinds of models that can achieve this: RNNs, 1D convolutional NNs and transformers. In this work, we employ LSTMs [32]—a commonly used flavour of RNNs. Testing other models is left as future work. At a high-level, RNNs are recurrent functions that accept a vector $\mathbf{h}_t$ of the current state and an input vector $\mathbf{x}_t$ and compute a new state vector $\mathbf{h}_{t+1} = RNN(\mathbf{x}_t, \mathbf{h}_t)$, which 'summarises' the sequence of inputs up to time t. A special initial state $\mathbf{h}_0$ is used at $t = 0$.

To encode a value v, we decompose it into a sequence of primitives $v = [p_0, p_1, \ldots]$ (integers, floats, and characters etc.). Each primitive p_i is then represented as a binary vector $\mathbf{b}_i = e(p_i)$ containing its bit representation padded to the largest primitive data type of the task. For example, if int64 is the largest primitive, then all $\mathbf{b}_i$s have dimensionality of 64. This allows us to represent all values (integers, floats, strings, structs, pointers etc.) as a unified sequence of binary vectors. We encode v into a D_V-dimensional vector by computing

$$\mathrm{ValEnc}(v) = LSTM_v(e(p_L)_L, \mathrm{ValEnc}([p_0, p_1, \ldots, p_{L-1}])),$$

where $LSTM_v$ is the LSTM that sequentially encodes the $\mathbf{b}_i$s. Note that we use the same ValEnc for encoding arguments and return values, as seen in Figure 2b. The intuition behind this approach is that the bits of each primitive can contain valuable information. For example, the bits corresponding to the exponent range of a float can provide information about the order of magnitude of the represented number, which in turn may be able to discriminate between passing and failing traces.

3.4.2 | Representing a single trace line

Armed with a NN component that encodes values, we can now represent a single line (n_p, n_c, r, a) of the trace. To do this, we use ValEnc to encode the arguments and the return value r. We concatenate these representations along with one-hot representations of the caller and callee identities, as shown in Figure 2b. Specifically, the vector encoding $\mathbf{t_i}$ of the ith trace line is the concatenation

$$\mathbf{t_i} = \left[\mathrm{ValEnc}(a), \mathrm{ValEnc}(r), \mathrm{1Hot}(n_p), \mathrm{1Hot}(n_c)\right],$$

where 1Hot is a function that takes as input the names of the parent or called methods and returns a one-hot vector that uniquely encodes that method name. For methods that are rare (appear fewer than k_{min} times) in our data, 1Hot collapses them to a single special Unknown name. This is similar to other machine learning and natural language processing models and reduces sparsity, often improving generalisation. The resulting vector $\mathbf{t_i}$ has size $2D_V + 2k$ where k is the size of each one-hot vector.

3.4.3 | Encoding traces

Now that we have built a NN component that encodes single lines within a trace, we design TrEnc that accepts a sequence of trace line representations $\mathbf{t}_0 \ldots \mathbf{t}_N$ and summarises them into a single D_T-dimensional vector. We use an LSTM with a hidden size D_T, and thus

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

$$\mathrm{TrEnc}(\mathbf{t}_0\ldots\mathbf{t}_N) = LSTM_{tr}(\mathbf{t_N}, \mathrm{TrEnc}(\mathbf{t}_0\ldots\mathbf{t}_{N-1})),$$

where $LSTM_{tr}()$ is an LSTM network that summarises the trace line representations.

3.4.4 | Classifying traces

With the NN components described so far we have managed to encode traces into fixed length vector representations. The final step is to use those computed representations to make a classification decision. We treat failing traces as the positive class and passing traces as the negative class since detecting failing runs is of more interest in testing. We compute the probability that a trace is failing as

$$P(\mathsf{fail}) = \mathrm{TraceClassifier}([\mathrm{TrEnc}(\mathbf{t}_0\ldots\mathbf{t}_N)]),$$

where the input of TRACECLASSIFIER is the output vector of TRENC. Our implementation of TRACECLASSIFIER is a MLP with sigmoid non-linearities and a single output, which can be viewed as the probability that the trace is a failing trace. It follows that $P(\mathsf{pass}) = 1 - P(\mathsf{fail})$.

3.4.5 | Training and implementation details

We train our network end-to-end in a supervised fashion, minimising the binary cross entropy loss. All network parameters (parameters of $LSTM_v$ and $LSTM_{tr}$ and parameters of the MLP) are initialised with random noise. For all the runs on our network we use $D_V = 128$, $D_T = 256$. The TRACECLASSIFIER is an MLP with three hidden layers of size 256, 128 and 64. We use the Adam optimiser [33] with a learning rate of $10e - 5$.

For our subject programmes, we find the aforementioned feature values to be optimal for performance and training time, after having experimented with other NN architectures, varying the D_V, D_T sizes, and the hidden layers in the MLP. We explored increasing D_V to 256, 512, D_T to 512, 1024 and the size of hidden layers to 512 and 1024.

To handle class imbalance in datasets, we explicitly counteract the imbalance in the loss function by down-weighting the samples within the most popular class such that samples of both class participate equally within this function.

Our implementation of the proposed approach is available at https://github.com/fivosts/Learning-over-test-executions.

4 | EXPERIMENT

In our experiment, we evaluate the feasibility and accuracy of the NN architecture proposed in Section 3 to classify execution traces for 15 subject programmes and their associated test suites. We investigate the following questions regarding feasibility and effectiveness:

Q1. Precision, Recall and Specificity: *What is the precision, recall and specificity achieved over the subject programmes?*

To answer this question, we use our tool to instrument the source code to record execution traces as sequences of method invocations, arguments and return values. A small fraction of the execution traces are labelled (*training set*) and fed to our framework to infer a classification model. We then evaluate the precision, recall and specificity achieved by the model over unseen execution traces (*test set*) for that programme. The test set includes both passing and failing test executions. We use *Monte Carlo cross-validation*, creating random splits of the dataset into training and test data. We created 15 such random splits and averaged the precision, recall and specificity computed over them.

Q2. Size of training set: *How does size of the training set affect precision and recall of the classification model?*

For each programme, we vary the size of the training set from 5% to 30% of the overall execution traces and observe its effect on the precision and recall achieved.

Q3. Comparison against state of art: *How does the precision, recall and specificity achieved by our technique compare against agglomerative hierarchical clustering, proposed by Almaghairbe* et al. *[16] in 2017?*

We choose to compare against the hierarchical clustering work as it is the most relevant and recent in classifying execution traces. Traces used in their work are sequences of method invocations, similar to our approach. Other test oracle work that use NNs is not used in the comparison as they do not work over execution traces and are limited in their applicability to programmes with numerical input and output, which is not the case for the programmes in our experiment.

Q4. Generalisation of the classification model: *Can a classification model inferred from execution traces of one programme be used to classify test executions over other programmes in the same domain?*

For the network protocol domain, we evaluate the accuracy of using a classification model inferred using traces from a single protocol detection finite state machine (FSM) to classify test executions from other protocol FSMs. In our experiments, we do not use a validation set to tune the hyper-parameters in the NN model.

All experiments are performed on a single machine with four Intel i5-6500 CPU cores, Nvidia RTX 2060 GPU, 16 GB of memory.

4.1 | Labelling traces

All our subject programmes are open source, and most of them were only accompanied by passing tests. This is not uncommon as most released versions of programmes are correct for the given tests. We take these correct programmes to be reference implementations. To enable evaluation of our approach that distinguishes correct versus incorrect executions, we need subject programmes with bugs. We, therefore,

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

generate PUTs by automatically mutating the reference implementation using common mutation operators [34] listed below:

1. Arithmetic operator replacement applied to {+, −, *, /, −−, ++}.
2. Logical connector replacement applied to {&&, ||, !}.
3. Bitwise operator replacement applied to {&, |, ∧, , ≪, ≫}.
4. Assignment operator replacement applied to {+=, − = , *=, /=, %=, ≪=, ≫=, & =, |=, ∧ =}.

A PUT is generated by seeding a single fault into the reference implementation at a random location using one of the above mutation operators. We used an independent open-source mutation tool[1] to generate PUTs from a given reference programme. Figure 3 shows a PUT generated by seeding a single fault into a reference programme. As seen in Figure 3, we run each test, T_i, in the test suite, through both the reference programme and PUT and label the trace as *passing* if the expected output, EO_i, from the reference matches the actual output, AO_i, from the PUT. If they do not match, the trace is labelled as *failing*. We rejected PUTs from mutations that did not result in any failing traces (outputs always match with the reference). This avoids the problem of equivalent mutants. All the PUTs in our experiment had both passing and failing traces.

4.2 | Subject programmes

We chose subject programmes from different domains to assess the applicability of our approach, namely from the blockchain, deep learning, encryption and text editing domains. A description of the programmes and associated tests is as follows:

1. Ethereum [35] is an open-source platform based on blockchain technology, which supports smart contracts. Within it, we evaluate our approach over the Difficulty module that calculates the mining difficulty of a block in relation to different versions (eras) of the cryptocurrency (Byzantium, Homestead, Constantinople etc.). The calculation is based on five fields of an Ethereum block, specified in the test input.

Tests: We use the default test inputs provided by Ethereum's master test suite for the Difficulty module. We test this module for the Byzantium era of the cryptocurrency (version 3.0). The test suite contains 2254 *automatically* generated test inputs. Each test input contains one hex field for the test input of the difficulty formula and another hex field for the expected output of the programme. All the test inputs provided with the module are passing tests with the actual output equal to the expected output. As a result, we use the provided module as a reference implementation. As described in Section 4.1, we seed faults into the reference implementation to generate PUTs, each

[1] https://github.com/chao-peng/mutec.

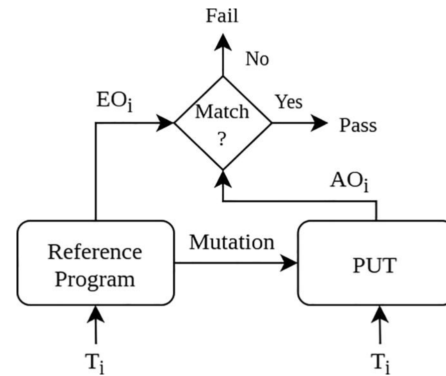

FIGURE 3 Labelling test executions by matching actual and expected behaviour. AO, actual output; EO, expected output; PUT, programme under test

containing a single mutation. For the difficulty module, we generate two PUTs: 1. Ethereum-Seal Engine (SE) with a seeded fault in the core functionality of the difficulty module and 2. Ethereum-common difficulty (CD) with a fault seeded in one of the functions that is external to the core function but appears in the call graph of the module. The balance between passing and failing tests varies between the two PUTs, Ethereum-CD being perfectly balanced and Ethereum-SE being slightly imbalanced (828 failing and 1426 passing tests).

2. Pytorch [36] is an optimised tensor library for deep learning, widely used in research. In our experiment, we evaluate our model over the intrusive_ptr class, which implements a pointer type with an embedded reference count. We chose this class because it had a sizeable number of tests (other modules had <20 published tests).

Tests: Implementation of the class is accompanied by 638 tests, all of which are passing. We, thus, use this as the reference implementation. As with Ethereum, we apply mutations to the intrusive_ptr implementation to generate a single PUT. Upon comparison with the reference, 318 of the existing tests are labelled as passing through the PUT and 320 as failing.

3. Microsoft SEAL [37] is an open-source encryption library. In our experiment, we study one component within Microsoft SEAL—the Encryptor module, which is accompanied by tests. This component is responsible for performing data encryption.

Tests: The Encryptor component is accompanied by 133 tests. The provided tests were all passing tests, with matching expected and actual output. As with previous programmes, we generate a PUT by mutating the original implementation. On the PUT, 11 tests fail and 122 pass.

4. Sed [38] is a Linux stream editor that performs text transformations on an input stream.

Tests: We use the fifth version of Sed available in the SIR repository [23]. This version is accompanied by 370 tests, of which 352 are passing and 18 are failing. The failing tests point to real faults in this version. Since the implementation was accompanied by both passing and failing, we used it as the PUT. We did *not* seed faults to generate the PUT.

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

5. L7-Filter [14] is a packet identifier for Linux. It uses regular expression matching on the application layer data to determine what protocols are used. It works with unpredictable, non-standard and shared ports. We study the following nine protocols, implemented as FSMs, separately in our evaluation:

1. Ares–P2P file sharing
2. BGP–Border Gateway Protocol
3. Biff–New mail notification
4. Finger–User information server
5. FTP–File Transfer Protocol
6. Rlogin–Remote login
7. TeamSpeak–VoIP application
8. Telnet–Insecure remote login
9. Whois–Query/response system (e.g. for domain name)

Tests: For each of the network protocol FSMs, we use test suites generated by Yaneva et al. [39] that provide all-transition pair coverage. The test suites for the FSMs have both passing and failing tests.

6. Commons-lang [40] is a java library from Apache Commons with utility classes for the java.lang API. This is a large codebase and contains Java classes such as Object and Class. We gather this subject programme from the Defects4J database that provides several versions of this library and a labelled set of passing and failing test cases for each version.

Tests: Defects4J contains different versions of commons-lang. Most of them have very few or even no failing tests. These versions do not provide our model with failing examples to learn and predict; therefore, we discard them. We use the 34th version of this programme, which contains 559 passing and 27 failing tests. These 27 tests are caused by real bugs found in this version of the subject programme; therefore, we do not seed faults to generate the PUT.

4.2.1 | Checks to avoid data leakage

We ensure that no test oracle data is leaked into traces. We remove expected outputs, assertions, exceptions, test names and any other information that may act directly or indirectly as a test oracle. For example, Ethereum uses the BOOST testing framework to deploy its unit tests. We remove expected outputs and assertions in the test code that compare the actual output with the expected output for example BOOST_CHECK_EQUAL.

For PUTs generated by seeding faults into the reference implementation, we only use one PUT for each reference implementation except in the case of Ethereum where we generated two PUTs, since faults were seeded in different files. Generating more PUTs for each reference implementation would be easy to do. However, we found that our results across PUTs for a given reference programme only varied slightly. As a result, we only report results over one to two PUTs for each reference implementation.

4.3 | Performance measurement

For each PUT, we evaluate performance of the classification model over unseen execution traces. As mentioned in Section 3.4, we use positive labels for failing traces and negative labels for passing. We measure

1. *Precision* as the ratio of number of traces correctly classified as 'fail' (TP) to the total number of traces labelled as 'fail' by the model (TP + FP).
2. *Recall* as the ratio of failing traces that were correctly identified (TP/(TP + FN)).
3. *Specificity* or true negative rate (TNR) as the ratio of passing traces that were correctly identified (TN/(TN + FP)).

TP, FP, TN, and FN represent true positive, false positive, true negative and false negative, respectively.

4.4 | Hierarchical clustering

In research question 3 in our experiment, we compare the classification accuracy of our approach against agglomerative hierarchical clustering proposed by Almaghairbe et al. [16]. Their technique also considers execution traces as sequences of method calls but only encoding callee names. Caller names, return values and arguments are discarded. We attempted to add the discarded information but found that the technique was unable to scale to large number of traces due to both memory limitations and a time complexity of $\mathcal{O}(n^3)$, where n is the number of traces. For setting clustering parameters for each subject programme, we evaluate different types of linkage (single, average, and complete) and a range of different cluster counts (as a percentage of the total number of tests): 1%, 5%, 10%, 20% and 25%. We use Euclidean distance as the distance measure for clustering. For each programme, we report the best clustering results achieved over all parameter settings.

5 | RESULTS AND ANALYSIS

In this section, we present and discuss our results in the context of the research questions presented in Section 4.

5.1 | Q1. Precision, recall and specificity

Table 1 shows the precision, recall and specificity achieved by the classification models in our approach for the different PUTs. Results achieved with the hierarchical clustering approach by Almaghairbe et al. [16] are also presented in Table 1 for comparison, but this is discussed in Q3 in Section 5.3. The column showing % of traces used in training varies across programmes; we show the lowest percentage that is needed to achieve near maximal precision and recall.

TABLE 1 Precision, recall and true negative rate (TNR) using our approach and hierarchical clustering

				Our approach			Hierarchical clustering [16]		
PUT	**Lines of code**	**% Traces for training**	**Total # traces**	**Precision**	**Recall**	**TNR**	**Precision**	**Recall**	**TNR**
Ethereum-CD	55,927	15	2254	0.80	0.82	0.79	1.0	0.49	1.0
Ethereum-SE	55,927	15	2254	0.99	0.82	0.86	1.0	0.25	1.0
Pytorch	21,090	10	638	0.99	0.98	0.99	0.48	1.0	0.16
SEAL encryptor	25,967	30	132	0.75	0.86	0.98	0.16	0.36	0.83
Sed	4492	10	370	0.94	0.94	0.99	0.35	0.63	0.86
Commons-lang	49,028	40	586	0.71	0.94	0.98	0.07	0.96	0.4
Ares protocol	1261	3	16,066	0.97	0.98	0.97	0.94	0.24	0.0
BGP protocol	1025	5	16,009	0.99	0.99	0.99	0.18	0.01	0.98
Biff protocol	627	15	1958	0.97	0.99	0.99	0.43	0.22	0.72
Finger protocol	791	10	2775	0.99	0.99	0.99	0.53	0.13	0.92
FTP protocol	995	10	9677	0.99	0.99	0.98	0.07	0.001	0.98
Rlogin protocol	955	10	4121	0.97	0.96	0.99	1.0	0.04	1.0
Teamspeak protocol	3284	10	1945	0.95	0.99	0.96	1.0	0.11	1.0
Telnet protocol	1019	10	319	0.98	0.96	0.95	0.29	0.02	0.87
Whois protocol	784	9	4412	0.98	0.99	0.99	0.49	0.03	0.98

Abbreviations: BGP, Border Gateway Protocol; CD, common difficulty; FTP, File Transfer Protocol; PUT, programme under test; SE, Seal Engine.

The classification models for all 15 PUTs achieve more than 71% precision and 86% recall, with an average of 93% and 94%, respectively. Our technique works particularly well for Pytorch, Sed and all networking protocols, achieving >= 94%. This implies that the number of false positives in the classification is very low and a large majority of the failing traces is correctly identified.

The classification models for all PUTs also achieve high specificity (>= 79%, average 96%). This implies that the NN models are able to learn runtime patterns that distinguish not only failing executions but also passing executions with a high degree of accuracy. These results are unprecedented as we are not aware of any technique in the literature that can classify both passing and failing executions at this level of accuracy.

5.1.1 | Analysis

To understand the results in Table 1, for each of the PUTs, we inspected and compared passing and failing traces using a combination of longest common subsequence, syntactic diffs, and manual inspection. We also performed *ablation*—systematically removing information (one parameter at a time) from the traces, training new classification models with the modified traces and observing their effect on precision, recall and specificity (TNR). In our experiments, we systematically remove the following parameters from the original traces: function call names, arguments, and return values. Table 2 shows the results from our ablation study. We discuss the results for each of the programmes in the following paragraphs:

Over SEAL Encryptor, our approach achieves 75% precision, 86% recall and 98% specificity when trained with 30% of the traces. The encryptor requires a higher fraction of traces for training when compared to other PUTs, as the number of failing traces is very small (=11), unlike other programmes. Although we handle imbalance in datasets by weighting samples in the loss function, the NN still needs some representatives of the failing class during training. Using 10% of the traces in training will only provide one example of failing trace (10% of 11), which is not enough for the NN model to learn to distinguish failing versus passing behaviour. Training using 30% of the traces includes 3 failing traces, which allows the NN to achieve 75% precision. The high precision with only 3 failing traces is because all the failing traces for this programme have the same call sequence, which is sufficiently different from the passing traces. Not all passing traces have the same sequence. However, due to the availability of a larger set of passing traces (training with 30% is 40 passing traces), the NN is able to identify the different method call patterns in passing traces accurately (98% specificity). The ablation study in Table 2 shows that all the parameters contribute to model performance as removing them has a detrimental effect.

For PyTorch, we achieve 99% precision, 98% recall and 99% specificity when trained with 10% of the traces. The dataset for PyTorch PUT is balanced (318 passing and 320 failing). 10% of the traces during training provides sufficient examples from both passing and failing classes for the NN to learn to distinguish them. We find that the reason for the superior performance of our model over PyTorch is that all failing traces have significantly fewer trace lines than passing

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

TABLE 2 Precision (P), recall (R) and specificity (TNR) for each PUT omitting certain trace information

PUT	Omitted info.	P	R	TNR
Ethereum-CD	Function names	0.63	0.64	0.62
	Return values	0.68	0.87	0.60
	Arguments	0.54	0.78	0.35
Ethereum-SE	Function names	0.96	0.84	0.35
	Return values	0.99	0.97	0.93
	Arguments	0.96	0.84	0.33
Pytorch	Function names	0.99	1.0	1.0
	Return values	0.99	0.99	0.99
	Arguments	0.51	0.99	0.04
Seal encryptor	Function names	0.53	0.87	0.92
	Return values	0.46	0.99	0.90
	Arguments	0.28	0.88	0.76
Sed	Function names	0.19	0.72	0.24
	Return values	0.48	0.52	0.85
	Arguments	0.30	0.40	0.73
Commons-lang	Function names	0.58	1.00	0.97
	Return values	0.74	0.88	0.99
	Arguments	0.78	0.95	0.99
Finger	Function names	0.99	0.95	0.99
	Return values	0.98	0.97	0.99
	Arguments	0.52	0.19	0.88
Telnet	Function names	0.93	1.0	0.76
	Return values	0.82	1.0	0.25
	Arguments	0.76	1.0	0.00
Ares	Function names	0.96	0.98	0.95
	Return values	0.95	0.99	0.75
	Arguments	0.93	0.96	0.68
BGP	Function names	0.99	0.98	0.98
	Return values	0.98	0.99	0.98
	Arguments	0.97	0.97	0.97
Biff	Function names	0.58	0.84	0.41
	Return values	0.56	0.92	0.35
	Arguments	0.51	0.64	0.40
FTP	Function names	0.99	0.99	0.98
	Return values	0.97	0.97	0.98
	Arguments	0.88	0.93	0.84
Rlogin	Function names	0.95	0.96	0.96
	Return values	1.0	0.92	1.0
	Arguments	0.85	0.91	0.94
Teamspeak	Function names	0.91	0.97	0.91
	Return values	0.94	0.98	0.94
	Arguments	0.77	0.86	0.77
Whois	Function names	0.96	0.96	0.96
	Return values	0.96	0.96	0.96
	Arguments	0.72	0.75	0.73

Abbreviations: BGP, Border Gateway Protocol; CD, Common Difficulty; FTP, File Transfer Protocol; PUT, programme under test; SE, Seal Engine.

traces. The consistent difference in the length of traces between the two classes allows the NN to easily distinguish them. The ablation study in Table 2 shows that arguments in traces matter for model performance, while method names and return values are irrelevant.

With Sed, our model achieves 94% precision and recall and 99% specificity using 10% of the traces in training. The dataset for Sed is unbalanced, with only 18 failing and 352 passing traces. 10% of the traces in training uses 2 failing tests and 35 passing tests. Given the extremely small sample of failing tests, it is surprising that the model classifies and identifies failing traces with such high precision and recall. To understand this, we examined both the passing and the failing trace lines. We find that the length of passing and failing traces is similar. All failing traces, however, have a call to a function, getChar, towards the end of the trace. This function call is absent in passing traces. We believe that associating this function call to failing traces may have helped the performance of the NN. The ablation study in Table 2 shows that all the parameters considered in our traces are important for the model performance.

For Ethereum-CD, our model achieves 80% precision, 82% recall and 79% specificity when trained with 15% of the traces—169 passing and 169 failing. Ethereum-CD was generated from the reference implementation using an arithmetic operator mutation in a function deeply embedded in the call graph for the difficulty module. Differences between failing and passing traces are not apparent, and analysing the longest common subsequence, syntactic difference and manual inspections did not reveal any characteristic feature for the failing or passing traces. We believe that the model performance of around 80% precision, recall and specificity is due to the similarity between the passing and failing traces and the esoteric nature of the mutation. The ablation study for this programme reveals that all the features in the traces slightly impact the model performance.

For commons-lang, our model achieves 71% precision, 94% recall and 98% specificity using 40% of the traces. This subject programme only contains 27 failing executions versus 559 passing executions. There is a stark imbalance between passing and failing traces for this programme, which impacts the precision achieved. We also observe that the failing

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

execution traces consist of multiple calls to a string conversion function, toString, towards the final parts of the sequence. We find this can serve as a distinguishing feature between passing and failing executions. It is worth noting that our classifier's performance significantly drops when removing function names in the ablation study and it may be because the toString function is no longer visible. In contrast, removing arguments or return values does not affect the performance visibly.

For Ethereum-SE, our model achieves 99% precision, 82% recall and 86% specificity with 15% traces in training—214 failing and 124 passing. Unlike Ethereum-CD, mutation to generate Ethereum-SE was in the core functionality. Failing traces when compared to passing traces had differences towards the end of the trace, which is easily distinguished by the NN. Curiously, removing return values in the ablation study increases recall and specificity. This may be because the model was previously overfitting to return values in traces, which may not have been relevant to the classification.

For L7-Filter networking protocols, all programmes have enough test inputs to help our model learn programme features with a small percentage of execution traces. Especially for Ares protocol with 16,066 test inputs, our model can achieve 97% precision, 98% recall and 97% specificity, labelling only 3% of the total traces for training. For BGP protocol, we train on 5% of the total traces and achieve 99% precision, 99% recall and 99% specificity. In all networking protocols, failing traces correspond to executions that lead to non-accepted states of the protocol's FSM. We observe that the sequence of function invocations is similar in both passing and failing executions. However, the state information in return values and arguments is critical in order to determine correctness. The ablation study supports this argument, as removing function names in any networking protocol has no effect in the classifier's performance. On the other hand, in all the protocols except for BGP, removing arguments dramatically decreases the model's precision, recall and specificity. Removing return values leads to a slight performance reduction. In Biff and Telnet, return values seem to be as important as arguments for our model's accuracy.

5.1.2 | Summary

Overall, we find that NN models for all our PUTs perform well as a test oracle, achieving an average of 93% precision, 94% recall and 96% specificity. The NN models perform exceptionally well for programmes whose traces have characteristic distinguishing features between passing and failing executions, such as differences in trace lengths or presence of certain function call patterns. In the absence of such features, NNs can still do well if they have enough training samples, as in Ethereum-CD. We also find that our approach can cope effectively with unbalanced data sets—4 of the 15 programmes in our experiment have unbalanced passing and failing traces.

5.2 | Q2. Size of training set

Figure 4 shows the precision and recall achieved by our approach with different training set sizes. The fraction of traces needed in training to achieve near maximal performance was 3% to 40% across the PUTs. Excluding SEAL Encryptor and commons-lang, all the other programmes only needed to be trained over 15% of the traces to achieve near maximal performance. Both SEAL Encryptor and commons-lang had very few failing traces, requiring a larger fraction of traces to get sufficient representation of failing classes during training. As seen in the plots in Figure 4, increasing the % of traces used in training does not increase precision and recall for all PUTs. For instance, Pytorch and Sed observe a dramatic increase in precision and recall when going from 5% to 10% traces in training. The performance, however, stagnates after that point with increasing traces. With Ethereum-CD and Ethereum-SE, precision or recall becomes worse after 20% traces. This may be because the model is overfitting to the training traces.

It is also worth noting that the absolute size of our training set varies across subject programmes. We find that our approach works with training sets with as few as 3 failing traces to as many as 214. The range of passing tests in training was between 31 and 169.

5.3 | Q3. Comparison against state of art

Table 1 presents precision, recall, and specificity (TNR) achieved by the agglomerative hierarchical clustering proposed by Almaghairbe et al. [16] on each of the PUTs. Comparing the precision, recall and TNR of our approach versus hierarchical clustering, we find our approach clearly outperforms the clustering approach on all but the Ethereum-CD PUT. This is because the hierarchical clustering assumption does not hold for these programmes. According to this assumption, the passing traces tend to be grouped in a few big clusters and failing traces are grouped into many small clusters. However, for these programmes, passing traces tend to be grouped in many small clusters based on their call sequence pattern, making it hard to distinguish them from failing traces by simply comparing cluster sizes.

With Ethereum-CD, the hierarchical clustering approach achieves precision and specificity of 100% and a recall of 49%. This is achieved with complete-linkage clustering, Euclidean distance and a cluster count equal to 10% of the total traces. In contrast, our approach achieves a precision of 80%, recall of 82% and specificity of 79%. To enable better comparison, we plot the precision-recall curve of the NN model in Figure 5 for Ethereum-CD, using 15% of the traces in training.

This curve shows the precision and recall of our trained model with respect to different values of the classification threshold. It is clear from the plot that for the same value of recall (49%), hierarchical clustering performs marginally better than our approach—100% versus 99%. Hierarchical clustering

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

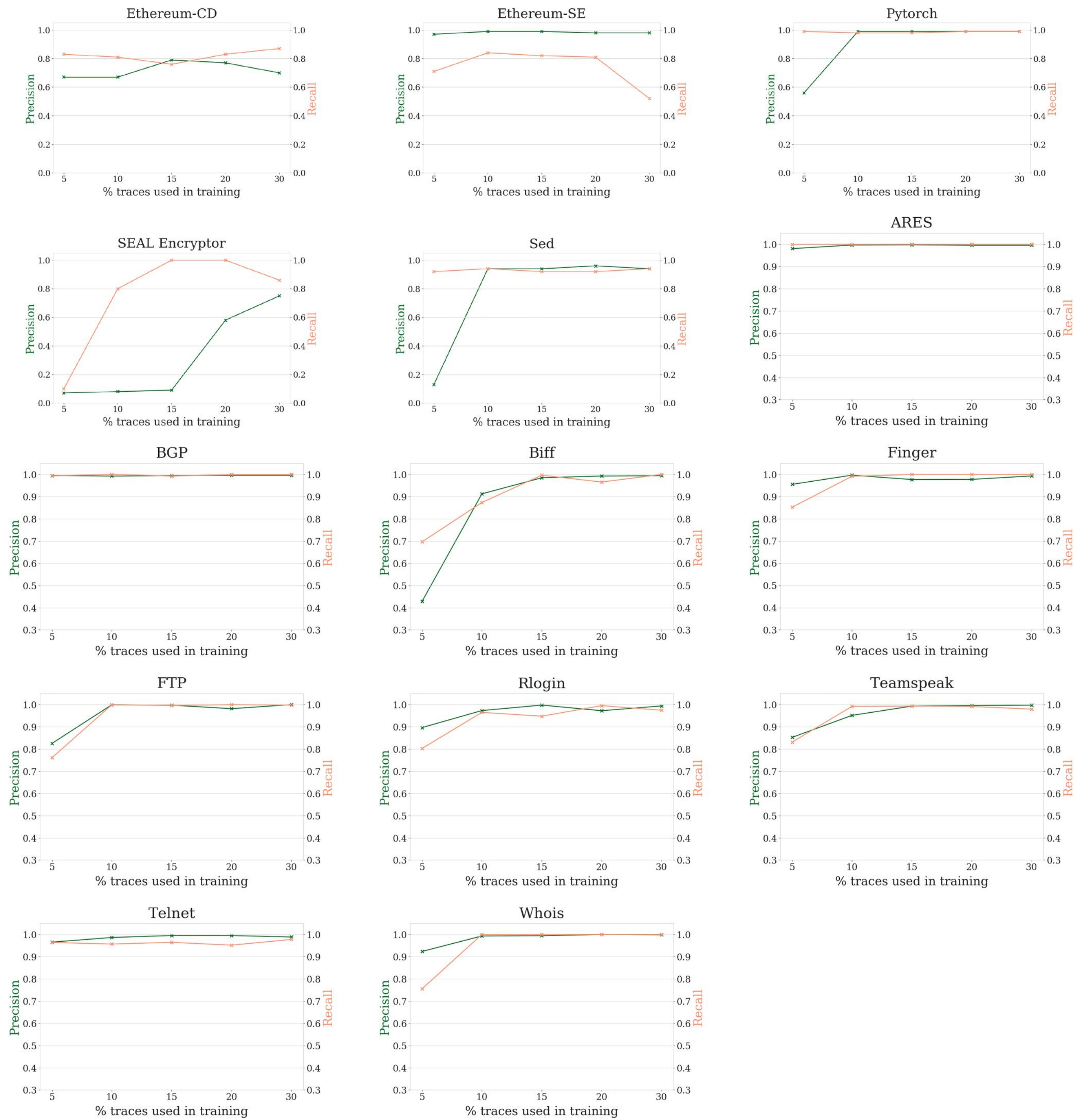

FIGURE 4 Precision and recall achieved by classification model over each programme under test. BGP, Border Gateway Protocol; FTP, File Transfer Protocol

works well over the Ethereum-CD PUT because the traces are clustered into just one big passing cluster and one failing cluster. The lack of cluster fragmentation improved the accuracy of the hierarchical clustering approach. Nevertheless, our model achieves comparable performance for such traces. In addition, our model allows tradeoff between precision and recall by changing the classification threshold, which may be driven by requirements or priorities of the use case. This tradeoff is not possible with the clustering approach.

5.4 | Q4. Generalisation

In this research question, we conduct an initial exploration into the ambitious possibility of using a model, trained using traces from one subject programme, to classify traces from other programmes in the same application domain. Figures 6 and 7 represent the precision and recall achieved by models trained using traces from the Biff protocol and Whois protocol, respectively, to classify traces produced by other FSMs.

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

We find that the model trained using traces from Biff achieves high accuracy over the Ares protocol with precision and recall close to 1.0 and reasonable precision (>0.8) for BGP, FTP, Rlogin, Teamspeak, and Whois protocols. The lowest precision (0.17) was observed with Telnet. The average precision achieved in classifying traces from unseen FSMs was 0.79. The recall achieved by the model is lower than the precision, indicating that the model missed identifying failing traces in each of these protocols. Overall, the model trained with Biff traces was successful in identifying failing traces in other FSMs that have similar patterns to Biff. Failing traces with differing patterns were missed. We confirmed this observation by checking the results from the Whois model. Although the precision and recall numbers are different from the Biff model, the reasoning for the classification success was the same—the extent of similarity in execution patterns between FSMs. With the current approach, we find there is scope to generalise a classification model from a single FSM to multiple FSMs in the networking domain. However, achieving high accuracies with generalisation is a difficult problem and we plan to take small, definitive steps towards addressing this challenge in the future. As a next step, we will explore tuning the classification model from an individual FSM with sample traces from other FSMs to improve the generalisation performance.

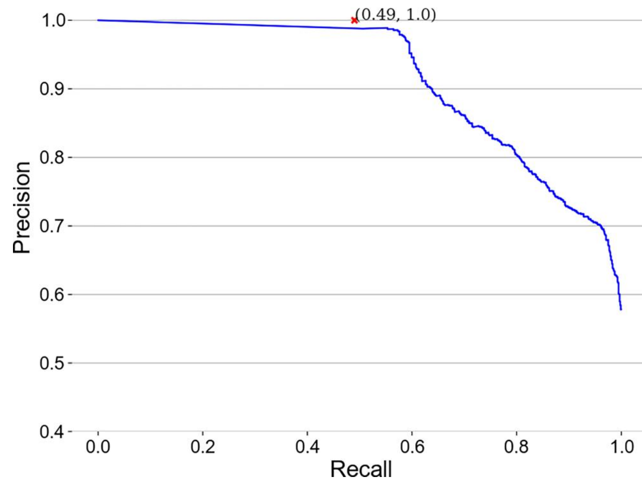

FIGURE 5 Precision–recall curve for Ethereum-CD

5.5 | Threats to validity

We see three threats to the validity of our experiment based on the selection of subject programmes and associated tests.

First, PUTs for 4 out of the 15 subject programmes in our experiment were generated by seeding faults into a reference implementation. A reference implementation with only passing tests is not suitable for evaluating our approach. To address this, we generated a faulty implementation and ran the original tests through the PUT to gather both passing and failing traces.

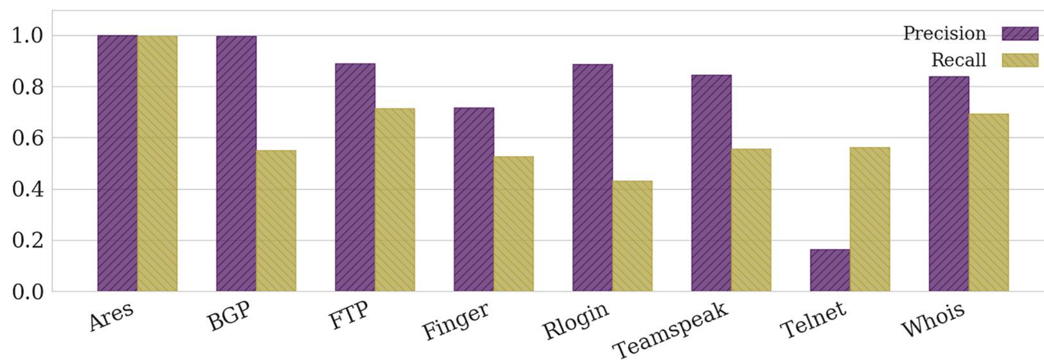

FIGURE 6 Biff trained model–Precision and recall for unseen finite state machines. BGP, Border Gateway Protocol; FTP, File Transfer Protocol

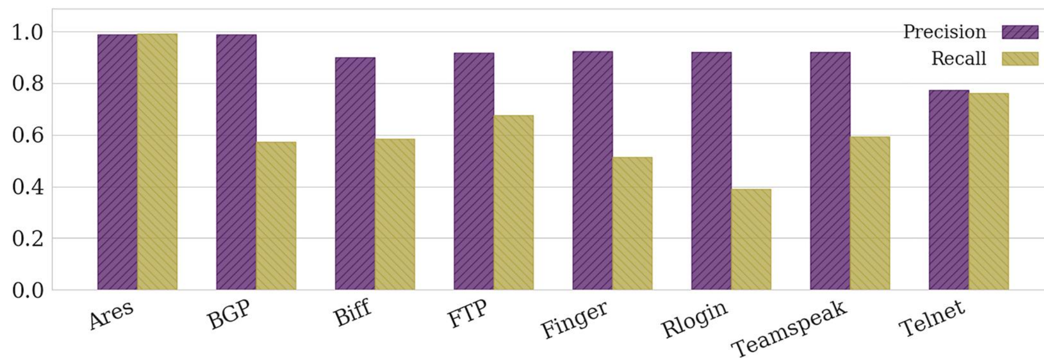

FIGURE 7 Whois trained model–Precision and recall for unseen finite state machines. BGP, Border Gateway Protocol; FTP, File Transfer Protocol

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

It is possible that using real faults in place of seeded faults may lead to different results. However, Andrews et al. have shown that the use of seeded faults leads to conclusions similar to those obtained using real faults [41, 42]. For one of the subject programmes, Sed, we did not artificially seed faults but instead used the existing implementation as it was accompanied by both passing and failing tests.

Second, the number of tests that accompanied our subject programmes was not very large, ranging from 132 to 16,066 tests. The NN models in our experiments produced good performance results with small- to medium-sized test suites that may be automatically or manually generated. Our approach is constrained by the amount of training data and not by the size of the test suite. As a result, for programmes accompanied by large test suites, the NN model will need a larger training set (the fraction of traces to be used in training might still be 14%). Nevertheless, the labelling effort for a fraction of the tests in our approach is still less than the current practice of labelling all the tests.

Finally, we conducted our study on subject programmes from five different application domains, which are not representative of all the application domains. Given that our approach has no domain specific constraints, we believe it will be widely applicable.

6 | CONCLUSION

In this study, we describe a novel approach for designing a test oracle as a NN model, learning from the execution traces of a given programme. We have implemented an end-to-end framework for automating the steps in our approach, (1) gathering execution traces as sequences of method invocations, (2) encoding variable length execution traces into a fixed length vector, and (3) designing a NN model that uses the trace information to classify the trace as pass or fail. We augmented our work in [7] by supporting Java programmes in addition to C/C++ in Step 1. In addition, we conducted an extensive evaluation using 15 realistic PUTs and tests. We found that the classification model for each PUT was effective in classifying passing and failing executions, achieving an average of 93% precision, 94% recall and 96% specificity while only training with an average 14% of the total traces. We outperform the hierarchical clustering technique proposed in recent literature by a large margin of accuracy for 14 out of the 15 PUTs and achieved comparable performance for the other PUTs. We carried out an initial experiment by generalising a classification model learnt over one protocol FSM to classify executions over other network protocol FSMs. The results for precision and recall over other unseen FSMs were not as high as the individual FSM classification models. In the future, we plan to explore techniques that will improve the generalisation performance of the NN models.

ACKNOWLEDGEMENTS

This work was supported by the EPSRC Centre for Doctoral Training in Pervasive Parallelism (EP/L01503X/1) at the University of Edinburgh, School of Informatics and the Facebook Testing and Verification Award 2018 and 2019.

CONFLICT OF INTEREST

No conflict of interest has been declared by the authors.

PERMISSION TO REPRODUCE MATERIALS FROM OTHER SOURCES

None.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available in github at https://github.com/fivosts/Learning-over-test-executions.

ORCID

Foivos Tsimpourlas https://orcid.org/0000-0001-8081-604X

REFERENCES

1. Chen, T.Y., et al.: An orchestrated survey on automated software test case generation. J. Syst. Software. 86(8), 1978–2001 (2013)
2. Bertolino, A.: Software testing research: Achievements, challenges, dreams. In: Future of Software Engineering, pp. 85–103. IEEE Computer Society (2007)
3. Barr, E., et al.: The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. 41(5), 507–525 (2015)
4. Nardi, P.A., Damasceno, E.: A survey on test oracles. Adv. Theor. Appl. Inf. 1(2), 50–59 (2015)
5. Langdon, W., et al.: Inferring automatic test oracles. In: Proceedings of the 10th Search-Based Software Testing, pp. 5–6. Buenos Aires (2017)
6. Liu, H., et al.: How effectively does metamorphic testing alleviate the oracle problem? IEEE Trans. Software Eng. 40(1), 4–22 (2014)
7. Tsimpourlas, F., Rajan, A., Allamanis, M.: Supervised learning over test executions as a test oracle. In: The 36th ACM/SIGAPP Symposium on Applied Computing-Software Engineering Track. ACM South Korea (2021)
8. Vanmali, M., Last, M., Kandel, A.: Using a neural network in the software testing process. Int. J. Intell. Syst. 17(1), 45–62 (2002)
9. Jin, H., et al.: Artificial neural network for automatic test oracles generation. In: Proceedings of International Conference on Computer Science and Software Engineering, vol. 2, pp. 727–730. IEEE Wuhan (2008)
10. Alon, U., et al.: Code2vec: Learning distributed representations of code. arXiv preprint arXiv:1803.09473 (2018)
11. Pradel, M., Sen, K.: Deepbugs: A learning approach to name-based bug detection. Proc. ACM Program Lang. 2(OOPSLA), 147 (2018)
12. Lattner, C.: LLVM: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana. http://llvm.cs.uiuc.edu (2002)
13. Vallée-Rai, R., et al.: Soot-a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research, CASCON'99, pp. 13. IBM Press (1999)
14. L7-filter: Application layer packet classifier for linux. ClearCenter. http://l7-filter.clearos.com/ (2013). Accessed 13 Aug 2021
15. Just, R., Jalali, D., Ernst, M.D.: Defects4j: A database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pp. 437–440. Association for Computing Machinery, New York (2014)
16. Almaghairbe, R., Roper, M.: Separating passing and failing test executions by clustering anomalies. Software Qual. J. 25(3), 803–840 (2017)

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License

17. Hierons, R.M.: Verdict functions in testing with a fault domain or test hypotheses. ACM Trans. Software Eng. Methodol. 18(4), 14 (2009)
18. Hierons, R.M.: Oracles for distributed testing. IEEE Trans. Softw. Eng. 38(3), 629–641 (2012)
19. Briand, L.C.: Novel applications of machine learning in software testing. In: 2008 International Conference on Quality Software (QSIC'08), pp. 3–10. IEEE Oxford (2008)
20. Bowring, J., Rehg, J.M., Harrold, M.J.: Active learning for automatic classification of software behavior. Software Eng. Notes. 29, 195–205 (2004)
21. Brun, Y., Ernst, M.D.: Finding latent code errors via machine learning over program executions. In: Proceedings of the 26th International Conference on Software Engineering, pp. 480–490. Edinburgh (2004)
22. Podgurski, A., et al.: Automated support for classifying software failure reports. In: Proceedings of 25th International Conference on Software Engineering 2003, pp. 465–475. IEEE Portland (2003)
23. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empir. Software Eng. 10(4), 405–435 (2005)
24. Aggarwal, K.K., et al.: A neural net based approach to test oracle. Software Eng. Notes. 29(3), 1–6 (2004)
25. Hinton, G., Osindero, S., Teh, Y.-W.: A fast learning algorithm for deep belief nets. Neural Comput. 18(7), 1527–1554 (2006)
26. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. In: International Conference on Learning Representations Arxiv (2018)
27. Alon, U., Levy, O., Yahav, E.: Code2seq: Generating sequences from structured representations of code. arXiv preprint arXiv:1808.01400 (2018)
28. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning (2016)
29. Wang, K., Singh, R., Su, Z.: Dynamic neural program embedding for program repair. In: International Conference on Learning Representations (2018)
30. Wang, K., Su, Z.: Learning blended, precise semantic program embeddings Arxiv, (2019)
31. Wang, K., Christodorescu, M.: Coset: A benchmark for evaluating neural program embeddings Arxiv, (2019)
32. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. 9(8), 1735–1780 (1997)
33. Diederik, P.K., Ba, J.: Adam: A method for stochastic optimization. In: 3rd International Conference for Learning Representations (2015)
34. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Software Eng. 37(5), 649–678 (2011)
35. Ethereum Project (release 3.5). Open Source. https://github.com/ethereum/aleth (2019). Accessed 13 Aug 2021
36. Paszke, A. & Chintala, S.: Pytorch https://pytorch.org/ (2017). Accessed 13 Aug 2021
37. Microsoft Research, Redmond. https://github.com/Microsoft/SEAL (2019). Accessed 13 Aug 2021
38. Sed, linux stream editor. https://linux.die.net/man/1/sed (2009). Accessed 13 Aug 2021
39. Yaneva, V., et al.: Accelerated finite state machine test execution using gpus. In: Asia-Pacific Software Engineering Conference (2018)
40. Commons lang. Apache Commons. https://commons.apache.org/proper/commons-lang/ (2020). Accessed 13 Aug 2021
41. Andrews, J.H, et al.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Software Eng. 32(8), 608–624 (2006)
42. Do, H., Rothermel, G.: On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Trans. Software Eng. 32(9), 733–752 (2006)

How to cite this article: Tsimpourlas, F., et al.: Embedding and classifying test execution traces using neural networks. IET Soft. 16(3), 301–316 (2022). https://doi.org/10.1049/sfw2.12038

17518814, 2022, 3, Downloaded from https://ietresearch.onlinelibrary.wiley.com/doi/10.1049/sfw2.12038 by Edinburgh University, Wiley Online Library on [20/02/2023]. See the Terms and Conditions (https://onlinelibrary.wiley.com/terms-and-conditions) on Wiley Online Library for rules of use; OA articles are governed by the applicable Creative Commons License