Deep Specification Mining

Tien-Duy B. Le
School of Information Systems
Singapore Management University, Singapore
btdle.2012@smu.edu.sg

David Lo
School of Information Systems
Singapore Management University, Singapore
davidlo@smu.edu.sg

ABSTRACT

Formal specifications are essential but usually unavailable in software systems. Furthermore, writing these specifications is costly and requires skills from developers. Recently, many automated techniques have been proposed to mine specifications in various formats including finite-state automaton (FSA). However, more works in specification mining are needed to further improve the accuracy of the inferred specifications.

In this work, we propose Deep Specification Miner (DSM), a new approach that performs deep learning for mining FSA-based specifications. Our proposed approach uses test case generation to generate a richer set of execution traces for training a Recurrent Neural Network Based Language Model (RNNLM). From these execution traces, we construct a Prefix Tree Acceptor (PTA) and use the learned RNNLM to extract many features. These features are subsequently utilized by clustering algorithms to merge similar automata states in the PTA for constructing a number of FSAs. Then, our approach performs a model selection heuristic to estimate F-measure of FSAs and returns the one with the highest estimated F-measure. We execute DSM to mine specifications of 11 target library classes. Our empirical analysis shows that DSM achieves an average F-measure of 71.97%, outperforming the best performing baseline by 28.22%. We also demonstrate the value of DSM in sandboxing Android apps.

CCS CONCEPTS

• **Software and its engineering → Dynamic analysis**;

KEYWORDS

Specification Mining, Deep Learning

1 INTRODUCTION

Due to rapid evolution to meet demands of clients, software applications and libraries are often released without documented specifications. Even when formal specifications are available, they may become outdated as software systems quickly evolve [51] in a short period of time. Finally, writing formal specifications requires necessary skill and motivation from developers, as this is a costly and time consuming process [23]. Furthermore, the lack of specifications negatively impacts the maintainability and reliability of systems. With no documented specifications, developers may find it difficult to comprehend a piece of code and software is more likely to have bugs due to mistaken assumptions. Furthermore, developers cannot utilize state-of-the-art bug finding and testing tools that need formal specifications as an input [10, 37].

Recently, many automated approaches have been proposed to help developers reduce the cost of manually drafting formal specifications [6, 13, 24, 26, 33]. In this work, we focus on the family of specification mining algorithms that infer finite-state automaton (FSA) based specifications from execution traces. Krka et al. [24] and many other researchers have proposed various FSA-mining approaches that have improved the quality of inferred FSA models as compared to prior solutions. Nevertheless, the quality of mined specifications is not perfect yet, and more works need to be done to make specification mining better. In fact, FSA based specification miners still suffer from many issues. For instance, if methods in input execution traces frequently occur in a particular order or the amount of input traces is too small, FSAs inferred by k-tails [7] and many other algorithms are likely to return FSAs that are not generalized and overfitted to the input execution traces.

To mine more accurate FSA models, we propose a new specification mining algorithm that performs deep learning on execution traces. We name our approach **DSM** which stands for Deep Specification Miner. Our approach takes as input a target library class C and employs an automated test case generation tool to generate thousands of test cases. The goal of this test case generation process is to capture a richer set of valid sequences of invoked methods of C. Next, we perform deep learning on execution traces of generated test cases to train a Recurrent Neural Network Language Model (RNNLM) [39]. After this step, we construct a Prefix Tree Acceptor (PTA) from the execution traces and leverage the learned language model to extract a number of interesting features from PTA's nodes. These features are then input to clustering algorithms for merging similar states (i.e., PTA's nodes). The output of an application of a clustering algorithm is a simpler and more generalized FSA that reflects the training execution traces. Finally, our approach predicts the accuracy of constructed FSAs (generated by different clustering algorithms considering different settings) and outputs the one with highest predicted value of F-measure.

We evaluate our proposed approach for 11 target library classes which were used before to evaluate many prior work [24, 26]. For each of the input class, we first run Randoop to generate thousands of test cases. Then, we use execution traces generated by running these test cases to infer FSAs. Our experiments show that DSM achieves an average F-measure of 71.97%. Compared to other existing specification mining algorithms, our approach outperforms all baselines that construct FSAs from execution traces (e.g., k-tails [7], SEKT [24], TEMI [24], etc.) by at least 28.22%. Some of the baselines first use Daikon to learn invariants that are then used to infer a better FSA. Our approach does not use Daikon invariants in the inference of FSAs. Excluding baselines that use Daikon invariants, our approach can outperform the remaining best performing miner by 33.24% in terms of average F-measure.

Additionally, we assess the applicability of FSAs mined by DSM in detecting malicious behaviors in Android apps. We propose a technique that leverages a FSA output by DSM mining algorithm as a behavior model to construct an Android sandbox. Our technique outputs a comprehensive sandbox that considers execution context of sensitive API methods to better protect app users. Our comparative evaluation finds that our technique can increase the True Positive Rate of Boxmate [21], a state-of-the-art sandbox mining approach, by 15.69%, while only increasing False Positive Rate by 4.52%. Replacing DSM with the best performing applicable baseline results in a sandbox that can achieve a similar True Positive Rate (as DSM) but substantially worse False Positive Rate (i.e., False Positive Rate increases by close to 10%). The results indicate it is promising to employ FSAs mined by DSM to create more effective Android sandboxes.

The contributions of our work are highlighted below:

(1) We propose DSM (Deep Specification Miner), a new specification mining algorithm that utilizes test case generation, deep learning, clustering, and model selection strategy to infer FSA based specifications. To the best of our knowledge, we are the first to use deep learning for mining specifications.
(2) We evaluate the effectiveness of DSM on 11 different target library classes. Our results show that our approach outperforms the best baseline by a substantial margin in terms of average F-measure.
(3) We propose a technique that employs a FSA inferred by DSM to construct a more comprehensive sandbox that considers execution context of sensitive API methods. Our evaluation shows that our proposed technique can outperform several baselines by a substantial margin in terms of either True Positive Rate or False Positive Rate.

The remainder of this paper is structured as follows. Section 2 highlights background materials. Sections 3 and 4 present DSM and its evaluation. Section 5 presents our proposed technique that employs a FSA inferred by DSM for detecting malicious behaviors in Android apps, along with its evaluation. We discuss threats to validity and related works in Section 6 and Section 7, respectively. Finally, we conclude and mention future work in Section 8.

2 BACKGROUND

Statistical Language Model: A statistical language model is an oracle that can foresee how likely a sentence $\texttt{s} = w_1, w_2, \ldots, w_n$ to occur in a language. In a nutshell, a statistical language model considers a sequence $\texttt{s}$ to be a list of words $w_1, w_2, ..., w_n$ and assigns probability to $\texttt{s}$ by computing joint probability of words: $P(w_1, \ldots, w_n) = \prod_{i=1}^{n-1} P(w_i|w_1, \ldots, w_{i-1})$. As it is challenging to compute conditional probability $P(w_i|w_1, \ldots, w_{i-1})$, each different language model has its own assumption to approximate the calculation. N-grams model, a popular family of language models, approximates in a way that a word w_k conditionally depends only on its previous N words (i.e., $w_{k-N+1}, \ldots, w_{k-1}$). For example, unigram model simply estimates $P(w_i|w_1, \ldots, w_{i-1})$ as $P(w_i)$, bigram model approximates $P(w_i|w_1, \ldots, w_{i-1})$ as $P(w_i|w_{i-1})$, etc. In this work, we utilize the ability of language models to compute $P(w_i|w_1, \ldots, w_{i-1})$ for estimating features of automaton states. We consider every method invocation as a word and an execution trace of an object as a sentence (i.e., sequence of method invocations). Given a sequence of previously invoked methods, we use a language model to output the probability of a method to be invoked next.

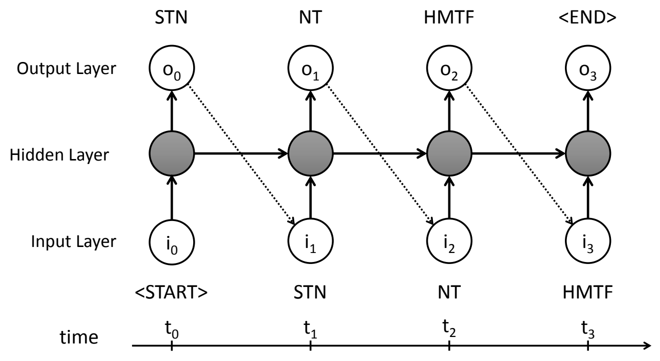


Figure 1: An unrolled Recurrent Neural Network from time t_0 to t_3 for predicting the next likely method given a sequence of invoked methods for `java.util.StringTokenizer`. "STN" : StringTokenizer(), "NT" : nextToken(), and "HMTF" : hasMoreTokens()==false.

Recurrent Neural Network Based Language Model: Recently, a family of language models that make use of neural networks is shown to be more effective than n-grams [38]. These models are referred to as neural network based language models (NNLM). If a NNLM has many hidden layers, we refer to the model as a *deep neural network language model* or *deep language model* for short. Among these deep language models, Recurrent Neural Network Based Language Model (RNNLM) [39] is well-known with its ability to use internal memories to handle sequences of words with arbitrary lengths. The underlying network architecture of a RNNLM is a Recurrent Neural Network (RNN) that stores information of input word sequences in its hidden layers. Figure 1 demonstrates how a RNN operates given the sequence `<START>`, `STN`, `NT`, `HMTF`, `<END>`. In the figure, a RNN is unrolled to become four connected networks, each of which is processing one input method at a time step. Initially, all states in the hidden layer are assigned to zeros. At time t_k, a method m_k is represented as an one-hot vector i_k by the input layer. Next, the hidden layer updates its states by using the vector i_k and the states previously computed at time t_{k-1}. Then, the output layer estimates a probability vector o_k across all methods for them to appear in the next time step t_{k+1}. This process is repeated at subsequent time steps until the last method in the sequence is handled.

3 PROPOSED APPROACH

Figure 2 shows the overall framework of our proposed approach. In our framework, there are three major processes: test case generation and traces collection, Recurrent Neural Network Based Language Model (RNNLM) learning, and automata construction. Our approach takes as input a target class and signatures of methods. Then, DSM runs Randoop [42] to generate a substantial number of test cases for the input target class. Then, we record the execution of these test cases, and retain traces of invocations of methods of the input target class as the training dataset. Next, our approach performs deep learning on the collected traces to infer a RNNLM that is capable of predicting the next likely method to be executed given a sequence of previously called methods. We choose RNNLM over traditional probabilistic language models since past studies show its superiority [39, 44].

Subsequently, we employ a heuristic to select a subset of traces that best represents the whole training dataset. From these traces, we construct a Prefix Tree Acceptor (PTA); we refer to each PTA's node as an automaton state. We select the subset of traces in order to optimize the performance when constructing PTA, but still maintaining accuracy of inferred FSAs. Utilizing the inferred RNNLM, we extract a number of features from automaton states, and input the feature values to a number of clustering algorithms (i.e., k-means [35] and hierarchical clustering [43]) considering different settings (e.g., different number of clusters). The output of a clustering algorithm are clusters of similar automaton states. We use these clusters to create a new FSA by merging states that belong to the same cluster. Every application of a clustering algorithm with a particular setting results in a different FSA. We propose a model selection strategy to heuristically select the most accurate model by predicting values of Precision, Recall, and F-measure. Finally, we output the FSA with highest predicted F-measure.

3.1 Test Case Generation and Trace Collection

This process plays an important role to our approach as it decides the quality of RNNLM inferred by the deep learning process. Previous research works in specification mining [24, 26, 27] collect traces from the execution of a program given unit test cases or inputs manually created by researchers. In this work, we utilize deep learning for mining specification. Deep learning requires a substantially large and rich amount of data. The more training inputs, the more patterns the resultant RNNLM can capture. In general, it is difficult to follow previous works to collect a rich enough set of execution traces for an arbitrary target library class. Firstly, it is challenging to look for all projects that use the target library class, especially for classes from new or unreleased libraries. Secondly, existing unit test cases or manually created inputs may not cover many of the possible execution scenarios of methods in a target class.

We address the above issues by following Dallmeier et al. [11, 12] to generate as many test cases as possible for mining specifications, and collect the execution traces of these test cases for subsequent steps. Recently, many test case generation tools have been proposed such as Randoop[1] [42], EvoSuite[2] [16], etc. Among the state-of-the-art test case generation tools, we choose Randoop because it is widely used and lightweight. Furthermore, Randoop is well maintained and frequently updated with new versions. As future work, we plan to integrate many other test case generation methods into our approach.

Randoop generates a large number of test cases, which is proportional to the time limit of its execution. In order to improve the coverage of possible sequences of methods under test, we provide class-specific literals aside from default ones to Randoop. For example, for `java.net.Socket`, we create string and integer literals which are addresses of hosts (e.g., "localhost", "127.0.0.1", etc.) and listening ports (e.g., 8888, etc.). Furthermore, we create driver classes that contain static methods that invoke constructors of the target class to initialize new objects. That helps speed up Randoop to create new objects without spending time to search for appropriate input values for constructors.

3.2 Learning RNNLM for Specification Mining

3.2.1 Construction of Training Method Sequences. Our set of collected execution traces is a series of method sequences. Each of these sequences starts and ends with two special symbols: <`START`> and <`END`>, respectively. These symbols are used for separating two different sequences. We gather all sequences together to create data for training Recurrent Neural Networks. Furthermore, we limit the maximum frequency of a method sequence `MAX_SEQ_FREQ` to 10 to prevent imbalanced data issue where a sequence appears much more frequently than the other ones.

3.2.2 Model Training. We perform deep learning on the training data to learn a Recurrent Neural Network Based Language Model (RNNLM) for every target library class. By default, we use Long Short-Term Memory (LSTM) network [20], one of the state-of-the-art RNNs, as the underlying architecture of the RNNLM. Compared to the standard RNN architecture, LSTM is better in learning long-term dependencies. Furthermore, LSTM is scalable for long sequences [44].

3.3 Automata Construction

In this processing step, our approach takes as input the set of training execution traces and the inferred RNNLM (see Section 3.2). The output of this step is a FSA that best

[1] https://randoop.github.io/randoop/
[2] http://www.evosuite.org/

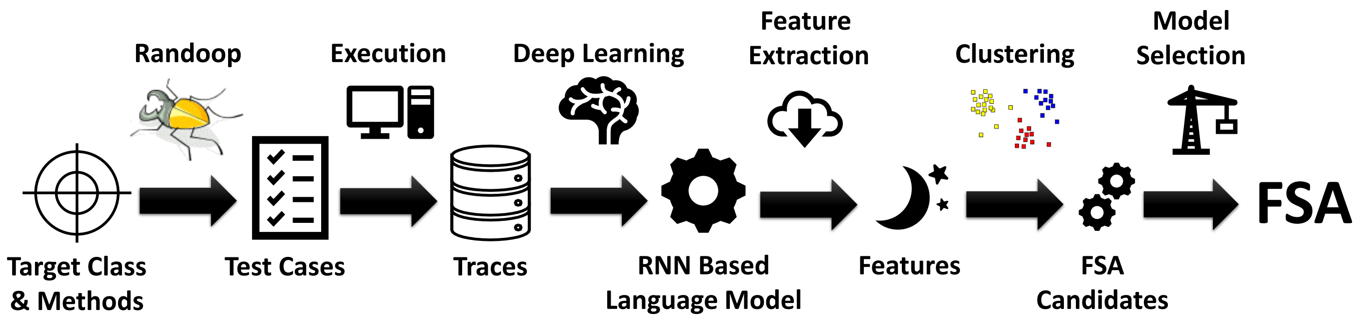


Figure 2: DSM's Overall Framework

captures the specification of the corresponding target class. The construction of FSA undergoes several substeps: trace sampling, feature extraction, clustering, and model selection.

At first, we use a heuristic to select a subset of method sequences that represents all training execution traces. The feature extraction and clustering steps use these selected traces, instead of all traces, to reduce computation cost. We construct a Prefix Tree Acceptor (PTA) from the selected traces and extract features for every PTA nodes using the inferred RNNLM. We refer to each PTA node as an automaton state. Figure 3 shows an excerpt of an example PTA constructed from sequences of invocations of methods from `java.security.Signature`. Our goal is to find similar automaton states and group them into one cluster. In the clustering substep, we run a number of clustering algorithms on PTA nodes with various settings to create many different FSAs. Finally, in the model selection substep, we follow a heuristic to predict the F-measure (see Section 4.2.1) of constructed FSAs and output the one with highest predicted F-measure. The full set of traces is used in this model selection step. In the following paragraphs, we describe details of each substep in this processing step:

Trace Sampling: Our training data contains a large number of sequences. Thus, it is expensive to use all of them for constructing FSAs. Therefore, the goal of trace sampling is to create a smaller subset that is likely to represent the whole set of all traces reasonably well. We propose a heuristic to find a subset of traces that covers all co-occurrence pairs[3] of methods in all training traces.

Algorithm 1 shows our heuristic to select execution traces from the whole set of execution traces S. At first, we determine all pairs (a, b) where a and b are methods that occur together in at least one trace in S and store them in P (lines 4 to 9). Next, we create a set O that contains selected traces – initially O is an empty set (line 10). Then, we iteratively choose a pair (a, b) which does not appear in any trace in O and occur in the least number of input traces in S (line 12). Given a selected pair (a, b), we look for the shortest trace $S_i \notin O$ where $a, b \in S_i$ (line 13). Once the trace is found, we include S_i to O. We mark the pair (a, b) as processed by removing it from P (line 15). We keep searching for sequences until O covers all co-occurrence pairs (a, b) in the input execution traces.

[3](m_1, m_2) is a co-occurrence pair if m_1 and m_2 appear together in at least one trace.

Algorithm 1: Selecting Subset of Execution Traces

```
Input : $S = \{S_i \mid 1 \le i \le N\}$: Collection of execution
        traces where $N$ is the number of traces
Output : $O$: Selected execution traces
1  Sort $S$ in ascending order of trace length
2  $D \leftarrow$ initialization of dictionary type
3  $P \leftarrow$ empty set
4  for $S_i \in S$ do
5      for $(a, b)$ where $a \in S_i \wedge b \in S_i \wedge a < b$ do
6          $D[(a, b)] += [S_i]$
7          Include $(a, b)$ to $P$
8      end
9  end
10 $O \leftarrow$ empty set
11 while $\exists (a, b) \in P$ do
12     Select $(a, b) \in P$ such that $D[(a, b)]$ has the least
         number of elements
13     Find $S_i \in D[(a, b)]$ such that $S_i$ is shortest $\wedge S_i \notin O$
14     Include $S_i$ to $O$ and remove all pairs $(a, b)$ from $P$
         where $a \in S_i \wedge b \in S_i \wedge a < b$
15 end
16 return $O$
```

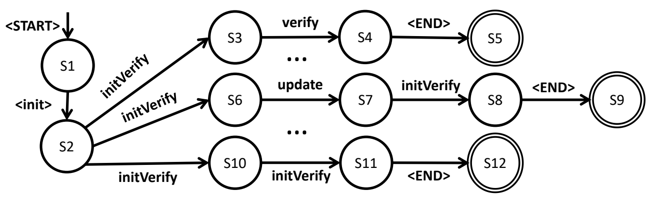


Figure 3: An Example Prefix Tree Acceptor (PTA)

Feature Extraction: From method sequences of the sampled execution traces, we construct a Prefix Tree Acceptor (PTA). A PTA is a tree-like deterministic finite automaton (DFA) created by putting all the prefixes of sequences as states, and a PTA only accepts the sequences that it is built from. The final states of our constructed PTAs are the ones

Table 1: Extracted Features for An Automata State

Feature ID	Value
Type I: Previously Invoked Methods	
F_m	1 if m is invoked before the automaton state. Otherwise, 0.
Type II: Next Methods to Invoke	
P_m	Probability computed by the learned Recurrent Neural Network based Language Model for method m to be called after a particular automaton state is reached ($0 \leq P_m \leq 1$).

have incoming edges with `<END>` labels (see Section 3.2). Figure 3 shows an example of a Prefix Tree Acceptor (PTA). Table 1 shows information of the extracted features. For each state S of a PTA, we are particularly interested in two types of features:

(1) **Type I**: This type of features captures information of previously invoked methods before the state S is reached. The values of type I features for state S is the occurrences of methods on the path between the starting state (i.e., the root of the PTA) and S. For example, according to Figure 3, the values of Type I features corresponding to node S_3 are: $F_{<\text{START}>} = F_{<\text{init}>} = F_{\text{initVerify}} = 1$ and $F_{\text{update}} = F_{\text{verify}} = F_{<\text{END}>} = 0$.

(2) **Type II**: This type of features captures the likely methods to be immediately called after a state is reached. Values of these features are computed by the inferred RNNLM in the deep learning step (see Section 3.2). For example, at node S_3 in Figure 3, `initVerify` and `<END>` have higher probabilities than the other methods to be called afterward. Examples of type II features and their values for node S_3 output by a RNNLM are as follows: $P_{\text{initVerify}} = P_{<\text{END}>} = 0.4$ and $P_{<\text{START}>} = P_{<\text{init}>} = P_{\text{verify}} = P_{\text{update}} = 0.15$.

Our intuition of extracting different types of features is to provide sufficient information for clustering algorithms in the subsequent substep to better merge PTA nodes.

Clustering: We run k-means [35] and hierarchical clustering [43] algorithms on the PTA's states with their extracted features. Our goal is to create a simpler and more generalized automaton that captures specifications of a target library class. Since both k-means and hierarchical clustering require the predefined input C for number of clusters, we try with many values of C from 2 to `MAX_CLUSTER` (refer to Section 4.2.2) to search for the best FSA. Overall, the execution of clustering algorithms results in $2 \times (\texttt{MAX\_CLUSTER} - 1)$ FSAs.

Model Selection: We propose a heuristic to select the best FSA among the ones output by the clustering algorithms. Algorithm 2 describes our strategy to predict precision of an automaton M given the set of all traces *Data* (see Section 3.1).

We predict Precision by first constructing a set P_{Data} containing all pairs (m_1, m_2), where m_1 and m_2 appear consecutively (i.e., m_1 is called right before m_2) in an execution

Algorithm 2: Predicting Precision of a finite-state automaton given a set of method sequence.

```
Input : $M$: an finite-state automaton
        $Data$: a set of training method sequences
Output : Predicted Precision of $M$
1  $P_{Data} \leftarrow$ empty set
2  for $seq \in Data$ do
3      for $0 \leq i < length(seq) - 1$ do
4          Include $(seq[i], seq[i+1])$ to $P_{Data}$
5      end
6  end
7  $P_M \leftarrow$ empty set
8  $E_M \leftarrow$ the set of transitions in $M$
9  for $s_1 \xrightarrow{m_1} s_2 \in E_M \wedge s_2 \xrightarrow{m_2} s_3 \in E_M$ do
10     Include $(m_1, m_2)$ to $P_M$
11 end
12 $precision \leftarrow \frac{|P_{Data}|}{|P_{Data} \cup P_M|}$
13 return $precision$
```

trace in *Data*. Then, we construct another set P_M containing all pairs (m_1, m_2) that appear consecutively in a trace generated by the automaton M. In Algorithm 2, lines 1 to 6 compute all pairs (m_1, m_2) occurring in *Data*, and lines 7 to 11 collect those pairs occurring in traces generated by the input automaton M. To find all pairs occurring in traces generated by M, we look for two transitions $s_1 \xrightarrow{m_1} s_1'$ and $s_2 \xrightarrow{m_2} s_2'$ of M, where $s_1' = s_2$. We take labels of the two transitions (i.e., m_1 and m_2) and add a pair of methods (m_1, m_2) to the set P_M. Line 12 computes the predicted value of Precision, which is the ratio between of number of all pairs in P_M and all pairs in $P_{Data} \cup P_M$. We input all FSAs created by clustering algorithms and all execution traces to Algorithm 2 for estimating the FSAs' Precision.

Next, we approximate the values of Recall by computing the percentage of all execution traces accepted by a given automaton M. Once all precision and recall of FSA models are predicted, we compute the expected value of F-measure (i.e., harmonic mean of precision and recall) for each of the automata. Finally, our approach returns the FSA with highest expected F-measure.

Example: From the PTA partially shown in Figure 3, we conduct *feature extraction* on each state of the PTA. Then, we input these states with extracted features to clustering algorithms (i.e., k-means and hierarchical clustering) to merge similar states. If `MAX_CLUSTER` is identified to be 20, in total, there would be 38 FSAs constructed by the two clustering algorithms. Figure 4 shows 2 out of 19 FSAs output by k-means. Next, we perform *model selection* on these FSAs to select the one with highest predicted F-measures.

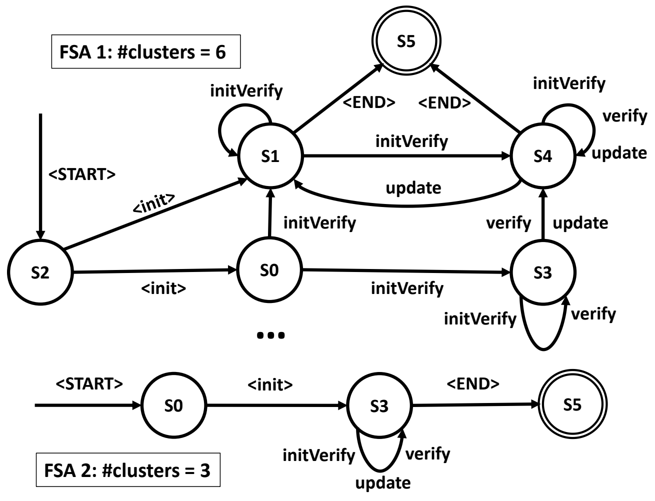


Figure 4: Example FSAs output by clustering algorithms from PTA shown in Figure 3.

Table 2: Target Library Classes. "#M" represents the number of class methods that are analyzed, "#Generated Test Cases" is the number of test cases generated by Randoop, "#Recorded Method Calls" is the number of recorded method calls in the execution traces, "NFST" stands for `NumberFormatStringTokenizer`**.**

Target Library Class	#M	#Generated Test Cases	#Recorded Method Calls
`ArrayList`	18	42,865	22,996
`HashMap`	11	53,396	67,942
`Hashtable`	8	79,403	89,811
`HashSet`	8	23,181	257,428
`LinkedList`	7	13,731	4,847
`NFST`	5	158,998	95,149
`Signature`	5	79,096	205,386
`Socket`	21	80,035	130,876
`StringTokenizer`	5	148,649	336,924
`StackAr`	7	549,648	132,826
`ZipOutputStream`	5	162,971	43,626

4 EMPIRICAL EVALUATION

4.1 Dataset

4.1.1 Target Library Classes. In our experiments, we select 11 target library classes as the benchmark to evaluate the effectiveness of our proposed approach. These library classes were investigated by previous research works in specification mining [24, 26]. Table 2 shows further details of the selected library classes including information of collected execution traces. Among these library classes, 9 out of 11 are from Java Development Kit (JDK); the other two library classes are `DataStructure.StackAr` (from Daikon project) and `NumberFormatStringTokenizer` (from Apache Xalan). For every library class, we consider methods that were analyzed by Krka et al. [24].

4.1.2 Ground Truth Models. We utilize ground truth models created by Krka et al. [24]. Among the investigated library classes, we refine ground truth models of five Java's Collection based library classes (i.e., `ArrayList`, `LinkedList`, `HashMap`, `HashSet`, and `Hashtable`) to capture "empty" and "non-empty" states of `Collection` objects. We also revise ground truth models of `NumberFormatStringTokenizer` and `Socket` by including missing transitions of the original models.

4.2 Experimental Settings

4.2.1 Evaluation Metrics. We follow Lo and Khoo's method [32] to measure precision and recall for assessing the effectiveness of our proposed approach. Lo and Khoo's method has been widely adopted by many prior specification mining works [24, 27]. Their proposed approach takes as input a ground truth and an inferred FSA. Next, it generates sentences (i.e., traces) from the two FSAs to compute their similarities. Precision of an inferred FSA is the percentage of sentences accepted by its corresponding ground truth model among the ones that are generated by that FSA. Recall of an inferred FSA is the percentage of sentences accepted by itself among the ones that are generated by the corresponding ground truth model. In a nutshell, precision reflects the percentage of sentences produced by an inferred model that are correct, while recall reflects the percentage of correct sentences that an inferred model can produce. We use F-measure, which is the harmonic mean of precision and recall, as a summary metric to evaluate specification mining algorithms. F-measure is defined as follows:

$$\textit{F-Measure} = 2 \times \frac{\textit{Precision} \times \textit{Recall}}{\textit{Precision} + \textit{Recall}} \tag{1}$$

To accurately compute precision, recall and F-measure, sentences generated from a FSA must thoroughly cover its states and transitions. To achieve that goal, we set the maximum number of generated sentences to 10,000 with maximal length of 50, and minimum coverage of each transition equals to 20. Similar strategies were adopted in prior works [24, 26].

4.2.2 Experimental Configurations & Environments.

Randoop Configuration. In test case generation step, for each target class, we repeatedly execute Randoop (version 3.1.2) with a time limit of 5 minute with 20 different initial seeds. We set the time limit to 5 minutes to make sure subsequent collected execution traces are not too long as well as not too short. We repeat execution of Randoop 20 times to maximize the coverage of possible sequences of program methods in Randoop generated test cases. Furthermore, we turn off Randoop's option of generating error-revealing test cases (i.e., `--no-error-revealing-tests` is set to `true`) as executions of these test cases are usually interrupted by exceptions or errors, which results in incomplete method sequences for subsequent deep learning process. We find that with this setup the generated traces cover 100% of target API methods; also, on average, 96.97% and 98.18% of edges and states in each target class' ground-truth model are covered.

RNNLM. We use a publicly available implementation of RNNLM based on TensorFlow[4]. We execute this implementation on a NVIDIA DGX-1 Deep Learning system. We use the default configuration included as part of the implementation.

Clustering Configuration. In clustering step, we run k-means and hierarchical clustering with the setting of number of clusters from 2 to MAX_CLUSTER = 20 for every target class. We use `sklearn.cluster.KMeans` and `sklearn.cluster.AgglomerativeClustering` of scikit-learn (version 0.18) with default settings.

4.2.3 Baselines. In the experiments, we compare the effectiveness of DSM with many previous specification mining works . Krka et al. propose a number of algorithms that analyze execution traces to infer FSAs [24]. These algorithms are k-tails, CONTRACTOR++, SEKT, and TEMI. CONTRACTOR++, TEMI, and SEKT infer models leveraging invariants learned using Daikon. On the other hand, k-tails construct models only from ordering of methods in execution traces. Despite the fact that DSM is not processing likely invariants, we include CONTRACTOR++, SEKT, and TEMI as baselines to compare the applicability of deep learning and likely invariant inference in specification mining. For k-tails and SEKT, we choose $k \in \{1, 2\}$ following Krka et al. [24] and Le et al. [26]'s configurations. In total, we have six different baselines: Traditional 1-tails, Traditional 2-tails, CONTRACTOR++, SEKT 1-tails. SEKT 2-tails, and TEMI

We use the implementation of provided by Krka et al.[5] [24]. We utilize Daikon [14] to collect execution traces from Randoop generated test cases and inferred likely invariants from all of the traces. Originally, Krka et al.'s implementation uses a 32-bit version of Yices 1.0 Java Lite API[6], which only works with 32-bit Java Virtual Machine and limited to use up to 4 GB heap memory. Since the amount of execution traces is large, we follow two experimental schemes to run Krka et al.'s code:

(1) **Scheme I:** Krka et al.'s implementation is updated to work with the 64-bit libraries of Yices 1 SMT solver[7]. Then, we input execution traces of all generated test case as well as Daikon invariants inferred by these traces to all baselines. For each application of Krka et al.'s code, we set the maximum allocated memory to 7 GB and time limit to 12 hours.
(2) **Scheme II:** We use the original Krka et al.'s implementation and a set of execution traces corresponding to test cases generated by Randoop with *one* specific seed.

4.3 Research Questions

RQ1: How effective is DSM? In this research question, we compute precision, recall, and F-measure of FSAs inferred by our approach for the 11 target library classes.

[4]https://github.com/hunkim/word-rnn-tensorflow
[5]http://softarch.usc.edu/wiki/doku.php?id=inference:start
[6]http://atlantis.seidenberg.pace.edu/wiki/lep/Yices_Java_API_Lite
[7]http://yices.csl.sri.com/old/download-yices1.shtml

Table 3: Effectiveness of DSM. "F" is F-measure.

Class	F (%)	Class	F (%)
`ArrayList`	22.21	`Signature`	100.00
`HashMap`	86.71	`Socket`	54.24
`HashSet`	76.84	`StackAr`	74.38
`Hashtable`	79.92	`StringTokenizer`	100.00
`LinkedList`	30.98	`ZipOutputStream`	88.82
`NFST`	77.52%	**Average**	71.97

RQ2: How does DSM compare to existing specification mining algorithms? In this research question, we compare DSM with a number of existing specification mining algorithms proposed by Krka et al. [24] in various experimental schemes.

RQ3: Which Recurrent Neural Network (RNN) is best for DSM? By default, DSM trains RNNLMs using Long-Short Term Memory (LSTM) networks from execution traces. In this research question, we first adapt DSM to use the standard RNN and Gated Recurrent Units (GRU) [9] networks for constructing language models with the same learning configuration as LSTM (see Section 3.2). Then, we analyze the effectiveness of DSM for each neural network architecture (i.e., Standard RNN, LSTM, and GRU).

4.4 Findings

RQ1: DSM's Effectiveness. Table 3 shows the F-measure of DSM for the eleven target library classes (Section 4.1). From the table, our approach achieves an average F-measure of 71.97%. Noticeably, for `StringTokenizer` and `Signature`, DSM infers models that exactly match ground truth models (i.e., F-measure of 100%). There are other 6 out of the 11 library classes where our approach achieves F-measure of 70% or greater.

By taking a closer look into DSM intermediary results, we find that across the 11 classes, DSM performs 26 - 1,072 merge operations (with an average of 303.45 operations) to construct the final FSAs from the PTAs. Also, the differences in the predicted F-measures of the FSAs produced in the clustering step range between 22.05% - 68.32% (with an average of 39.13%). These shows that DSM components need to perform substantial amount of work to bring the PTAs to the final FSAs.

RQ2: DSM vs. Previous Works.

Scheme I: The first part of Table 4 highlights the F-measure of 6 baselines proposed by Krka et al. [24] following Scheme I. In this experimental scheme, we use all of execution traces that we collect from generated test cases and likely invariants inferred from these traces as input data to the baselines. According to the figure, CONTRACTOR++ is the most effective baseline in terms of average F-measure (i.e., 55.22%) in this experimental scheme; Traditional 1-tails is the best performing baseline that only uses execution traces to construct FSAs (i.e., average F-measure of 33.42%). DSM is more effective than all of the baselines in terms of average precision, recall, and F-measure. In terms of average F-measure,

Table 4: F-measure (%): Scheme I vs. Scheme II. "T1" is Traditional 1-tails, "T2" is Traditional 2-tails, "C+" is CONTRACTOR++, "S1" is SEKT 1-tails, "S2" is SEKT 2-tails, "OT" is Optimistic TEMI, "NFST" is NumberFormatStringTokenizer, "-" means that the result is not available.

Target Library Class	Scheme I						Scheme II					
	T1	T2	C+	S1	S2	OT	T1	T2	C+	S1	S2	OT
`ArrayList`	13.96	13.13	36.03	13.86	13.07	16.87	5.61	3.08	36.03	4.34	3.08	35.82
`HashMap`	25.41	8.71	68.94	-	-	-	37.35	21.93	68.94	37.35	21.93	68.94
`HashSet`	20.88	21.27	52.22	20.88	21.27	23.34	22.96	15.35	52.22	22.96	15.35	55.62
`Hashtable`	42.39	33.58	92.78	-	-	-	78.42	73.37	92.78	81.69	65.47	92.78
`LinkedList`	27.15	25.72	86.02	26.67	24.52	7.51	26.03	8.07	86.02	18.45	6.59	86.02
`NFST`	24.57	25.52	30.40	24.56	25.78	11.80	68.72	51.04	30.40	66.58	49.51	33.40
`Signature`	61.54	64.25	66.88	62.05	63.98	39.06	100	95.41	66.88	95.41	89.78	66.88
`Socket`	35.89	31.52	55.15	34.73	28.37	-	37.30	21.98	55.15	35.32	20.87	55.62
`StackAr`	16.54	16.54	34.91	16.54	16.54	-	42.57	42.57	34.91	42.57	42.57	-
`StringTokenizer`	52.88	52.97	21.30	52.15	-	-	100	89.69	21.30	92.21	84.77	0.00
`ZipOutputStream`	46.36	47.42	62.80	47.91	-	-	82.51	78.08	62.08	86.47	67.84	66.20
Average	33.42	30.97	55.22	33.26	27.65	19.72	54.70	45.50	55.22	53.03	42.52	56.13

we outperform CONTRACTOR++ (i.e., the best performing baseline) and Traditional 1-tails (i.e., the best baseline that only use execution traces) by 30.33% and 115.35%, respectively. Furthermore, we note many baselines have no available results for precision, recall, and F-measure. This is because these baselines are not able to return FSAs mostly due to out of heap memory errors or time limit exceeded errors.

Scheme II. The second part of Table 4 shows the F-measure of 6 baselines proposed by Krka et al. [24] following Scheme II. Optimistic TEMI is the best performing baseline in terms of average F-measure; Traditional 1-tails is the best baseline that constructs models from execution traces with the average F-measure of 53.97%. In comparison with DSM, we note that the average recall, and F-measure of our approach is higher than those of all the six baselines. That indicates our inferred models are more generalized and less overfitted. In terms of average F-measure, our approach outperforms the best baseline (i.e., Optimistic TEMI) by 28.22%. DSM is also more effective than Traditional 1-tails (i.e., the best baseline that infers FSAs only from method orderings in traces) by 31.57%. Noticeably, compared to the baselines that construct automata from execution traces (i.e., Traditional 1-tails and Traditional 2-tails), DSM's F-measures are higher for all target library classes. For the other baselines that use likely invariants, our approach is more effective in terms of F-measures for 7 out of 11 target library classes except `ArrayList`, `LinkedList`, `Hashtable`, and `Socket`.

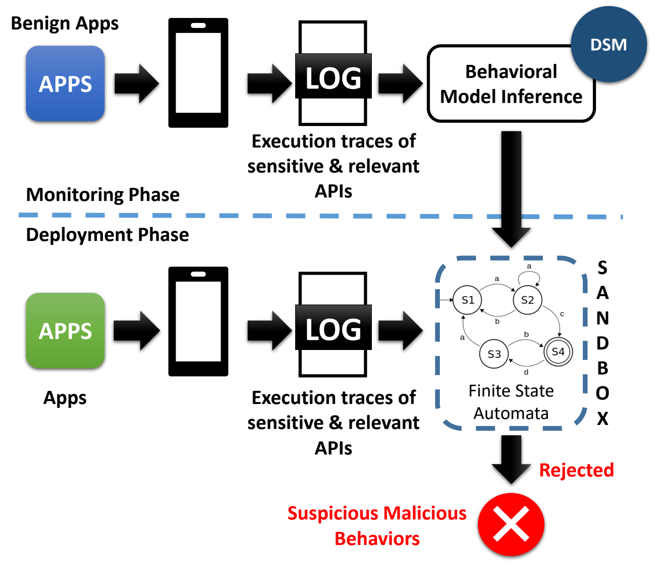
RQ3: Best RNN Architecture. Table 5 shows the effectiveness of DSM configured with the three different RNN architectures. We can note that DSM$^{\text{LSTM}}$ (our default configuration) and DSM$^{\text{GRU}}$ outperform DSM$^{\text{RNN}}$ in terms of F-measure by 7.92% and 6.88% respectively. DSM$^{\text{GRU}}$ performs almost equally as well as DSM$^{\text{LSTM}}$.

Table 5: F-measure (%): DSM with standard RNN (DSM$^{\text{RNN}}$), LSTM (DSM$^{\text{LSTM}}$), and GRU (DSM$^{\text{GRU}}$).

Target Library Class	DSM$^{\text{LSTM}}$	DSM$^{\text{RNN}}$	DSM$^{\text{GRU}}$
`ArrayList`	22.21	4.95	9.39
`HashMap`	86.71	97.76	100.00
`HashSet`	76.84	68.86	73.88
`Hashtable`	79.92	87.94	87.94
`LinkedList`	30.98	32.29	34.86
`NFST`	77.52	70.09	73.31
`Signature`	100.00	65.31	95.41
`Socket`	54.24	51.31	58.34
`StackAr`	74.38	71.79	77.67
`StringTokenizer`	100.00	95.88	100.00
`ZipOutputStream`	88.82	87.36	73.29
Average	71.97	66.69	71.28

5 MINING FSA FOR DETECTING ANDROID MALICIOUS BEHAVIORS

Nowadays, Android is the most popular mobile platform with millions of apps and supported devices. As the matter of fact, Android users easily become targets of attackers. Recently, several approaches have been proposed to protect users from potential threats of malware. Among state-of-the-art approaches, Jamrozik et al. propose Boxmate that mines rules to construct Android sandboxes by exploring behaviors of target benign apps [21]. The key idea of Boxmate is it prevents a program to change its behaviour; it can prevent hidden attacks, backdoors, and exploited vulnerabilities from compromising the security of an Android app. Boxmate works on two phases: monitoring and deployment. In the monitoring phase, Boxmate employs a test case generation tool, named Droidmate [22], to create a rich set of GUI test cases. During the execution of these test cases, Boxmate records invocations



Benign Apps
APPS
LOG
Behavioral Model Inference
DSM
Execution traces of sensitive & relevant APIs
Monitoring Phase
Deployment Phase
APPS
Apps
LOG
Execution traces of sensitive & relevant APIs
S1
S2
S3
S4
Finite State Automata
SANDBOX
Rejected
Suspicious Malicious Behaviors

Figure 5: Malware detection framework leveraging behavior models inferred by DSM

of sensitive API methods (e.g., methods that access cameras, locations, etc.), and use them to create sandbox rules. The rules specify what sensitive API methods are allowed to be invoked during deployment. During deployment, when an app accesses a sensitive API method that is not recorded in the above rules, the sandbox immediately intercepts that operation and raises warning messages to users about the suspicious activity.

Boxmate's mined sandbox rules constitute a behavior model that accepts a sensitive API method as input and predicts its invocation as *benign* or *malicious*. Nevertheless, we find that attackers can bypass Boxmate's models if they can perform malicious activities by invoking the same sensitive APIs as those used by the apps during their normal operations. Therefore, we propose a technique to employ FSAs inferred by DSM to construct more comprehensive sandboxes that consider the *context of sensitive API invocations* – namely series of methods called prior to the sensitive API invocations – that can better protect Android users from attackers. For each sensitive API method M, we select W[8] previously executed non-sensitive API methods before M to create training traces. Then, we group all of the training traces into one single set and input it to DSM. The output of DSM is a FSA that captures interactions between sensitive and non-sensitive API methods. The FSA is then leveraged as an automaton based behavior model to build a sandbox for more effective protection.

Figure 5 demonstrates the proposed framework of our malware detection system. According to the figure, our framework has two phases:

- **Monitoring Phase:** This phase accepts as input a benign version of the target Android app. We first leverage GUI test case generation tools (i.e., Monkey [1], GUIRipper [2], PUMA [18], Droidmate [22], and Droidbot [30]) to create a diverse set of test cases. Next, we execute the input app with generated test cases and monitor API methods called. In particular, every time the app invokes a sensitive API method X, we select a sequence of W previously executed API methods before X and include them to the training traces. Then, we employ DSM's mining algorithm on the gathered traces to construct a FSA based behavior model BM. The constructed model reflects behaviors of the app when calling sensitive API methods. Subsequently, we employ BM to guide an automaton based sandbox in the deployment phase for malware detection.
- **Deployment Phase:** In this phase, our framework leverages the inferred model BM to build an automaton based sandbox. The sandbox is used to govern and control execution of an Android app. Every time an app invokes a sensitive API X, the sandbox selects the sequence of W previously executed methods before X, and input them to the behavior model BM to classify the invocation of X as *malicious* or *benign*. If execution of X is predicted as *malicious* by model BM, the sandbox informs users by raising warning messages about suspicious activities. Otherwise, the sandbox allows the app to continue its executions without notifying users.

We evaluate our proposed malware detection framework using a dataset of 102 pairs of Android apps that were originally collected by Li et al. [29]. Each pair of apps contains one benign app and its corresponding malicious version. The malicious apps are created by injecting malicious code to their corresponding unpacked benign apps [29]. All these apps are real apps that are released to various app markets. Recently, Bao et al. [4] used the above 102 pairs to assess the effectiveness of Boxmate's mined rules with 5 different test case generation tools (i.e., Monkey [1], GUIRipper [2], PUMA [18], Droidmate [22], and Droidbot [30]). In our evaluation, we utilize execution traces of the 102 Android app pairs collected by Bao et al. [4][9]. We set W (i.e., number of selected methods before an invoked sensitive API method) to 3, and employ these traces to infer several behavior models by using Boxmate, DSM as well as k-tails ($k = 1$). We include k-tails ($k = 1$) since according to Table 4 this is the best baseline mining algorithm that infers FSAs from raw traces of API invocations. We let the comparison between DSM and invariant based miners (i.e., CONTRACTOR++ and TEMI) for future work as Daikon is currently unable to mine invariants for Android apps. Next, we evaluate the effectiveness of inferred behavior models in detecting malware using the following evaluation metrics:

- **True Positive Rate (TPR):** Percentage of apps that are correctly classified as malicious. TPR is computed as follows

$$TPR = \frac{TP}{TP + FN} \tag{2}$$

- **False Positive Rate (FPR):** Percentage of apps that are incorrectly classified as malicious. FPR is computed as

[8] By default, we set $W = 3$.

[9] https://github.com/baolingfeng/SANER2018Sandboxes

Table 6: DSM vs. Baselines. "APR" stands for "Approach", "TP" is True Positives, "FN" is False Negatives, "TN" is True Negatives, "FP" is False Positives, "TPR" is "True Positive Rate", and "FPR" is "False Positive Rate".

APR	TP	FN	TN	FP	TPR	FPR
DSM	93	9	398	112	91.18%	21.97%
k-tails	94	8	350	160	92.16%	31.37%
Boxmate	77	25	421	89	75.49%	17.45%

follows

$$FPR = \frac{FP}{FP + TN} \tag{3}$$

In the above equations, TP (true positives) refers to the number of malicious apps that are classified as malicious, TN (true negatives) refers to the number of benign apps that are classified as benign, FP (false positives) refers to the number of benign apps are classified as malicious, and FN (false negatives) refers to the number of malicious apps that are classified as benign. Both TPR and FPR are important; TPR is important as otherwise malicious behaviors are permitted, while FPR is as important as otherwise users would be annoyed with false warnings. The two metrics are well-known and widely adopted by state-of-the-art approaches in Android malware detection (e.g., [3, 50]).

To compute True Positives and False Negatives, for each pair of benign and malicious app we employ DSM, k-tails, and Boxmate to construct behavior models using training traces of the benign apps, and deploy these models on traces of the corresponding malicious app. Following Equation 2, we use the values of True Positives and False Negatives to estimate True Positive Rates (TPR). The higher the values of TPR, the more malicious activities are detected. On the other hand, to calculate True Negatives and False Positives, for each benign Android app we perform cross-validation among the five test case generation tools (i.e., Monkey [1], GUIRipper [2], PUMA [18], Droidmate [22], and Droidbot [30]). In particular, we use execution traces of 4 tools as training data to learn behavior models by DSM, k-tails and Boxmate. Then, we deploy the inferred models on traces of the remaining tool to check whether the models detect benign apps as *malicious* (i.e., false positive). In total, we analyze $5 \times 102 = 510$ combinations between benign apps and test case generation tools. Then, we utilize values of True Negatives and False Positives to compute False Positive Rate (FPR) (see Equation 3). The smaller the values of FPR, the smaller the number of false alarms.

Table 6 shows results for DSM, k-tails, and Boxmate. Boxmate can detect 75.49% of the malicious apps as such while suffering a false positive rate of 17.43%.[10] We note that DSM outperforms Boxmate in terms of True Positive Rate by 15.69% while only losing in terms of False Positive Rate by 4.52%. Comparing DSM with k-tails, we note that they have similar True Positive Rate (difference is less than 1%), but the latter has substantially higher False Positive Rate (difference is close to 10%). Clearly, DSM achieves the best trade-off considering both True Positive Rate and False Positive Rate.

[10]This is inline with results that were reported in Boxmate's original paper [21]. They reported that for the 18 use cases considered, two false alarms were raised (see Table 2 of their paper) – this results in a False Positive Rate of 11.11%. In our experiment, we consider many more use cases with the aid of different test case generation tools. Jamrozik et al. did not report true positive rate since no malicious apps are considered in the study.

6 THREATS TO VALIDITY

Threats to internal validity. We have carefully checked our implementation, but there are errors that we did not notice. There are also potential threats related to correctness of ground truth models created by Krka et al. [24] that we used. To mitigate this threat, we have compared their models against execution traces collected from Randoop generated test cases as well as textual documentations published by library class writers (e.g., Javadocs). We revised the ground truth models accordingly.

Another threat to validity is related to parameter values of target API methods. We use traces collected by Bao et al. [4] which exclude *all* parameter values. This is different from Jamrozik et al.'s work [21] that excludes *most* (but not all) parameter values. We decide to exclude all parameter values since all specification mining algorithms considered in this paper (including DSM) produce FSAs that have no constraints on values of parameters. As future work, we plan to extend DSM to generate models that include constraints on parameter values.

Threats to External Validity. These threats correspond to the generalizability of our empirical findings. In this work, we have analyzed 11 different library classes. This is larger than the number of target classes used to evaluate many prior studies, e.g., [24, 33, 34]. As future works, we plan to reduce this threat by analyzing more library classes to infer their automaton based specifications.

Threats to Construct Validity. These threats correspond to the usage of evaluation metrics. We have followed Lo and Khoo's approach that uses precision, recall, and F-measure to measure the accuracy of automata output by a specification mining algorithm against ground truth models [32]. Furthermore, Lo and Khoo's approach is well known and has been adopted by many previous research works in specification mining e.g., [5, 6, 8, 13, 24, 27, 31, 33]. Additionally, True Positive Rate and False Positive Rate are well-known metrics and widely adopted by state-of-the-art approaches in Android malware detection (e.g., [3, 50]).

7 RELATED WORK

Mining Specifications. Aside from the state-of-the-art baselines considered in Section 4, there are other related works that mine FSA-based specifications from execution traces. Lo et al. propose SMArTIC that mines a FSA from a set of execution traces [13] using a variant of k-tails algorithm that constructs a probabilistic FSA. Mariani et al. propose k-behavior [36] that constructs an automaton by analyzing one single trace at a time. Walkinshaw and Bogdanov propose an approach that allows users to input temporal properties to support a specification miner to construct a FSA from execution traces [45]. Lo et al. further extend Walkinshaw and Bogdanov's work to automatically infer temporal properties from execution traces, and use these properties to automatically support model inference process of a specification miner [33]. Synoptic infers three kinds of temporal invariants from execution traces and uses them to generate a concise FSA [6]. SpecForge [26] is a meta-approach that analyzes FSAs inferred by other specification miners and combine them together to create a more accurate FSA. None of the above mentioned approaches employs deep learning.

Deep Learning for Software Engineering Tasks. Recently, deep learning methods are proposed to learn representations of data with multiple levels of abstraction [28]. Researchers have been utilizing deep learning to solve challenging tasks in software engineering [17, 25, 47–49]. For example, Gu et al. propose DeepAPI that takes as input queries in natural languages and outputs sequences of API methods that developers should follow [17]. In the nutshell, DeepAPI replies on a RNN based model that can translate a sentence in one language to a new sentence in another language. Different from DeepAPI, DSM takes as input method sequences of an API or library and outputs a finite-state automaton that represents behaviors of that API or library. Prior to our work, deep learning models have not been employed to effectively mine specifications.

Language Models for Software Engineering Tasks. Statistical language models have been utilized for many software engineering tasks. For example, Hindle et al. employ n-gram model on code tokens to demonstrate a high degree of local repetitiveness in source code corpora and leverage it to improve Eclipse's code completion engine [19]. Several other works have extended Hindle et al. work to build more powerful code completion engines; for example, Raychev et al. leverage n-gram and recurrent neural network language model to recommend likely sequences of method calls to a program with holes [41], while Nguyen et al. leverage Hidden Markov Model to learn API usages from Android app bytecode for recommending APIs [40]. Beyond code completion, Wang et al. use n-gram model to detect bugs by identifying low probability token sequences [46]. Our work uses language model for a different task.

8 CONCLUSION AND FUTURE WORK

Formal specifications are helpful for many software processes. In this work, we propose DSM, a new approach that employs Recurrent Neural Network Based Language Model (RNNLM) for mining FSA-based specifications. We apply Randoop, a well-known test cases generation approach, to create a richer set of execution traces for training RNNLM. From a set of sampled execution traces, we construct a Prefix Tree Acceptor (PTA) and extract many features of PTA's states using the learned RNNLM. These features are then utilized by clustering algorithms to merge similar automata states to construct many FSAs using various settings. Then, we employ a model selection heuristic to select the FSA that is estimated to be the most accurate and output it as the final model. We run our proposed approach to infer specifications of 11 target library classes. Our results show that DSM achieves an average F-measure of 71.97%, outperforming the best performing baseline by 28.82%. Additionally, we propose a technique that employs a FSA mined by DSM to detect malicious behaviors in an Android app. In particular, our technique uses the inferred FSA as a behavior model to construct a more comprehensive sandbox that considers execution context of sensitive API methods. Our evaluation shows that the proposed technique can improve the True Positive Rate of Boxmate by 15.69% while only increasing the False Positive Rate by 4.52%.

As future work, we plan to improve DSM's effectiveness further by integrating information of likely invariants into our deep learning based framework. We also plan to employ EvoSuite [16] and many other test case generation tools to generate an even more comprehensive set of training traces to improve the effectiveness of DSM. Furthermore, we plan to improve DSM by considering many more clustering algorithms aside from k-means and hierarchical clustering, especially the ones that require no inputs for the number of clusters (e.g., DBSCAN [15], etc.). Finally, we plan to evaluate DSM with more classes and libraries in order to reduce threats to external validity.

Additional Resources and Acknowledgement. Our dataset, DSM's implementation, and a technical report that includes additional results are publicly available at: https://github.com/lebuitienduy/DSM. This research was supported by the Singapore National Research Foundations National Cybersecurity Research & Development Programme (award number: NRF2016NCR-NCR001-008).

REFERENCES

[1] Last accessed on February 25, 2017. In *https://github.com/linkedin/camus.*

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012.* 258–261.

[3] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014.*

[4] Lingfeng Bao, Tien-Duy B. Le, and David Lo. 2018. Mining sandboxes: Are we there yet?. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018.* 445–455.

[5] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D. Ernst, and Arvind Krishnamurthy. 2015. Using Declarative Specification to Improve the Understanding, Extensibility, and Comparison of Model-Inference Algorithms. *IEEE Trans. Software Eng.* 41, 4 (2015), 408–428.

[6] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European*

conference on Foundations of software engineering. ACM, 267–277.

[7] Alan W Biermann and Jerome A Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.* 100, 6 (1972), 592–597.

[8] Zherui Cao, Yuan Tian, Tien-Duy B Le, and David Lo. [n. d.]. Rule-based specification mining leveraging learning to rank. *Automated Software Engineering* ([n. d.]), 1–30.

[9] Junyoung Chung, Çaglar Gülçehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *CoRR* abs/1412.3555 (2014).

[10] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 1999. *Model Checking*. MIT Press, Cambridge, MA, USA.

[11] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Gordon Fraser, Sebastian Hack, and Andreas Zeller. 2012. Automatically Generating Test Cases for Specification Mining. *IEEE Trans. Software Eng.* 38, 2 (2012), 243–257.

[12] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. 2010. Generating test cases for specification mining. In *Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010*. 85–96.

[13] David Lo and Siau-Cheng Khoo. 2006. SMArTIC: towards building an accurate, robust and scalable specification miner. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*. 265–275.

[14] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.* 69, 1-3 (2007), 35–45.

[15] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*. 226–231.

[16] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 416–419.

[17] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*. 631–642.

[18] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *The 12th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'14, Bretton Woods, NH, USA, June 16-19, 2014*. 204–217.

[19] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar T. Devanbu. 2012. On the naturalness of software. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 837–847.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[21] Konrad Jamrozik, Philipp von Styp-Rekowsky, and Andreas Zeller. 2016. Mining sandboxes. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 37–48.

[22] Konrad Jamrozik and Andreas Zeller. 2016. DroidMate: a robust and extensible test generator for Android. In *Proceedings of the International Conference on Mobile Software Engineering and Systems, MOBILESoft '16, Austin, Texas, USA, May 14-22, 2016*. 293–294.

[23] John C. Knight, Colleen L. DeJong, Matthew S. Gibble, and Lus G. Nakano. 1997. Why Are Formal Methods Not Used More Widely?. In *Fourth NASA Formal Methods Workshop*. 1–12.

[24] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. 178–189.

[25] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. 2015. Combining Deep Learning with Information Retrieval to Localize Buggy Files for Bug Reports (N). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 476–481.

[26] Tien-Duy B. Le, Xuan-Bach D. Le, David Lo, and Ivan Beschastnikh. 2015. Synergizing Specification Miners through Model Fissions and Fusions (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 115–125.

[27] Tien-Duy B. Le and David Lo. 2015. Beyond support and confidence: Exploring interestingness measures for rule-based specification mining. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*. 331–340.

[28] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

[29] Li Li, Daoyuan Li, Tegawendé F. Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Trans. Information Forensics and Security* 12, 6 (2017), 1269–1284.

[30] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*. 23–26.

[31] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 557–566.

[32] David Lo and Siau-Cheng Khoo. 2006. QUARK: Empirical Assessment of Automaton-based Specification Miners. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*. 51–60.

[33] David Lo, Leonardo Mariani, and Mauro Pezzè. 2009. Automatic steering of behavioral model inference. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*. 345–354.

[34] David Lo, Leonardo Mariani, and Mauro Santoro. 2012. Learning extended FSA from software: An empirical assessment. *Journal of Systems and Software* 85, 9 (2012), 2063–2076.

[35] James MacQueen et al. 1967. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, Vol. 1. Oakland, CA, USA., 281–297.

[36] Leonardo Mariani, Fabrizio Pastore, and Mauro Pezzè. 2011. Dynamic Analysis for Diagnosing Integration Faults. *IEEE Trans. Software Eng.* 37, 4 (2011), 486–508.

[37] Weikai Miao and Shaoying Liu. 2012. A Formal Specification-Based Integration Testing Approach. In *SOFL*. 26–43.

[38] Tomas Mikolov, Anoop Deoras, Stefan Kombrink, Lukás Burget, and Jan Cernocký. 2011. Empirical Evaluation and Combination of Advanced Language Modeling Techniques. In *INTERSPEECH 2011, 12th Annual Conference of the International Speech Communication Association, Florence, Italy, August 27-31, 2011*. 605–608.

[39] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. 2010. Recurrent Neural Network Based Language Model. In *Eleventh Annual Conference of the International Speech Communication Association*.

[40] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. 2016. Learning API usages from bytecode: a statistical approach. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 416–427.

[41] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 419–428.

[42] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. 2011. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In *ASE 2011: Proceedings of the 26th Annual International Conference on Automated Software Engineering*.

Lawrence, KS, USA, 23–32.

[43] Lior Rokach and Oded Maimon. 2005. Clustering Methods. In *The Data Mining and Knowledge Discovery Handbook*. 321–352.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*. 3104–3112.

[45] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring Finite-State Models with Temporal Constraints. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*. 248–257.

[46] Song Wang, Devin Chollak, Dana Movshovitz-Attias, and Lin Tan. 2016. Bugram: bug detection with n-gram language models. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 708–719.

[47] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 297–308.

[48] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. 87–98.

[49] Martin White, Christopher Vendome, Mario Linares Vásquez, and Denys Poshyvanyk. 2015. Toward Deep Learning Software Repositories. In *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. 334–345.

[50] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. 1105–1116.

[51] Hao Zhong and Zhendong Su. 2013. Detecting API documentation errors. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 803–816.