Using Finite-State Models for Log Differencing

Hen Amar  
Tel Aviv University  
Israel

Lingfeng Bao∗  
Zhejiang University City College  
China

David Lo  
Singapore Management University  
Singapore

Nimrod Busany  
Tel Aviv University  
Israel

Shahar Maoz  
Tel Aviv University  
Israel

ABSTRACT

Much work has been published on extracting various kinds of models from logs that document the execution of running systems. In many cases, however, for example in the context of evolution, testing, or malware analysis, engineers are interested not only in a single log but in a set of several logs, each of which originated from a different set of runs of the system at hand. Then, the difference between the logs is the main target of interest.

In this work we investigate the use of finite-state models for log differencing. Rather than comparing the logs directly, we generate concise models to describe and highlight their differences. Specifically, we present two algorithms based on the classic k-Tails algorithm: 2KDif, which computes and highlights simple traces containing sequences of k events that belong to one log but not the other, and nKDif, which extends k-Tails from one to many logs, and distinguishes the sequences of length k that are common to all logs from the ones found in only some of them, all on top of a single, rich model. Both algorithms are sound and complete modulo the abstraction defined by the use of k-Tails.

We implemented both algorithms and evaluated their performance on mutated logs that we generated based on models from the literature. We conducted a user study including 60 participants demonstrating the effectiveness of the approach in log differencing tasks. We have further performed a case study to examine the use of our approach in malware analysis. Finally, we have made our work available in a prototype web-application, for experiments.

CCS CONCEPTS

• Software and its engineering → Software testing and debugging;

KEYWORDS

log analysis, model inference

∗Lingfeng Bao was affiliated with Singapore Management University, Singapore when this work was performed.

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 others than ACM must be honored. 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.

ESEC/FSE ’18, November 4–9, 2018, Lake Buena Vista, FL, USA  
© 2018 Association for Computing Machinery.  
ACM ISBN 978-1-4503-5573-5/18/11... $15.00  
https://doi.org/10.1145/3236024.3236069

1 INTRODUCTION

Logs, which document the execution of running systems, contain valuable information about their behavior. Much work has been published on extracting various kinds of models from such logs, from finite-state machines that approximate the behavior of the system that generated the log to candidate temporal properties that characterize their behavior over time (see, e.g., [8, 22, 28, 31, 39]). These models and properties may provide useful information for engineers, for tasks such as debugging, testing, and comprehension.

In many cases, however, engineers are interested not only in a single log but in a set of several logs, which each originated from a different set of runs of the system at hand. For example, in the context of evolution, an engineer may be interested in the differences between logs generated by different versions of the system. Such differences may highlight bugs or new functionality that have been eliminated or introduced. As another example, in the context of testing and deployment, an engineer may be interested in the differences between logs generated by a system when deployed in different environments, e.g., lab tests vs. field. Then, the differences may highlight, e.g., behaviors that occur in the field but not in the lab, and thus may call for updates to the lab tests. As another example, in the context of malware analysis, engineers may be interested in finding the differences between two versions of a system, the original one and a suspected infected one, and try to identify these differences based on logs produced by runs of the two systems. In all these cases, one does not require an analysis of each log alone, but rather a comparative analysis, which focuses on the differences between a number of available logs.

As logs are long, complex, and often very detailed, a direct comparison between them is neither feasible nor desired. Instead, one would be interested in a concise, expressive, and engineer-friendly representation of their differences. This motivates the use of models.

In this work, we investigate the use of finite-state machine (FSM) models for log differencing. Rather than comparing the logs themselves, we generate concise models that describe and highlight their differences. Specifically, we present two very different algorithms, both based on the classic k-Tails algorithm [9]. k-Tails has been
implemented and used over the last two decades in several variants, by many, e.g., [3, 8, 13, 24, 25, 28, 32]. Roughly, given a log and a positive integer \( k \), \( k \)-Tails extracts a FSM that over-approximates the system that generated the log using \( k \)-sequences, i.e., event sequences of length \( k \) or less.

One may suggest the following approach to compare logs using models: build a model from each log alone, e.g., using \( k \)-Tails, and then compare the models, e.g., by intersecting one with the complement of the other, or by enumerating traces that are accepted by one and not the other. Indeed this was recently suggested in [16]. However, this approach has a number of weaknesses in terms of soundness, completeness, and performance. Since the models over-approximate the logs, their comparison may yield spurious differences, which have no evidence in the logs. Moreover, a complete comparison may become very costly for inferred models, which typically include loops and non-determinism. Finally, this approach does not scale to compare many logs because it requires combinatorially many comparisons between subsets of logs. Our approach is different.

First, we present 2KDiff, a basic algorithm to compare two logs. 2KDiff compares two logs by focusing on their \( k \)-differences, i.e., \( k \)-sequences that belong to one log but not the other. 2KDiff computes the set of \( k \)-differences and selects from the logs representative traces containing them. Then, it computes the \( k \)-Tails FSM of each log separately, and presents the selected traces on top of them, with the \( k \)-differences highlighted. 2KDiff is limited to compare two logs.

Second, we present an advanced algorithm we call nKDiff, which extends the classic \( k \)-Tails algorithm: it takes a set of \( n \) logs as input, and it outputs a single labeled FSM that represents their differences. The labeling, consisting of subsets of \( \{1, \ldots, n\} \), provides two-way traceability between the behaviors in the \( n \) input logs and the behaviors induced by the labeled FSM. nKDiff is built to compare many logs at once.

Most importantly, both algorithms, 2KDiff and nKDiff, guarantee soundness and completeness modulo the \( k \)-Tails abstraction, i.e., their over-approximation is not worse than the over-approximation induced by the use of \( k \)-Tails. They do not yield spurious differences, and they do yield all differences. We present the two algorithms and discuss their properties in Sect. 4.

To evaluate our work, we implemented the two algorithms, validated the implementations, and evaluated their performance on mutated logs generated based on publicly available, non-trivial models from the literature. We conducted a user study including 60 participants, which were given log comparison tasks. We measured both the correctness and the time required to perform each task using our approach and two alternative tools as baselines. The results demonstrate the effectiveness of the approach in improving both aspects. We have further performed a case study to examine the use of our approach in Android malware analysis. Finally, we have made our work available in a prototype web-application, for experiments [1]. We present the evaluation in Sect. 5.

While much work has been published on extracting various kinds of models and temporal properties from logs, see, e.g., [8, 22, 28, 31, 39], almost no work has considered the problem of using models for log differencing. We discuss related work in Sect. 6.

![Figure 1: Four logs of the shopping cart system in four columns, \( L_1 \) to \( L_4 \) (left to right), each consists of several traces, separated by ‘- -’.](image-url)

### 2 EXAMPLE

In [8], Beschastnikh et al. use an example of a shopping cart. The log used in their example contains a bug, where the user can use an invalid coupon to reduce the price. Below we present a variant of this example where an engineer has four logs, some of which from a version of the shopping cart with the bug and some from a version without it. We present the example semi-formally, for illustration purposes. Formal definitions appear later in the paper.

Fig. 1 shows excerpts from four logs of the shopping cart system, from left to right, \( L_1 \) to \( L_4 \), each containing several traces. Although this is only an excerpt of traces from each of the four logs, it is already difficult to identify any difference. Real-world logs are much longer and more complex. How can the engineer find the bug that is hidden in some of them?

The first algorithm we present, 2KDiff, allows the engineer to compare two logs. Assume that the engineer is interested in comparing \( L_1 \) and \( L_3 \). 2KDiff visualizes the differences between the logs by highlighting sequences of length \( k \) or less that appear only in one of the two logs. The parameter \( k \) is set by the engineer; the higher \( k \) the more differences are expected to appear. For example, if the engineer selects \( k=2 \), 2KDiff finds that the sequence \((\text{invalid-coupon}, \text{reduce-price})\) appears only in \( L_3 \). 2KDiff highlights this \( k \)-sequence by superimposing a concrete trace from \( L_3 \) that includes this \( k \)-sequence over the \( k \)-Tails model of \( L_3 \), as illustrated in Fig. 2, produced by our prototype implementation of 2KDiff, where the trace is highlighted in red and the \( k \)-sequence is emboldened. The model shows that the bug appears in \( L_3 \) but not in \( L_1 \).
Thus the model that nKDiff presents reveals and highlights the bug.

Figure 2: The output of 2KDiff when comparing logs \( L_1 \) and \( L_3 \). A trace visualizing the differences between the models is superimposed over the K-FSM model of \( \overline{L}_3 \). Specifically, the third trace from \( L_3 \), \((INITIAL, check-out, invalid-coupon, reduce-price, check-out, get-credit-card, TERMINAL)\), is highlighted, to reflect the k-sequence \((invalid-coupon, reduce-price)\), which does not appear in any trace in \( L_1 \). This k-sequence is emboldened to emphasize the difference.

Figure 3: The output of nKDiff when comparing the four logs \( L_1 \) to \( L_4 \). Note the red transition labeled \([2, 3]\), signifying that the transition invalid-coupon after which reduce-price occurs, appears in \( L_2 \) and \( L_3 \), but not in the other two logs.

The second algorithm we present, nKDiff, compares many logs at once. Given all four logs, \( L_1 \) to \( L_4 \), as input, nKDiff outputs a single model that highlights the difference in behaviors between them. Specifically, Fig. 3 shows the output of our prototype implementation of nKDiff (with \( k=1 \)), a finite-state machine extended with colored and labeled transitions. Black transitions represent behaviors that are common to all logs. Red transitions represent behaviors that occur in only some of the logs, whose numbers appear as a label. In our example, the red transition labeled invalid-coupon, after which reduce-price has occurred, appears only in \( L_2 \) and \( L_3 \). Thus the model that nKDiff presents reveals and highlights the bug.

Note that the algorithms are complementary. 2KDiff highlights concrete traces of discovered differences between two logs. nKDiff identifies differences between many logs at once, but highlights no concrete traces. One is not a generalization of the other.

3 PRELIMINARIES

Basic Definitions. A trace over an alphabet \( \Sigma \) is a finite word \( tr = (e_1, e_2, \ldots, e_m) \) where \( e_1, \ldots, e_m \in \Sigma \). For \( j \geq 1 \) we use \( tr(j) \) to denote the \( j \)th element in \( tr \). We use \( |tr| \) to denote the length of \( tr \). For a positive integer \( k \), a \( k \)-sequence is a consecutive sequence of \( k \) or less events, denoted by \( k_{seq} \). \( k_{seq} \) is the set of all \( k \)-sequences over \( \Sigma \). A log \( L \) over an alphabet \( \Sigma \) is a set of traces \( L = \{tr_1, \ldots, tr_n\} \).

Definition 3.1 (Finite-State Machine (FSM)). A finite-state machine (FSM) is a structure \( M = (Q, Q_i, \Sigma, \delta) \) where: \( Q \) is a set of states; \( Q_i \subseteq Q \) is a set of initial states; \( Q_a \subseteq Q \) is a set of accepting states; \( \Sigma \) is an alphabet; and \( \delta : Q \times \Sigma \rightarrow \mathcal{P}(Q) \) is a transition relation, where \( \mathcal{P}(Q) \) is the power set of the set of states \( Q \).

Let \( M \) be an FSM over an alphabet \( \Sigma \). We use \( L(M) \subseteq \Sigma^* \) to denote the set of all words accepted by \( M \).

k-Tails. k-Tails, first introduced in [9], is a classic model inference algorithm. Over the last two decades, k-Tails has been presented in several variants and implemented in many works, e.g., [8, 13, 24, 25, 28]. We use a definition inspired by [7].

k-Tails takes a log and a parameter \( k \) as input. It starts by representing the log as an FSM \( M_{lin} \) composed of linear sub-FSMs, one per trace, which are then joined with a single initial state \( q_{init} \) transitioning to the start of each trace via a unique \( \alpha \) label, and a single terminal state \( q_{acc} \) to which all traces transition to at the end via a unique \( \omega \) label. Notice that the language of \( M_{lin} \) equals the set of traces in the log, given that each trace is encapsulated by \( \alpha \) and \( \omega \) events. We refer to this version of the log as the encapsulated version, denoted by \( L_{enc} \). k-Tails iteratively merges states in the \( M_{lin} \) FSM: Two states are merged iff they are \( k \)-equivalent, i.e., if their future of length \( k \) or less, is identical. When no two remaining states are \( k \)-equivalent, the algorithm terminates and outputs the resulting FSM, called a k-FSM.

More formally, we define a function \( future : Q_{M_{lin}} \rightarrow \mathcal{P}(\Sigma_{\leq k}) \), mapping states in \( M_{lin} \) to \( k \)-sequences. The \( k \)-equivalence relation induces a partition of the states of the initial FSM \( M_{lin} \) into equivalence classes \( E = \{e_1, e_2, \ldots, e_m\} \), where each of the equivalence classes in \( E \) is uniquely defined by its future sequences of length \( k \) or less. Two states \( s_1, s_2 \in e_i \) iff \( future(s_1) \approx future(s_2) \). When lifted from \( Q_{M_{lin}} \) to \( E \), the function \( future \) becomes the injective function \( id : E \rightarrow \mathcal{P}(\Sigma_{\leq k}) \). For all \( s \in e_i \), \( future(s) = id(e_i) \). Formally:

Definition 3.2 (k-FSM). k-FSM, the FSM computed by k-Tails for a log \( L \) and a positive integer \( k \), is an FSM \( M_k = (Q, Q_i, \Sigma, \delta) \) where: \( Q = E \), where \( E \) is the set of equivalence classes defined above; \( \Sigma \) is the alphabet of the log \( L \); \( \forall e \in E, a \in \Sigma : \delta(e, a) = \bigcup \{(e') \mid q, s' \in M_{lin} s.t. s' \in \delta_{M_{lin}}(q, a) \wedge s \in e \wedge e' \in \Sigma_{\leq k}\} ; \) \( Q_i = \{q_{init}\} \) is an artificial initial state; and \( Q_a = \{q_{acc}\} \) is an artificial terminal state.

When clear from the context, we write FSM instead of k-FSM.

For a given k-FSM \( M_k \), generated by running k-Tails on log \( L \), we use \( L(M_k) \) to denote the set of all words accepted by \( M_k \).

Among other properties, the correctness of the k-Tails algorithm implies that \( M_k \) may over approximate the set of traces in \( L \) but may not under approximate it, i.e., \( L \subseteq L(M_k)\). Consequently, every k-sequence included in any trace in \( L \) is part of at least one accepting word of \( M_k \). Additional useful properties of the k-FSM are that all its states are reachable from the initial state \( q_{init} \), and that the accepting state \( q_{acc} \) is reachable from all states.

4 USING MODELS FOR LOG DIFFERENCING

We present the main contributions of our work, 2KDiff, for differencing of two logs, and nKDiff, for differencing of many logs. We give formal definitions and examples, but omit the proofs, which can be found in [1].
Algorithm 1 2KDiff Algorithm

1: procedure 2KDiff(L₁, L₂, k)
2: k-sequ₁₁ ← find_kst(L₁, k); k-sequ₁₂ ← find_kst(L₂, k)
3: k-sequ₂₁ \ L₁ = k-sequ₁₁ \ k-sequ₁₂
4: GenModel(L₁, k-sequ₁₁, L₂)
5: procedure GenModel(L, k-sequ)
6: traces2ks ← SelectTraces(L, k-sequ)
7: M ← k-Tails(L, k)
8: highlight(M, traces2ks) \ displays the model marking the
9: procedure SelectTraces(L, k-sequ) \ returns a mapping of
10: traces2ks ← map()
11: while k-sequ.empty() do
12: tr, seqs ← find_top_covering_trace(k-sequ, L) \ returns
13: a trace in L (tr) covering the max. num. of k-sequences (seqs)
14: k-sequ.remove(seqs)
15: traces2ks[tr] ← seqs \ map the trace to the k-sequences
16: that it covers
17: return traces2ks

4.1 2KDiff: Differencing Two Logs

Given a positive integer k, 2KDiff compares two logs by focusing on k-differences, i.e., k-sequences that appear in one log but not the other, and presenting them in the context they appear in.

The 2KDiff Algorithm. First, 2KDiff computes the sets of k-sequences included in each of the logs, and compares the two sets to find the k-sequences that are unique to each log, i.e., the set of k-differences (if there are any, Alg. 1, lines 2-3). Second, in order to present the k-differences in their context, over concrete traces, for each of the two logs the algorithm looks for a (locally) minimal set of traces such that every k-sequence is included in at least one trace (Alg. 1, lines 9-15). The set of traces is computed in a greedy, iterative manner: in each step, the algorithm goes over the traces of the logs and selects the trace with the highest coverage of k-differences that have not yet been covered. The iteration terminates when all the k-differences are covered. Finally, 2KDiff computes the k-FSM for each log. It replays the selected traces from the previous step over these k-FSMs and highlights transitions visited during the replay in red (Alg. 1, lines 4-8). Transitions that belong to a k-difference are emboldened. When there are multiple traces, the engineer can iterate over them, one trace at a time.

Example 4.1. Consider running 2KDiff on L₁ and L₃ from Sect. 2, with k = 2. First, 2KDiff searches for the k-differences between the logs. It finds that while all k-sequences in L₁ appear in L₃, L₃ contains a single k-sequence that does not appear in L₁; k_seq=(invalid-coupon, reduce-price). Next, 2KDiff searches for a trace containing k_seq and finds the third trace in L₃: tr=(a, check-out, invalid-coupon, reduce-price, check-out, get-credit-card, o). Finally, 2KDiff computes the K-FSM for L₃, and highlights the trace tr over it, while emboldening the transitions in k_seq, as we show in Fig. 2.

It is important to note that 2KDiff is sound and complete modulo the k-sequences abstraction. Specifically, any k-sequence that appears in one log and not the other is included in at least one highlighted trace on the k-FSM of the respective log, and any such highlighted trace contains at least one such k-sequence. Roughly, these strong notions of soundness and completeness are guaranteed thanks to properties of k-FSM built by the k-Tails algorithm.

Theorem 4.2 (2KDiff Soundness and Completeness). Let k be a positive integer and let L₁, L₂ be two logs compared using 2KDiff with k-FSM₁ and k-FSM₂ their corresponding k-FSM models. Then, any trace highlighted by 2KDiff over k-FSM₁ is a trace from L₁ that includes at least one k-sequence missing from L₂; and every k-sequence that appears in L₁ and does not appear in L₂ is highlighted by at least one accepted trace in k-FSM₁. The same holds for k-FSM₂.

In search for a small set of traces that covers the k-differences, as described above, we chose to implement a greedy algorithm, which ensures that we find a locally minimal covering set: removing any trace from this set will reduce coverage. Still, there may exist a smaller covering set of traces. As finding a globally minimal set may require the enumeration of all possible subsets of traces from the log, we chose a greedy algorithm to ensure reasonable performance.

Time and Space Complexity. To construct the k-Tails model, 2KDiff uses the k-Tails variant from [6], which yields quadratic time complexity with respect to the number of events in the log. Searching for k-differences and highlighting traces over the resulting FSM is linear in the number of events in the logs. Hence, k-Tails model construction dominates the time complexity of 2KDiff. Its space complexity is linear in the number of events in the log.

4.2 nKDiff: Differencing Many Logs

2KDiff is limited to comparing two logs. We now present nKDiff, a sound and complete extension of k-Tails, from one to many logs. Roughly, given a set of n logs, {L₁, ..., Lₙ}, and a positive integer k, our goal is to compute a single model, an FSM labeled with subsets of log indexes, which will be sound and complete: its projection on any given index will result in the k-FSM we could have computed for the log with that index (soundness), and any behavior that appears in at least one of the logs will be included in it (completeness).

This labeled FSM is inspired by a similar model named featured transition system (FTS), which has been presented for the purpose of model-checking of software product lines [11] (see related work in Sect. 6).

Labeled FSM (LFSM) and k-DiffLFSM. To formalize the above, we first extend the basic definition of FSM from Def. 3.1 to a Labeled FSM (LFSM). The extension is made by labeling each of the FSM transitions with a subset of log indexes. Formally:

Definition 4.3 (Labeled Finite-State Machine (LFSM)). A labeled FSM is a structure M = (Q, Qₛ, Σ, I, δ, label) where: Q, Qₛ, Σ, and δ, are defined as in an FSM; I is a set of indexes (for us, log indexes); and label : Σ × Qₛ × Q → P(I) is a labeling function, which maps every transition in δ to a subset of indexes from I.

A trace tr is accepted by an LFSM M iff there exists an index ind ∈ I s.t. tr reaches an accepting state on a path whose all transition labels include ind. More formally, a path = (s₁, ..., sₘ) is accepting for a trace tr in an LFSM M iff ∃ind ∈ I s.t. s₁ ∈ Qₛ ∧ sᵢ ∈ Qₛ ∧ Vi, j s.t. j = i + 1 ∧ 1 ≤ i < |tr|, sᵢ ∈ δ(sᵢ, tr(i)) ∧ ind ∈ label(sᵢ, tr(i), sᵢ). As in an FSM, the language of the LFSM is the set of all traces it accepts.
An LFSM induces a projection operation \( \text{proj} : \text{LFSM} \times I \rightarrow \text{FSM} \): Given an index \( i \in I \), \( \text{proj} \) removes from the LFSM all transitions whose set of labels does not include \( i \), removes all states that become unreachable from the initial state, and then removes all labels from the remaining transitions. The result of \( \text{proj} \) is an FSM.

**Example 4.4.** The model presented in Fig. 3 represents an LFSM over the set of logs \( \{L_1, L_2, L_3, L_4\} \). The transition \( \text{invalid-coupon} \) from state 2 to state 3 is labeled with a set of log indexes, in this case, the set \( \{2, 3\} \) (to avoid clutter in Fig. 3, we do not show the label for transitions labeled with all log indexes). When applying \( \text{proj} \) to this LFSM, with index 1 or 4, the result is an FSM that does not include the transition \( \text{invalid-coupon} \) from state 2 to state 3.

**Theorem 4.5.** The language of an LFSM \( M \) is equal to the union of the languages of all projections of \( M \) to indexes from \( I \). Formally: \( L(M) = \bigcup_{i \in I} L(\text{proj}(M, i)) \).

We now extend the definition of k-FSM from Def. 3.2 to a k-DiffLFSM. Roughly, the k-DiffLFSM is a labeled k-FSM, which accepts all traces from all logs (inclusion) and whose projection to any label \( j \) results in the k-FSM generated by running k-Tails only on \( L_j \) (projection). Inclusion and projection (soundness and completeness) are important. Inclusion is important, as it guarantees that just like the k-FSM for each log, the k-DiffLFSM accepts all traces from all logs (no under approximation). Projection is important, as it guarantees that the over approximation in the k-DiffLFSM is exactly like that of the k-FSM for each log, not worse.

More formally, let \( L = \bigcup \{L_i \mid 1 \leq i \leq n\} \). Let \( L_i \) be the set of traces, so it is a valid log. Recall the k-Tails algorithm from Sect. 3. Let \( M_{lin}^j \) denote the FSM of linear sub-FSMs of \( L_j \). Let future\(^j \) be the mapping of states to their future(s) of length \( k \) or less in \( M_{lin}^j \).

**Definition 4.6.** (k-DiffLFSM). For set of logs \( \{L_1, \ldots, L_n\} \) and a positive integer \( k \), a k-DiffLFSM \( M_{lin}^{\ldots} \) is an LFSM \((Q, Q_i, \Sigma, I, \delta, \text{label})\) where: \( Q = E \), the set of equivalence classes of states from the k-FSM \( M_{lin}^{\ldots} \); \( \Sigma \) is the set of indexes \( \{1 \ldots n\} \); \( \Sigma \) is the union of the alphabets of the logs \( L_1 \) to \( L_n \); \( \forall e, e' \in E, a \in \Sigma: \text{label}(e, a, e') = \{j \mid 3s \in e, e' \in e' \in e' \in e' \in E, a \in \Sigma \} \) iff \( \text{label}(e, a, e') \neq \emptyset \). \( \Sigma \) is an artificial initial state; and \( Q_\Sigma = \{q_{acc}\} \) is an artificial terminal state.

We now formally define k-DiffLFSM’s soundness (inclusion) and completeness (projection), and illustrate them with our example.

**Theorem 4.7.** (k-DiffLFSM Soundness and Completeness). Let \( M_{lin}^{\ldots} \) be the k-DiffLFSM for a set of logs \( \{L_1, \ldots, L_n\} \) and a positive integer \( k \). Then, for all \( 1 \leq i \leq n \), \( L_i \subseteq L(M_{lin}^{\ldots}) \); and for all \( 1 \leq i \leq n \), \( \text{proj}(M_{lin}^{\ldots}, i) \) is identical to the k-FSM \( M_{lin}^{\ldots} \), generated by running k-Tails only on \( L_i \). In particular, for all \( 1 \leq i \leq n \), \( L(\text{proj}(M_{lin}^{\ldots}, i)) = L(M_{lin}^{\ldots}) \).

**Example 4.8.** Consider the four logs shown in Fig. 1, and their corresponding k-DiffLFSM model shown in Fig. 3, resulting by executing nDiff on these logs. One can check that inclusion (soundness) holds, as every trace in any of the four input logs is part of \( L(M_{lin}^{\ldots}) \). Projection (completeness) holds too, since, e.g., for \( i \neq 2, 3 \), removing the only transition \( t \) whose label is \( \{2, 3\} \), will result in the exact k-FSM generated by running k-Tails on \( L_1 \) or \( L_1 \) alone.

The nKDiff Algorithm. nKDiff takes as input a set of logs \( L = \{L_1, \ldots, L_n\} \) and a positive integer \( k \); it outputs a k-DiffLFSM \( M_{lin}^{\ldots} \). nKDiff computes the labels of the logs and the logs’ labels (Alg. 2, lines 2-3). Then, instead of an unlabeled initial FSM \( M_{lin}^{\ldots} \), it builds an initial LFSM \( M_{lin}^{\ldots} \) where each trace’s linear sub-FSM is labeled with the single index of the log from which it came from (Alg. 2, line 4).

Second, it merges all states in \( M_{lin}^{\ldots} \) into a set of equivalence classes \( E \) based on the states’ futures of length \( k \) or less (Alg. 2, line 5). \( E \) is defined as the set of states of the output k-DiffLFSM \( M_{lin}^{\ldots} \). Further, the equivalence classes holding the dummy initial and terminal states are defined accordingly (Alg. 2, lines 6-8).

Third, to construct the transition function \( \delta \) of \( M_{lin}^{\ldots} \) and the transitions’ labeling function \( \text{label} \), for each ordered pair of states \( e, e' \in E \). The algorithm checks if the future of \( e \) is succeeded by \( e' \), and if so, finds the next event \( a \). Then, it computes the maximal set of indexes \( i \) s.t. for each \( j \in s_i \), \( e \prime \in e \prime \in e \prime \in e \prime \in E, a \in \Sigma: \text{label}(e, a, e') \neq \emptyset \). If \( s_i \neq \emptyset \), nKDiff adds the transition and label to \( M_{lin}^{\ldots} \) (Alg. 2, lines 10-14).

**Time and Space Complexity.** nKDiff uses the k-Tails variant of [6] to construct a model from all logs. Its time complexity is dominated by states merging phase of k-Tails (Alg. 2, line 5) and is quadratic in the number of events in all logs. The additional steps in nKDiff of denoting transitions with labels and computing the label function (Alg. 2, line 4, 12-14), require a linear time in the number of events in all logs. Space complexity is linear in the number of events in all logs.

**4.3 Implementation and Validation**

**Implementation.** We have implemented 2KDiff and nKDiff by extending the k-Tails implementation used in [8]. The implementation
includes all steps, from parsing the logs, to computing the model, to visualizing it. We made the implementation publicly available as a prototype web application that allows review and experiments. We encourage the interested reader to check it out, see [1].

**Validation.** To validate 2KDiff, we implemented unit tests covering the steps of the algorithm, k-sequences extraction (from log), k-differences coverage (by traces from the logs), and trace highlighting over the generated model. Further, we implemented an integration test: run the algorithm over pairs of manually constructed example logs, and manually compare the output with the expected results.

To validate nKDiff, we have created and executed automated validation. The validation code runs k-Tails on each log in the input set and runs nKDiff on the set of logs. It then checks that the output models satisfy the inclusion and projection (soundness and completeness) requirements by comparing the generated models. We repeated the automated validation many times with many different logs generated from models.

The above procedures provide evidence that our implementations are correct.

### 5 Evaluation

We present an evaluation in three parts. The first evaluates the performance of 2KDiff and nKDiff. The second is a controlled user study to examine the potential use of 2KDiff and nKDiff by engineers. The third is a case study in malware analysis.

#### 5.1 Performance Evaluation

We conducted a preliminary evaluation of the performance of 2KDiff and nKDiff, guided by the following research questions:

- **RQA1** How is the performance of 2KDiff and nKDiff affected by the number of k-differences between the compared logs?
- **RQA2** How is nKDiff performance affected by the number of logs?

**5.1.1 Models Used.** We used 15 finite-state machine models in our evaluation, all taken from Lo et al. [24] and from Pradel et al. [30]. The models vary in size and complexity, i.e., the alphabet size ranges from 7 to 42, the number of states ranges from 6 to 24, and the number of transitions ranges from 15 to 209. The complete list of models and their statistics are available in supporting materials [1].

**5.1.2 Experiment Design and Setup.** We generated logs from the 15 models described above, using a publicly available trace generator [24], configured to produce state coverage and yielding logs of roughly thousand traces each.

For 2KDiff and nKDiff, in all experiments we used $k = 2$, a value of $k$ that is commonly used in the literature on k-Tails.

To introduce k-differences into the logs, we used the following log mutation procedure: clone a randomly selected trace and flip a random pair of consecutive events in it; if the modified trace consists of a k-sequence missing from the log, add it to the mutated log; Otherwise, repeat the procedure.

In measuring computation times we included all steps, from parsing the logs, to computing the models, to exporting to DOT format for visualization. We executed all experiments on an ordinary laptop computer, Intel i5 CPU 2.4GHz, 8GB RAM with Windows 8 64-bit OS, Java 1.8.0_45 64-bit. We executed all runs 10 times, to average out measurement noise from the Java execution.

#### 5.1.3 Experiment I: varying mutation type. We aim to investigate how the performance of 2KDiff and nKDiff depends on the number of k-differences between the logs. First, we selected a model and generated a log from it. Second, we created a mutated version of the log by following the mutation procedure described above. In our experiments, we consider three types of log mutation policies: no mutation (N), one mutation (O), and multiple mutations (M). No mutation means we compare two identical logs. One mutation means that the mutated log includes a new trace, with at least one new k-sequence. Multiple mutations (M) means that we repeat the process of mutating each log 10 times, effectively adding 10 new traces to the mutated log. Lastly, we run both methods, 2KDiff and nKDiff, over the original log and the mutated log, and measured their running times. We repeated each combination of model and mutation policy 10 times.

#### 5.1.4 Experiment II: varying the number of logs. We aim to check the effect of the number of logs on the performance of nKDiff. For each model and for each mutation policy, we generated a varying number of logs: 2, 4, 6, 8. For experiments with the (N) mutation policy, all logs were kept identical; for experiments with the (O) mutation policy, $n-1$ logs were kept identical, and one log contained a single mutation; and, for experiments with the (M) mutation policy, for each $i \leq i \leq n$, the $i$-rh log included a single additional mutation over the $i$-1 log. We repeated each combination of model, number of logs, and mutation policy 10 times.

#### 5.1.5 Results.** We run 2KDiff and nKDiff on the mutated logs generated from the 15 models, using each of the three mutation policies. For each model, we measured the average number of traces, average trace length (in the generated logs), and the average running times of 2KDiff and nKDiff, per mutation policy. The results show acceptable average running times for logs of realistic sizes originating from different models with an average running time below 10 seconds for 11 out of the 15 models, and where the longest average running time did not exceed 200 seconds for both methods.

Furthermore, while running times of both methods vary much across different models, the mutation policy seems to have no significant effect. nKDiff requires twice the time of 2KDiff, a phenomena which is consistent across all models. This is not surprising as nKDiff constructed a model from both logs, while 2KDiff only constructed a model from the mutated version of the logs, due to the nature of the mutation, which makes one of the logs contain all k-sequences of the other. As a result, 2KDiff only constructed the model for the log containing the additional k-sequence. For identical logs (i.e., the N mutation policy), 2KDiff constructed a single model without superimposing any of the traces.

Detailed performance results are available in [1].

To answer [RQA1], we have evidence that 2KDiff and nKDiff are applicable to systems of different size and complexity and logs of varying similarities. Both methods generate models from large logs in acceptable times.

The results of experiment II, considering average running times of four selected models, with different mutation policies, when growing the number of logs from 2 to 8, reveal that in all mutation
We choose to conduct a controlled study to focus on evaluating we conducted a controlled user study to quantitatively measure the
trace in this version and in all the following versions. We then randomly chose one of the versions (apart from
the first version) and replaced the original trace with the mutated
in the log. We then randomly chose a trace from the first log and
produced by a trace generator for a model. We copied the log five times
system. Then, we randomly chose a trace from the first log and
Five versions of the system. The participant is tasked to identify
a behavioral difference and when it was first introduced into the
new pair of consecutive events does not appear in any of the traces
of k-Tails. To guarantee that the flip added a new behavior, we checked that the
mutated it by flipping two consecutive events, i.e., a 2-sequence. To
system. By answering a set of questions.

5.1.6 Threats to Validity. First, the selection of models in our eval-
ation may not represent typical systems. To mitigate this, we used
15 publicly available models with non-trivial size and complexity,
taken from two previous works (see Sect. 5.1.1). Yet, we do not
know to what extent these are representative of real-world sys-
tems and do the mutations that we performed are representative of
real-world changes.

Second, to generate logs from the 15 publicly available models
and their mutations we used a publicly available trace genera-
tor [24], as described above. It is possible that one may get different
results if a different trace generator or a different coverage criterion
is used.

5.2 Controlled User Study
We conducted a controlled user study to quantitatively measure the
benefit that 2KDiff and nKDiff can provide to their potential users.
We choose to conduct a controlled study to focus on evaluating
pertinent features of the algorithms.

The research questions guiding our user study are:

**RQB1** Can using 2KDiff and nKDiff help participants more accu-
ately identify behavioral differences between different versions
of the same system?

**RQB2** Do 2KDiff and nKDiff shorten the time required for par-
ticipant in identifying if and when a behavioral difference was
introduced into a system?

5.2.1 Experiment Setup. To answer the research questions, we
capture a scenario where a behavioral difference is introduced into
a system. A participant is given access to logs of different runs of
five versions of the system. The participant is tasked to identify
a behavioral difference and when it was first introduced into the
system, by answering a set of questions.

To capture this scenario, we generated a log with 20 traces pro-
duced by a trace generator for a model. We copied the log five times
and numbered the copies to represent consecutive versions of the
system. Then, we randomly chose a trace from the first log and
mutated it by flipping two consecutive events, i.e., a 2-sequence. To
guarantee that the flip added a new behavior, we checked that the
new pair of consecutive events does not appear in any of the traces
in the log. We then randomly chose one of the versions (apart from
the first version) and replaced the original trace with the mutated
trace in this version and in all the following versions.

Independent and Dependent Variables. The experiment’s pur-
pose is to examine whether 2KDiff and nKDiff provide partici-
pants with support in finding log differences better than some alternatives
(baselines), while considering a number of different logs and usage
scenarios. Thus, our experiment has three independent variables,
the tool used to find log differences, the log set, and the usage sce-
nario, and two dependent variables, correctness of the task solution
(i.e., answers given by participants) and completion time.

We consider three tools, i.e., 2KDiff & nKDiff, a popular web-
based text differencing tool [2], and k-Tails; six sets of logs, i.e.,
Columba, cruiseControl.net, ctsas.net, cvs.net, java.util.StringTokenizer,
and roomcontroller.net, generated as above from models found in
existing literature [24, 30]; and two usage scenarios, i.e., Regression
Test and User Interaction. The Regression Test scenario simulates
a case when an engineer runs a test suite on multiple versions of
a software system, while the User Interaction scenario simulates
a case when a user tries various features of multiple versions of
a system. To capture the Regression Test scenario, we randomly
applied a mutation according to the procedure described above
while maintaining similar trace order between different logs. To
capture the User Interaction scenario, we shuffled the traces mim-
icking different interactions with the application. In both scenarios,
a single random mutation in the form of a new 2-sequence was the
only behavioral difference between the logs.

Participants and Task Assignments. We invited 60 graduate
students with background in software engineering from two uni-
versities. We divided the 60 participants into six groups of 10 partici-
pants each. One factor that could have an impact on the participants’
performance is experience level. We collect participants’ personal
information (e.g., the year they start their post-graduate program,
their prior experience in industry, etc.) and use it to categorize
the 60 participants into junior and senior participants. The ratio
of junior and senior participants for each group was kept approxi-
ately 3:2. Every participant is required to perform six tasks by
analyzing six log sets. He/she needs to use a log differencing tool
twice, one for the User Interaction scenario and another for the Unit
Test scenario. The participants in all groups were presented with
the log sets in a similar order. To avoid biases, we designed the
experiment such that each log was analyzed by each of the tools in
each of the usage scenarios, covering all different orders.

Detailed Procedure. At the beginning of the study, participants
are required to read a tutorial and watch a video explaining the
three log differencing tools and how they can be used to complete
the tasks. Participants typically spend 20 to 30 minutes doing this.
Then, they attempted each of the six tasks one by one. To complete
each task, participants are required to analyze a log set using a
specified tool and eventually answer a several questions through a
web interface. The following are the four questions that we asked
participants for each task: (1) Is there a log that contains any 2-
sequence that does not appear in its preceding log? (2) What is the id
of the earliest log that introduces a new 2-sequence? (3) What is the
2-sequence that appears in the new version but not in the old version?
and (4) What is the trace that shows the 2-sequence difference?

Note that if a participant answers ‘No’ to the first question, they
will not be asked the subsequent questions. Our web interface
recorded participants’ answers and the amount of time they used to complete each task.

5.2.2 Results. We report experiment results by answering the research questions mentioned earlier as follows:

RQB1: Correctness. After all participants completed the experiments, we evaluated the correctness of the participant answers. If a participant chose “No” option for the first question of a task, the other three questions are labeled as incorrect. Figure 4 shows the number of participants who gave correct answers for questions 2, 3, and 4 for the different tasks. Note that we don’t show the count for the first question since a “Yes” answer for the first question does not mean that the participant found the difference among the logs. From the figure, we observe the following:

- Most participants who use 2KDiff & nKDiff answered the questions correctly. Only three did not answer the questions correctly; these happen when they use 2KDiff & nKDiff for the first time to complete a task. We talked with them and found that they were not very familiar with the tool the first time they used it.
- For the text differencing tool, the correctness for tasks in User Interaction scenario is much lower than that for tasks in Unit Test scenario. Note that for the User Interaction scenario, traces in logs are shuffled. For such logs, the text differencing tool often returns a large number of syntactic differences, which make it difficult for participants to identify the 2-sequence difference. For the Unit Test scenario, the number of returned syntactic difference is much lower. Hence, the correctness of participants who use the text differencing tool in Unit Test scenario is close to that of participants who use 2KDiff & nKDiff.
- For k-Tails, the difference in correctness between the two scenarios is minor. However, we find that the complexity of the log set impacts correctness. For example, the model used to generate cruiseControl log set is much more complex than that used to generate Columba log set. Comparing the results for these two log sets, we find that using k-Tails, participants produced substantially fewer correct answers for cruiseControl than for Columba. We also find that some participants who performed tasks with Columba using k-Tails answered the second question correctly but the next two questions incorrectly. This might be because participants found that the two models generated by k-Tails are different but they could not identify which transitions are the new 2-sequences by comparing two k-Tails models manually.

We further compute the average correctness for the different tools when used to complete tasks in each of the two scenarios (see Table 1). The average correctness for tasks completed using 2KDiff & nKDiff is very high – more than 97% for each scenario. On the other hand, the average correctness for tasks in User Interaction scenario completed using the text differencing tool is the lowest – only 17.2%. To measure whether the differences on correctness between 2KDiff & nKDiff and baselines were statistically significant for the two scenarios, we apply Wilcoxon signed-rank test with Bonferroni Correction. The corrected p-values are all smaller than 0.01, which indicates that the difference is statistically significant at a confidence level of 99%. We also calculated Cliff’s delta\(^1\), which is a non-parametric effect size measure, to show the effect sizes of the correctness difference between 2KDiff & nKDiff and each of the baselines. The Cliff’s deltas are all large, demonstrating the effectiveness of our proposed tool in helping participants produce correct results for the tasks.

2KDiff & nKDiff can help participants accurately identify behavioral differences among different logs. The differences in average correctness between tasks completed using our tool and those using a baseline are statistically significant with large effect sizes.

RQB2: Completion time. Table 2 shows the average participant completion time for each task using our tool and the baselines. From the table, we can note the following:

- The average completion time for tasks performed using 2KDiff & nKDiff is lower than that of the two baselines, except for two tasks: one uses the cvs log set considering the Unit Test scenario, and the other uses the StringTokenizer log set considering the Unit Test scenario.
- For the first of the two tasks mentioned above, the average completion time of participants using k-Tails is slightly lower but close to that of participants using 2KDiff & nKDiff (173.3 vs. 177.1 seconds). For the second, the average completion time of participants using the text differencing tool is slightly lower but close to that of participants using 2KDiff & nKDiff (230.0 vs. 244.7 seconds). Note that a participant with the baseline tools might quit the tasks in a short time if they believed that it was very hard

---

\(^1\)Cliff defines a delta of less than 0.147, between 0.147 to 0.33, between 0.33 and 0.474, and above 0.474 as negligible, small, medium, and large effect size, respectively [12].
for them to find the difference. This could make the average time for these tasks performed using the baseline tools lower. This hypothesis is supported by the fact that the average accuracy in completing these two tasks using the two baselines are lower than when 2KDiff & nKDiff is used (see Figure 4).

- Participants using text differencing tool spend much less time for Unit Test tasks than for User Interaction tasks. This is because it is difficult for participants to identify 2-sequence difference among the large number of syntactic differences produced by the text differencing tool when applied to User Interaction logs. This is also reflected by the high variance in the completion time, as can be viewed in the corresponding box-plot.

- The completion times of both k-Tail and 2KDiff & nKDiff are influenced by the model. To investigate this, we run linear regression, using the number of transitions as an independent variable and the completion time as the dependent variable. The coefficient, p-value, and adjusted R-squared for k-Tails and 2KDiff & nKDiff are (2.91, 0.017, 0.05) and (0.79, 0.1, 0.017) resp. This shows that k-Tails significantly depends on the model complexity while 2KDiff & nKDiff has a weaker statistical dependence with a smaller effect (i.e., coefficient).

Figure 5 shows the box-plots of completion times for different tools considering each of the two scenarios. We performed Wilcoxon signed-rank test with Bonferroni Correction and find that the differences are all statistically significant at a confidence level of 95% (see Table 3). The effect sizes of the differences on completion time for User Interaction scenario are medium and large, while the effect sizes for Unit Test scenario are both small.

### 5.2.3 Threats to Validity
Several threats may affect the validity of our findings. First, there may be errors in the tools and web interface that we provide to user study participants. We have tried to reduce possibility of error by performing a thorough check and by conducting a pilot study with a few participants whose results we have excluded from the ones reported above. Second, all our user study participants are students; it is possible that the findings would be different if professional engineers are used as participants instead. To mitigate this threat, we did not invite undergraduates but rather graduate students with substantial years of programming experiences. Many of our participants have worked in the industry prior to joining the graduate program. Additionally, a number are currently still working on industrial projects while completing their master degree. Students are used as participants in many past software engineering studies, e.g., [17, 36, 38]. Moreover, a recent work by Salman et al. highlights that there are only minor differences between students and professionals in their user study [33]. Third, results of our controlled experiment may differ from a field study. We choose controlled experiments to allow us to control study variables. This enables us to investigate the performance of our approach and the baselines when some of these variables are varied. We can also prevent unwanted variables from affecting the results. Basili has highlighted these and many other benefits of controlled experiments [5]. Many prior software engineering work have also chosen to perform controlled experiments [26, 37, 40].

### 5.3 Case Study
We conducted a case study to examine the potential of 2KDiff and nKDiff on malware analysis in practice. Due to the popularity of Android platform, a large amount of Android malware are produced by attackers. Most Android malware are generated by infecting benign apps with malicious code, which results in a different behavior from the original benign apps, e.g., accessing privacy or security data. Therefore, in this study, we want to investigate whether 2KDiff and nKDiff can identify malicious behavior by comparing the API logs of malware with those of the original benign apps.

We use the log dataset from the study of Bao et al. [4] in which they use five automated test case generation tools to generate log traces by running more than 100 pairs of malware and benign apps in order to test their performance. They instrumented the tested apps to record the API calls and the format of each record is caller → callee. The app pairs are from a real life benign piggybacked Android app dataset collected by Li et al. [20]. The malicious piggybacked apps are built by attackers by unpacking benign apps and then grafting some malicious code to them.
Our work is part of a larger project aiming to build tools that help engineers make better use of execution logs. In this context, we envision the following challenges ahead. First, our present work is limited to identifying k-differences. It may be useful to investigate additional notions of behavior differences that we can infer from the logs, e.g., temporal invariants or other, extra-functional properties [29]. Second, our current approach reports all differences, but in many cases some differences may be more important than others. A quantitative extension that takes frequencies into consideration and applies a statistical approach [10], may help engineers to rigorously distinguish between significant and insignificant differences.

ACKNOWLEDGMENT
This work is partly supported by the Len Blavatnik and the Blavatnik Family Foundation, Blavatnik Interdisciplinary Cyber Research Center at Tel Aviv University, and Singapore National Research Foundation’s National Cybersecurity Research & Development Programme (award number: NRF2016NCR-NCR001-008).

REFERENCES
REFERENCES


