Mining Task Precedence Graphs from Real-Time Embedded System Traces

Oleg Iegorov and Sebastian Fischmeister
University of Waterloo, Canada
{oiegorov, sfischme}@uwaterloo.ca

***Abstract*—Real-time embedded systems have evolved from simple, self-contained single-processor computers to distributed multiprocessor systems that are extremely hard to develop and maintain. Execution tracing has proved itself to be a useful technology to gain a detailed knowledge of runtime behavior of software systems. However, the size and complexity of execution traces generated by modern embedded systems make manual trace analysis impossible. Therefore, software developers need tools to extract high-level system models from raw trace data.**

In this paper, we address the problem of mining task precedence graphs (TPG) from embedded system traces. A TPG can be helpful in performing several crucial software development and maintenance activities: understanding legacy systems, finding runtime bugs, and detect and diagnose anomalies in running systems. We rely on the recurrent nature of real-time systems to solve the TPG mining problem.

We propose algorithms to train a TPG on a set of system traces, as well as an algorithm to detect anomalies in trace streams using a TPG. We evaluate our algorithms on industrial execution traces generated on production cars.

I. Introduction

Modern Real-Time Embedded Systems (RTES) have reached an unprecedented level of complexity. An obvious example of an industry impacted by this rise of complexity of RTES are automobile manufacturers who are rapidly becoming software-intensive companies [17]. Mere forty years ago, software made its first appearance in a car (the 1977 GM's Oldsmobile Toronado) to control the ignition system of the car's engine [6] [8]. Modern cars, on the other hand, have dozens of embedded microprocessors which need to communicate with each other in real-time to deliver advanced safety and entertainment features. Therefore, it comes as no surprise that the size of software running in cars have grown exponentially and gone from zero lines of code in 1977 to over a hundred million lines of code in modern automobiles [8]. The same trend can be observed in other industries that extensively rely on RTES, such as avionics, space [19], medical [18] and mobile phone industries. Among the outcomes of the ever-growing software complexity in modern RTES are the extreme difficulties confronting software developers in understanding legacy code, finding bugs in new versions of software, and detecting attacks on target systems.

Software execution tracing is a popular technique to get insights into runtime operation of a system. Tracing allows software developers to perform a fine-grained analysis of the execution behavior of relevant parts of the code. At the same time, the complexity of software running on modern RTES requires software developers to analyze the totality of system operation and not just the system's components in isolation. This way, software developers have to work with *system traces* which capture activities and states of the whole system. Unfortunately, the volume and complexity of system traces usually exceed a human's ability to navigate and make sense of the captured data. Therefore, it is necessary to extract higher-level models from raw trace data. This process is usually referred to as reverse engineering, model inference, or else *model mining*.

There is a wide choice of models that can be mined from system traces. Conceptually, these models can be divided into two categories: invariants and temporal properties [12]. An invariant is a property that holds at a certain point or points in a program; for example, a variable being equal to a constant ($x = a$), being a function from a library ($x = \text{fn}(y)$), and others [15]. Temporal properties define the timing constraints of system components. This type of model is especially useful in characterizing RTES since their software must operate under stringent real-time constraints.

In this paper, we are interested in mining a particular type of temporal property called *task precedence graph* from embedded system traces. A task precedence graph (TPG) is a directed acyclic graph that shows the partial order of execution of system tasks. Embedded software developers use TPG to model precedence constraints between real-time tasks. These constraints are essential to accomplishing complex control activities of the target RTES. Precedence graphs, like other parts of software documentation, are often not properly maintained during the software lifecycle. Hence, having a tool to mine task precedence graphs from system traces would be highly beneficial to embedded software developers.

Models extracted from execution traces can be useful not only to enhance understanding of the underlying system but also to *detect anomalies* in system executions. Indeed, if a running system violates a model representing normal system behavior, then this can be a sign of an internal bug, or an external attack on the system. We believe that a practical anomaly detector is not only accurate but also has the following properties: (1) facilitates diagnostics by indicating the specific part of the code responsible for the observed anomaly; (2) processes traces in real time (online) fashion, allowing the system to halt or take preventive actions as soon as an anomaly has been detected; (3) is interactive, that is, makes it possible for the user to flag a detected anomaly as valid behavior and update the model accordingly. Despite their practical usefulness, these properties are not addressed simultaneously by anomaly detection methods [7]. In this paper, we propose a TPG-based anomaly detection approach that is characterized by all the properties mentioned above. The implementation of the proposed approach is available as open-source [1].

The key contributions of this paper include:

- an algorithm to mine a task precedence graph from a set of valid RTES traces;
- a TPG-based anomaly detection method;
- two case studies on the application of the proposed mining and anomaly detection methods on a set of industrial execution traces generated on production cars.

The rest of the paper is organized as follows. Section II presents the relevant work on mining temporal properties from execution traces. Section III introduces the necessary definitions and notation. In Section IV, we state a theorem which we further use as a ground truth for our contribution. In Section V, we formulate the problem statement of this work. Section VI presents our method of mining a TPG from a set of system traces. Section VII explains how a TPG can be used to detect anomalies in trace streams. We conduct case studies of the proposed mining and anomaly detection approach in Section VIII and discuss its limitations in Section IX.

II. Related Work

Most of the existing research on mining temporal properties from execution traces focuses on the qualitative aspect of time, i.e., the order of events in traces, and ignores the quantitative aspect, i.e., the actual duration of time between events [12]. In the following, we review the work that considers only the qualitative aspect of time, since task precedence graphs describe precedence relations, that is, the order among system tasks.

A finite-state machine (FSM) is a well-studied formalism that can be used to model sequential data. Inferring FSM from symbolic sequences has been actively studied since the 1960s in the context of grammar induction [14]. Since grammar induction methods require negative examples, i.e., sequences that can not be generated by the system, they are not practical for system traces [32]. The k-tails algorithm [5], which allows to infer an FSM from positive examples only, influenced a number of works on mining FSM from execution traces, for example [10] [32] [2] [25]. The basic idea of the k-tails algorithm is to merge a pair of states if they generate the same sequences of k events, where k is a user-specified number.

There exists a large body of work on mining process models from workflow logs [37]. A workflow log is a counterpart of an execution trace in the domain of business process management; it captures a sequence of performed business activities. Process models have been mined in different forms, for example, as a Petri net [36], a directed graph [1], a workflow net [39], and others.

[1]https://bitbucket.org/oiegorov/tpg_miner

An important limitation of the majority of FSM and process mining algorithms is their implicit assumption that the underlying system is sequential. That is, if a system component B depends on a system component A, then an execution of B must directly follow an execution of A in system traces. Some works allow events to occasionally occur between the executions of A and B, but these situations are treated as noise [10]. On the other hand, the interleaving of independent system activities is an inherent property of embedded system traces. This means that if a component B appears directly after a component A in a system trace, A and B do not necessarily have a precedence relation: they may be independent and scheduled one after another by chance.

Some authors address the problem of mining FSM from execution traces of non-sequential systems. Beschastnikh et al [4] mine communicating FSM (CFSM) from execution traces of distributed systems. They do not require related components to directly follow each other in traces. However, the user must specify which system components can communicate, and what is the set of allowed communication patterns between components. Yang et al [41] propose to mine a single type of patterns from execution traces: "A is followed by B before the next A". Their tool Peracotta then merges the mined patterns into a chain. Our approach is similar to Peracotta in this regard, but it mines a more general type of patterns "B executes n times between the m^{th} and the $(m+1)^{\text{th}}$ executions of A". This allows us to capture precedence relations of recurrent activities in RTES.

III. Terminology

In this section, we present the terminology that will be used in Section V to define the problem statement of our work.

A. String terminology

An *item* e is a symbolic representation of some entity. A complete set of available items is called an *alphabet* $\mathcal{I}$.

A *string* s is an ordered list of items, denoted as $s = \langle e_1, e_2, \ldots, e_n \rangle$ where $e_i \in \mathcal{I}$ for $1 \leq i \leq n$. A string having n elements is called a n-string. The i-th element of s is $s[i]$.

A m-string $s' = \langle f_1, f_2, \ldots, f_m \rangle$ is a *substring* of a n-string $s = \langle e_1, e_2, \ldots, e_n \rangle$, denoted as $s' \sqsubset s$, if $m < n$ and $\exists i \in [1, n-m+1]$, such that $\langle f_1, f_2, \ldots, f_m \rangle = \langle e_i, e_{i+1}, \ldots, e_{i+m-1} \rangle$. If the equality holds when $i = 1$, s' is called a *prefix* of s, written as $s' \sqsubset_p s$. If the equality holds when $i = n - m + 1$, s' is called a *suffix* of s, written as $s' \sqsubset_s s$. Also, $s[i, j]$ $(1 \leq i \leq j \leq n)$ denotes a substring of s which starts at $s[i]$ and ends at $s[j]$.

Given a n-string s and its substring $s' = s[i, j]$ $(1 \leq i \leq j \leq n)$, i is said to be an *occurrence* of s' in s. We make $O(s', s)$ denote a list of occurrences of s' in s sorted in ascending order. $O_q(s', s)$ denotes the position of the q^{th} occurrence of s' in s.

We say that a string s' *covers* a string s when s is a concatenation of some number of s', possibly ending with a prefix of s'. Formally, a m-string s' *covers* a n-string s if (1) $O_1(s', s) = 1$; (2) $O_{i+1}(s', s) = O_i(s', s) + m$; (3) if

$O_N(s', s) + m < n$, then $s[O_N(s', s) + m + 1, n] \sqsubset_p s'$, where $O_N(s', s)$ is the last occurrence of s' in s. For example, a 5-string $s' = \langle 1, 0, 0, 1, 0\rangle$ covers both $s_1 = \langle 1, 0, 0, 1, 0, 1, 0, 0, 1, 0\rangle$ and $s_2 = \langle 1, 0, 0, 1, 0, 1\rangle$, but s' does not cover $s_3 = \langle 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0\rangle$.

B. Trace terminology

In real-time systems, a *task* τ is an independent thread of execution that can compete with other concurrent tasks for processor execution time [24]. If we make $\mathcal{I}_S$ denote a task set of a particular system S, then each task $\tau_i \in \mathcal{I}_S$ can be viewed as an item from an alphabet $\mathcal{I}_S$.

A *trace* $W = [\varepsilon_1, \dots \varepsilon_N]$ of a RTES S is a chronologically ordered list of events. An event $\varepsilon = (t, \tau)$ is a tuple consisting of a time stamp t and a task τ which started its execution at time t, $\tau \in \mathcal{I}_S$. We refer to the timestamp and the task of an event ε as ε.time and ε.task correspondingly. In the following, we will view traces as strings of tasks, ignoring the time stamps of trace events. This way, the k-th element of a trace W, written as $W[k]$, is the task ε_k.task which started its execution at time ε_k.time.

A *τ_j-transactionalized trace* W, written as W_{τ_j}, is a list of substrings of W obtained by splitting it at the occurrences of τ_j. For example, given a trace W from Figure 1(a) and $\tau_j = B$, the B-transactionalized W is shown in Figure 1(b). We call a string $s \in W_{\tau_j}$ a *transaction.*

D,A,B,D,B,D,C,B,D,A,C,B,D,B,C,A,D,B,C,D,B,D,B,D,A,B,D,B,D,A,B

(a) example trace W

⟨⟨D,A⟩, ⟨D⟩, ⟨D,C⟩, ⟨D,A,C⟩, ⟨D⟩, ⟨C,A,D⟩, ⟨C,D⟩, ⟨D⟩, ⟨D,A⟩, ⟨D⟩, ⟨D,A⟩⟩

(b) B-transactionalized W (W_B)

Fig. 1: Trace examples

Given a transactionalized trace W_{τ_j} and a task τ_i, $\tau_i \neq \tau_j$, an *occurrence string* of τ_i in W_{τ_j}, written as $os(\tau_i, \tau_j)$, is an ordered list of numbers of occurrences of τ_i in W_{τ_j}'s transactions. For example, given the W_B from Figure 1(b) and $\tau_i = A$, $os(A, B) = \langle 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1\rangle$. Given an occurrence string s, an *occurrence pattern* p of s is a string that covers s, and the length of p is minimal among all strings that cover s. The occurrence pattern of $os(A, B)$ from the above example is $\langle 1, 0, 0, 1, 0\rangle$. An occurrence pattern of the occurrence string of a task τ_i in a transactionalized trace W_{τ_j} is denoted as $op(\tau_i, \tau_j)$.

A *trace stream* is a trace whose events (a) arrive online, (b) have to be processed in the order of arrival, and (c) are discarded as soon as they have been processed [3]. A trace may be considered as a trace stream recorded for some period of time.

C. Task precedence graph terminology

Real-time tasks may interact according to a fixed partial order. This creates precedence relations among tasks [11]. Given a pair of tasks τ_i and τ_j, τ_i is a *predecessor* of τ_j, written as $\tau_i < \tau_j$, if τ_j starts its execution after the completion of τ_i; τ_j is a *successor* of τ_i. A task τ_i is an *immediate predecessor* of a task τ_j, written as $\tau_i \ll \tau_j$, if the output produced by τ_i is communicated as input to τ_j.

Precedence relations among tasks of a RTES S are known before execution and can be represented by a directed acyclic graph called a *task precedence graph* (TPG) of S and denoted as TPG(S). The nodes of a TPG represent tasks and the edges (e) correspond to immediate precedence relations among nodes. This way, if $\tau_i \ll \tau_j$ in a system S, then $e(\tau_i, \tau_j) \in$ TPG(S); if $\tau_i < \tau_j$, then there exists a path from τ_i to τ_j in TPG(S).

IV. A THEOREM ON PRECEDENCE RELATIONS OF REAL-TIME TASKS

In this section, we present a theorem that serves as the ground truth to our TPG mining method. In Section II, we drew the reader's attention to the fact that the interleaving of independent system activities is an inherent property of system traces. This makes the problem of mining precedence relations from system traces non-trivial. The following theorem tells us how to find pairs of real-time tasks with a precedence relation from system traces.

Theorem 1. *If a task τ_i is an immediate predecessor of a task τ_j in a real-time system S, then there exists op(τ_i,τ_j) in any execution trace of S captured during at least two hyperperiods of S.*

Proof. Assume that we are given a trace W of a real-time system S and a pair of tasks τ_i, $\tau_j \in W$ such that τ_i is an immediate predecessor of τ_j ($\tau_i \ll \tau_j$). Consider first a *simple* precedence relation, when $\tau_i \ll \tau_j$ implies that the k-th execution of τ_i must precede the k-th execution of τ_j. In this case, *op*(τ_i, τ_j) obviously exists and is equal to $\langle 1\rangle$. In case of a *complex* precedence relation, the periods of tasks, $T(\tau_i)$ and $T(\tau_j)$, are different [11]. However, during the system's hyperperiod $\mathcal{P}$ there is a specific number of occurrences of τ_i and τ_j ($\mathcal{P}/T(\tau_i)$ and $\mathcal{P}/T(\tau_j)$ correspondingly), and execution of τ_i must precede execution of τ_j whenever τ_i and τ_j are both in the ready queue since τ_j requires τ_i's output. Put differently, the number of occurrences of τ_i between any pair of consecutive invocations of τ_j is deterministic. Finally, if a trace captures system's activity during at least two hyperperiods, then op(τ_i, τ_j) is guaranteed to cover os(τ_i, τ_j). Figure 2 illustrates this when $T(\tau_1)$=50 ms and $T(\tau_2)$=20 ms. □

As a supporting example, consider Figure 3 which shows a TPG and periods of tasks (in milliseconds) of an embedded software running on a small unmanned helicopter discussed in [40]. Tasks τ_1 and τ_2 ($\tau_1 \ll \tau_2$) have the same period of

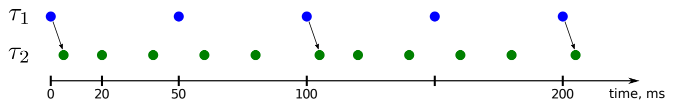

Fig. 2: An example of a complex precedence relation between a pair of real-time tasks ($\tau_1 \ll \tau_2$); *op*(τ_1, τ_2) = $\langle 1, 0, 0, 1, 0\rangle$.

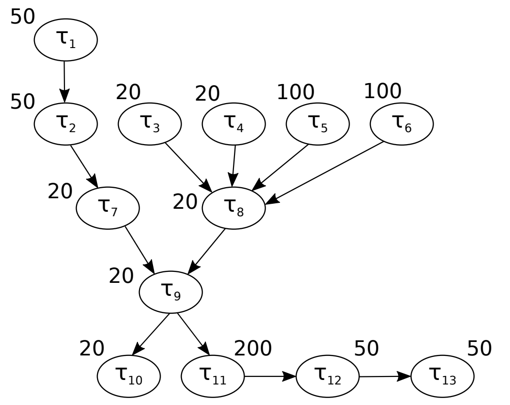

Fig. 3: The TPG of an unmanned helicopter presented in [40] annotated with periods of tasks (in milliseconds).

50 ms, so op(τ_1,τ_2) = $\langle 1\rangle$ in traces produced by this system. At the same time, τ_2 and τ_7 ($\tau_2 \ll \tau_7$) have different periods ($T(\tau_2)$ = 50 ms and $T(\tau_7)$ = 20 ms), which makes their precedence relation complex and op$(\tau_2, \tau_7) = \langle 1, 0, 0, 1, 0\rangle$ (see Figure 2). Another example of a complex precedence relation is the one between tasks τ_9 and τ_{11} ($\tau_9 \ll \tau_{11}$): $T(\tau_9)$ = 20 ms, $T(\tau_{11})$ = 200 ms; $op(\tau_9, \tau_{11})$ = $\langle 10\rangle$.

Note that Theorem 1 addresses only immediate predecessors/successors and does not require a non-immediate predecessor τ_i of a task τ_j to have an occurrence pattern in W_{τ_j}. For example, given a trace W= $\langle A, B, C, A, B, A, C, B, C\rangle$, the task A is a predecessor of the task C since $A \ll B$, $op(A, B)$=$\langle 1\rangle$, and $B \ll C$, $op(B, C)$=$\langle 1\rangle$, but $op(A, C)$ does not exist in W.

V. Problem statement

The following definitions help us to formulate the problem statement of this work.

We call a TPG G *conformant* to a set of traces $\mathbb{W}$ when G contains an edge $e(\tau_i, \tau_j)$ only if there exists $op(\tau_i, \tau_j)$ in all $W \in \mathbb{W}$, and there is no set of tasks $\mathcal{R} = \{\tau_1, \tau_2, \dots \tau_n\}$, $n \geq 1$, $\tau_i \notin \mathcal{R}$ and $\tau_j \notin \mathcal{R}$, such that occurrence patterns $op(\tau_i, \tau_1), op(\tau_1, \tau_2), \dots, op(\tau_n, \tau_j)$ exist in all $W \in \mathbb{W}$. Note that if a TPG G is conformant to a trace W, then it follows from the definition of occurrence pattern that G is conformant to any prefix of W.

An event $\varepsilon' = \langle t, \tau_j\rangle$ in a trace W is called an *anomaly* with respect to a TPG G, if there exists a task τ_i such that $e(\tau_i, \tau_j)$ is in G but there is no $op(\tau_i, \tau_j)$ in $W' = W[\varepsilon_1, \varepsilon']$. Simply put, an anomalous event in W violates a precedence relation in G.

We are now ready to formally state the two problems addressed in this work:

P1 (TPG mining). Given a set of traces $\mathbb{W}$, mine a task precedence graph G conformant to all traces in $\mathbb{W}$.

P2 (TPG-based anomaly detection). Given a task precedence graph G and an incoming trace stream W, determine if there is an anomalous event ε in W with respect to G.

VI. Task precedence graph mining

In this section, we present a method to mine a TPG from a set of system traces. The method mines a complete set of occurrence patterns from a single trace (Section VI-A), then builds a TPG conformant to this trace (Section VI-B), and then trains the mined TPG on the remaining traces (Section VI-C).

A. Mining occurrence patterns

We first address the problem of finding the occurrence pattern of a n-string s. According to the definition, the occurrence pattern of s is a prefix of s. Therefore, we proceed by checking if there exists a prefix that covers s, starting from the prefix of length 1. This way, if a prefix p is found to cover s, p is necessarily the occurrence pattern of s since all the shorter prefixes do not cover s. This approach is specified in Algorithm 1. The worst-case complexity of Algorithm 1 is $O(n^2)$, which is observed when all but the last element of s have the same value.

Algorithm 1: Find the occurrence pattern of a string

```
Data: $n$-string $s$
Result: occurrence pattern $p$ or NULL if $p$ does not exist
1  $max\_p\_length \leftarrow n/2$
2  foreach $p\_length = 1, 2, \dots, max\_p\_length$ do
3      $is\_op \leftarrow$ TRUE
4      $p \leftarrow s[1 : p\_length]$
5      $num\_included \leftarrow ceil(n/p\_length)$
6      foreach $k = 0, 1, \dots, (num\_included - 1)$ do
7          if !$is\_op$ then
8              break
9          foreach $l = 1, 2, \dots, p\_length$ do
10             if !$is\_op$ then
11                 break
12             $element\_number \leftarrow k \times p\_length + l$
13             if $element\_number > n$ then
14                 break
15             if $s[element\_number] \neq p[l]$ then
16                 $is\_op \leftarrow$ FALSE
17         end
18     end
19     if $is\_op$ then
20         return $p$
21 end
22 return NULL
```

Algorithm 2 shows how to extract a set of occurrence patterns Q from a trace W using Algorithm 1. An element of Q is a a tuple *(parent, child, op)*, where *parent* and *child* is a pair of tasks such that *parent* has an occurrence pattern *op* in the *child*-transactionalized W. Algorithm 2 has the worst-case complexity of $O(|\mathcal{T}|^2|W|^2)$, where $|\mathcal{T}|$ is the number of tasks in W and $|W|$ is the number of events in W, since an occurrence pattern is mined for each ordered pair of tasks in W. It is important to note, however, that the `for` loop on lines 5-10 can be run in parallel for all $\tau_i \in \mathcal{T}\backslash\tau_j$, which can greatly

Algorithm 2: Mine a set of occurrence patterns of a trace

```
Data: trace $W$
Result: set of occurrence patterns $Q$
1  $Q \leftarrow \emptyset$
2  $\mathcal{T} \leftarrow$ unique tasks from $W$
3  foreach $\tau_j \in \mathcal{T}$ do
4      $W_{\tau_j} \leftarrow \tau_j$-transactionalized $W$
5      foreach $\tau_i \in \mathcal{T} \backslash \tau_j$ do
6          $s \leftarrow os(\tau_i, W_{\tau_j})$
7          $p \leftarrow$ occurrence pattern of $s$ (see Algorithm 1)
8          if $p \neq \mathit{NULL}$ then
9              $Q \leftarrow Q \cup$ (parent= $\tau_i$,child= $\tau_j$, op= $p$)
10     end
11 end
12 return $Q$
```

reduce the computational time of Algorithm 2. Moreover, if the user knows that the trace W captures system activity for longer than two system's hyperperiods, they can reduce accordingly the value of max_p_length (line 1 in Algorithm 1) to speed up the mining. In other words, max_p_length can be set to $n/k - 1$, where k is the number of system's hyperperiods recorded in W.

B. Building a TPG

A task precedence graph G of an embedded system is defined by the system's task set (nodes of the graph) and pairs of tasks with an immediate predecessor/successor relation (edges of the graph). According to Theorem 1, the edges of G can be extracted from the set of occurrence patterns Q of a trace W generated by the target system. However, if we simply draw an edge from a task τ_i to a task τ_j such that (*parent* $= \tau_i$, *child* $= \tau_j$, *op*) $\in Q$, the obtained graph may not be conformant to W, because there may exist another path from τ_i to τ_j in G.

The problem of making a TPG G, obtained by drawing an edge $\tau_i \rightarrow \tau_j$ when (*parent* $= \tau_i$, *child* $= \tau_j$, *op*) $\in Q$, conformant to a trace W can be viewed as the problem of finding a transitive reduction of a directed acyclic graph (DAG). Indeed, a transitive reduction of a DAG H does not contain an edge $e(v_i, v_j) \in H$, if the vertex v_j is reachable from the vertex v_i via another path in H, that is, if there exists a set of vertices $\{v_1, v_2, \ldots, v_N\}$ such that $\{e(v_i, v_1)$, $e(v_1, v_2)$,..., $e(v_N, v_j)\} \in H$. Gries et al. [16] proposed an algorithm to construct a transitive reduction of a DAG H by firstly finding the transitive closure of H using the Warshall's algorithm [38], and then extracting the transitive reduction from the transitive closure, since the transitive closure of a DAG has the same transitive reduction as the DAG itself. Algorithm 3 shows how we can find a TPG G conformant to a trace W using the Gries' algorithm. The worst-case complexity of Algorithm 3 is the same as the complexity of the Gries' algorithm, which is $O(\mathcal{T}^3)$. This makes the total complexity of mining a task precedence graph from a system trace W equal to $O(|\mathcal{T}|^2|W|^2 + |\mathcal{T}|^3)$.

Algorithm 3: Mine a task precedence graph conformant to a trace

```
Data: trace $W$, set of occurrence patterns $Q$
Result: task precedence graph $G$ conformant to $W$
1  $V \leftarrow$ unique tasks from $W$          // set of nodes
2  $E \leftarrow \emptyset$                       // set of edges
3  foreach $q \in Q$ do
4      $E \leftarrow E \cup (q$.parent,$q$.child)
5  end
6  $G \leftarrow E \cup V$
7  $G^c \leftarrow$ transitive closure of $G$ (see [38])
8  $G \leftarrow$ transitive reduction of $G^c$ (see [16])
9  return $G$
```

C. Training a TPG

Next, we present a method to train a task precedence graph on a set of traces (see Algorithm 4). Observe that Algorithm 3 allows mining a TPG conformant to a single trace. It is likely, however, that software developers would like to run their system multiple times to capture as much normal system behavior as possible and then mine a single TPG conformant to all the execution traces.

The main idea behind Algorithm 4 is to verify for each event ε of a trace W that all occurrence patterns $q \in Q$, such that $q.\mathrm{child} = \varepsilon.\mathrm{task}$, have been respected up to $\varepsilon.\mathrm{time}$. This is done by introducing an additional pair of elements to each tuple $q \in Q$: *tokens* is the number of times the $q.\mathrm{parent}$ task has appeared in W since the previous appearance of the q.child task (line 2), and *current_os* is the current occurrence string of the $q.\mathrm{parent}$ task in $W_{q.\mathrm{child}}$ (line 3). Tokens are used to update occurrence strings by appending q.tokens to q.current_os each time the q.child task appears in W (line 8); q.current_os is reset if q.current_os $=$ q.current_op (line 14). If q.current_os is not a prefix of q.op, then G needs to be updated because the trace W no longer has the occurrence pattern q. If done naively, the update requires recomputation of the transitive closure and reduction of G using the Gries' algorithm, which has the cubic worst-case complexity with respect to the number of vertices in G (see Section VI-B). La Poutre et al. [22] proposed an approach to update a transitive reduction after an edge deletion which has the quadratic worst-case complexity with respect to the number of vertices in G. Since this method has a better complexity than the naive one, we apply it on line 12 to update the precedence graph G after the removal of a violated occurrence pattern q (line 11).

VII. TPG-BASED ANOMALY DETECTION

In this section, we explain how a task precedence graph trained on valid system traces can be used to find anomalies in trace streams. Interestingly, Algorithm 4 presented in the previous section can be easily adapted to perform anomaly detection in trace streams. Indeed, our TPG training algorithm processes input traces one event at a time in the increasing

Algorithm 4: Train a TPG on a set of traces

```
Data: set of traces $\mathbb{W}$, set of occurrence patterns $Q$, task
      precedence graph $G$
Result: $Q$ and $G$ trained on traces from $\mathbb{W}$
1  foreach $q$ in $Q$ do
2      $q$.tokens $\leftarrow 0$
3      $q$.current_os $\leftarrow \langle\rangle$            // empty string
4  end
5  foreach $W \in \mathbb{W}$ do
6      foreach $\varepsilon \in W$ in the increasing order of $\varepsilon$.time do
7          foreach $q \in Q$, such that $q$.child $= \varepsilon$.task do
8              $q$.current_os $\leftarrow \langle$ $q$.current_os, $q$.tokens $\rangle$
9              $q$.tokens $\leftarrow 0$
10             if $q$.current_os $\not\sqsubseteq_p$ $q$.op then
11                 $Q \leftarrow Q \backslash q$
12                 update $G$ (see [22])
13             if $q$.current_os $=$ $q$.op then
14                 $q$.current_os $\leftarrow \langle\rangle$
15         end
16         foreach $q' \in Q$, such that $q'$.parent $= \varepsilon$.task do
17             $q'$.tokens $\leftarrow q'$.tokens $+1$
18         end
19     end
20 end
21 return $Q$,$G$
```

order of their timestamps (line 6), which makes it applicable to trace streams.

The first necessary modification to Algorithm 4 reduces the number of occurrence patterns that need to be verified during anomaly detection process. According to the definition of an anomalous event (Section V), only a violation of an occurrence pattern between tasks connected by an edge in the TPG will result in an anomaly. Therefore, if there is no edge from $q.\text{parent}$ to $q.\text{child}$ in the TPG, where q is one of the existing occurrence patterns, then the anomaly detection algorithm can proceed to the next event in the trace stream. The following lines inserted into Algorithm 4 between lines 7 and 8 does this check:

```
if $e$($q$.parent, $q$.child) $\notin G$ then
    next
```

The second modification to Algorithm 4 makes it return the anomalous event ε and the violated occurrence pattern q as soon as an anomaly is found. Therefore, the following line must replace lines 11 and 12 :

```
return $\varepsilon, q$
```

Finally, if no anomaly is found in the trace stream, the algorithm must return `NULL` on line 21. With these modifications in place, Algorithm 4 becomes an anomaly detection algorithm on trace streams.

Besides being able to work on trace streams, our TPG-based anomaly detection algorithm has two other important features outlined in Section I: it can make the anomaly detection interactive, and it facilitates the diagnostics of the found anomalies.

The interactivity feature makes it possible to mark a detected anomaly as a valid system behavior, let the algorithm update the TPG, and proceed with anomaly detection from the next event in the trace. For this, we keep lines 11 and 12 in Algorithm 4, which are responsible for updating the TPG, but report an anomaly right after line 10. This feature can be helpful when the amount of training data is limited, so that software developers can decide themselves if a particular unseen system behavior is valid.

The ability of our anomaly detection technique to return the anomalous event and the violated occurrence pattern can be extremely useful for guiding software developers through the diagnostics of the detected anomaly. Indeed, instead of simply saying that a given trace is anomalous, our approach returns the precise moment of time and the violated precedence relation that made the system enter the anomalous state.

VIII. Case studies

In this section, we present two case studies of mining task precedence graphs from system traces and using the mined graphs to detect anomalies in target systems. In the first case study, we consider four CAN message traces that were captured on a Hyundai YF Sonata car and were made available for download [2] by the researchers from the HCRL lab (Korea University). The car remained parked during trace collection. Each trace starts with a period of normal CAN bus activity followed by a period of anomalous activity caused by a simulated attack on the system. For the second case study, we obtained 15 CAN message traces from our partner in the automotive industry. The traces were captured on a production vehicle exercised on a test track during normal, anomaly-free operation. Therefore, in this case study we focus on the variation in the number of false positives returned by our algorithm with respect to the amount of training data.

The traces in both case studies are standard CAN bus activity logs, where each line shows the field values of the captured frame as well as the time when the frame appeared on the bus. Figure 4 shows a snippet of one of the CAN traces from the first case study. In this work, we dispose of all but the "message ID" field, which we interpret as the task name.

In the next two sections, we introduce an important type of attacks on modern vehicles called spoofing and explain how a TPG-based intrusion detection system can help in detecting this type of intrusions.

A. Background on Spoofing Attacks on Modern Vehicles

A *spoofing attack* is an intrusion technique when an attacker forges its identity to pass as another device. Spoofing has gained a special notoriety in wireless networks where cryptographic authentication is prohibitively expensive to apply due to limited power and resources available to wireless devices [9].

[2]http://ocslab.hksecurity.net/Datasets/CAN-intrusion-dataset

```
time                message ID   bytes   b1     b2     b3     b4     b5     b6     b7     b8
....                ....         ....    ....   ....   ....   ....   ....   ....   ....   ....
1478193190.235495   0545         8       d8     5f     00     8b     00     00     00     00
1478193190.235633   05f0         2       01     00
1478193190.236088   0130         8       0b     80     00     ff     0f     80     0c     ea
1478193190.236324   0131         8       f2     7f     00     00     15     7f     0c     35
1478193190.236565   0140         8       00     00     00     00     1e     0d     2c     3b
1478193190.236815   02c0         8       15     00     00     00     00     00     00     00
....                ....         ....    ....   ....   ....   ....   ....   ....   ....   ....
```

Fig. 4: A snippet of a CAN trace from Case Study I

Spoofing attacks have been recently shown to be a major security threat for modern automobiles [26] [28] [31]. A particular demonstration of a spoofing attack performed by Miller and Valasek [28] triggered a recall of 1.4 million of Chrysler cars in 2015 [3]. Another example is a series of successful hackings of a Tesla Model S by researchers from Tencent KeenLab in 2016 and 2017 [4].

We next explain what makes cars vulnerable to spoofing. Modern cars are pervasively computerized: dozens of embedded microprocessors called Electronic Control Units (ECU) run tens of millions of lines of code monitoring sensors, components, the driver, and the passengers [8]. The number of ECUs is constantly growing to meet the demand for new features that make driving more safe and enjoyable. Many car features, for example, park assist, adaptive cruise control, and collision prevention require complex real-time communications between ECUs [21] [13]. The prevailing in-vehicle communication protocol is Control Area Network (CAN) [30]. With the CAN protocol, ECUs are interconnected via a single or multiple CAN buses and communicate by broadcasting messages. This makes it possible for an attacker to send a message to any ECU once they have successfully infiltrated the CAN bus. Moreover, CAN messages contain no authenticator or even source identifier fields making it possible for an attacker to send messages that are indistinguishable from normal CAN traffic [21]. Finally, telematics, wifi, and bluetooth modules found in new cars allow an attacker to infiltrate the CAN bus wirelessly. These weaknesses of the CAN protocol make it possible to spoof CAN messages without physically accessing the vehicle.

One of the strategies to improve the resilience of cars to spoofing attacks consists in employing an intrusion detection system (IDS) running inside a car on one or multiple embedded microprocessors [27]. Existing anomaly-based IDS for the CAN bus are mainly based on timing information of CAN messages, since the messages are normally periodic [34]. The frequencies of CAN messages are learned during vehicle validation and then used to detect an unusual number of messages in the incoming CAN traffic using a sliding window [29] [26] [35] [33]. The motivation behind these IDS is that an attacker needs to introduce a high volume of spoofed messages to override the messages sent by the original ECU [27]. Therefore, these approaches detect an intrusion only after the number of spoofed messages in a fixed interval of time reaches a specific threshold.

B. *Detecting Spoofing Attacks with TPGs*

As we explained in Section VII, a task precedence graph trained on normal system traces can be used for online anomaly detection in trace streams. A TPG-based IDS would need to store the graph's nodes (i.e., an array of character strings), its incidence matrix, and a set of occurrence patterns corresponding to graph edges. The IDS would also need to keep two data structures for each edge: a current occurrence string and a number of tokens. Computation performed on an incoming trace event is a simple prefix verification (line 10 in Algorithm 4). The light computational and memory requirements make this method compatible with the hardware used in ECUs. Therefore, our TPG-based IDS can be programmed on an onboard car computer.

In contrast to intrusion detection methods presented in Section VIII-A, a TPG-based IDS makes it possible to detect even a single spoofed message, as long as the spoofed message has a predecessor and/or a successor message. In other words, an anomalous event ε can be detected using a task precedence graph G only if $\varepsilon.\mathrm{task}$ has an incoming and/or an outgoing edge in G. Indeed, if for a pair of tasks τ_i and τ_j, $\exists op(\tau_i, \tau_j)$, the number of occurrences of τ_i between consecutive invocations of τ_j does not correspond to the current element in $op(\tau_i, \tau_j)$, then the IDS will immediately detect the anomaly. The same holds in the situation where τ_j appears an unexpected number of times between consecutive invocations of τ_i: the $op(\tau_i, \tau_j)$ is violated which makes our IDS immediately report the anomaly.

C. *Case Study I*

The four traces we use in this case study contain CAN bus activity of a parked Hyundai YF Sonata before and during three types of attacks: a denial-of-service attack in trace #1 (*DoS_dataset.csv*), a fuzzing attack [21] in trace #2 (*Fuzzy_dataset.csv*), a spoofing attack on the drive gear in trace #3 (*gear_dataset.csv*), and a spoofing attack on the RPM gauge in trace #4 (*RPM_dataset.csv*). Each message in these traces is annotated with a flag which says whether the message is normal (flag "R") or forged (flag "T"). Therefore, it is possible to evaluate the accuracy of an IDS on these traces. We decided to disregard the first two attacks since they are performed by introducing CAN messages with random IDs (fuzzing) or

[3]https://www.theverge.com/2015/7/24/9032179

[4]https://www.pcmag.com/news/355281/tesla-model-s-hackers-return-for-encore-attack

with the highest-priority ID (denial-of-sevice), which makes their detection trivial. Indeed, it would suffice to remember the message IDs appearing on the bus in normal conditions, and then report any new message ID observed on the bus during the testing. On the other hand, the spoofing attacks (traces #3 and #4) forge messages with existing IDs, which makes their detection more difficult.

Given a lack of traces capturing normal car behavior in this case study, we trained a TPG on the first 1000 messages (all flagged "R", i.e., normal) from traces #1 and #2. We did not use more messages as training data since the attacks begin shortly after the 1000th message in these traces. The process of mining a TPG conformant to a set of traces consists in mining a TPG from a single trace, and then training the graph on the remaining traces. Therefore, we first mined a TPG from the trace #1 using Algorithms 2 and 3, and then trained the graph on the trace #2 using Algorithm 4. Figure 5 shows the trained TPG where each edge $\tau_i \rightarrow \tau_j$ is annotated with the occurrence pattern $op(\tau_i, \tau_j)$.

Next, we present the results of detecting anomalies in the trace #3 (drive gear spoofing) and in the trace #4 (RPM gauge spoofing) using the trained TPG.

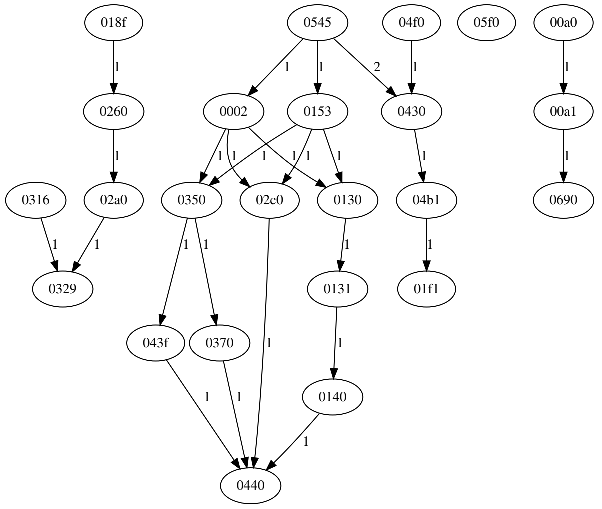

Fig. 5: A trained TPG from Case Study I

1) Detecting a spoofing attack on the drive gear: Our TPG-based anomaly detector reported 28 anomalies prior to the first forged message in the trace #3. The reported anomalies can be split into three groups: 18 anomalies were detected near the beginning of the trace (within the first 32 lines), one anomaly was signaled on line #57, and 9 anomalies were detected within 29 lines before the first spoofed message which appears on line #2142. The first group is clearly false positives. All of these anomalies occur when an occurrence pattern $op(\tau_i, \tau_j)$ is violated at the first occurrence of τ_j in the trace. Therefore, we suspect that our approach returns false positives because of a lack of synchronization between the beginning of trace recording and the start of system execution. Indeed, if a trace does not capture the first execution of τ_i but does capture the first execution of τ_j, then $op(\tau_i, \tau_j)$ will be violated if it exists in the training traces. The anomaly in the second group (also a false positive) can be attributed to insufficient training data, since the violated occurrence pattern $op(\texttt{0350}, \texttt{0370})$ was respected in the training data. Next, we explain why we consider the third group of detected anomalies as true positives. During a manual analysis of the trace, we noticed that the bus activity plunges moments before the first spoofed message appears on the bus. We speculate that this can be a result of activating an external board that the researchers used to introduce spoofed messages to the bus (using a procedure similar to the one they described in [23]). Our TPG-based IDS had detected this event and had raised an alarm before the first spoofed message manifested itself on the bus.

2) Detecting a spoofing attack on the RPM gauge: Our TPG-based anomaly detector reported 9 anomalies prior to the first forged message in the trace #3 appearing on line #1720. The first 6 anomalies were reported near the beginning of the trace (within the first 38 lines), one anomaly – on line #316, and the other two anomalies – close to the first spoofed frame (within 7 lines). The first 6 anomalies (false positives) can be explained in the same way as the first group of the anomalies in the drive gear example described above. Similarly, the single anomaly on line #316 (also a false positive) can be attributed to insufficient training data. As in the drive gear case, we consider the two anomalies reported prior to the appearance of the first spoofed message as true positives. In fact, the anomalies come from the violations of $op(\texttt{018f}, \texttt{02a0})$ and $op(\texttt{0260}, \texttt{02a0})$. When we manually analyzed the temporal behavior of task `02a0`, we found out that its period being regular throughout the normal part of the trace (10ms) was slightly violated prior to the spoofed message (9.5ms). We assume that this slight period violation was somehow caused by the connection to the CAN bus for the injection of spoofed messages.

D. Case Study II

In this case study, we evaluate the number of false positives returned by our TPG-based IDS while it is being trained on traces of normal CAN bus activity captured on a production car exercised on a test track. Table I shows the statistics of the 15 traces that we used in this case study.

Trace	Duration (s)	# events	# tasks
1	312	793,653	252
2	245.5	653,932	240
3	361.3	907,038	224
4	390	1,044,558	251
5	352.6	891,195	235
6	287	727,508	243
7	288.7	767,148	239
8	307.2	1,260,299	243
9	600	1,425,667	219
10	343.2	866,187	229
11	319	847,882	241
12	249.9	635,463	235
13	342.1	991,105	276
14	314.3	834,942	257
15	283.8	754,786	239

TABLE I: Trace statistics from Case Study II

We applied the same procedure of training a TPG on a set of traces that we used in the previous case study. The TPG mined from the first trace contains 116 nodes with at least one incoming and/or outgoing edge and 180 edges. At the same time, the TPG trained on all the 15 traces consists of 44 nodes with at least one incoming and/or outgoing edge and 43 edges. The trained graph with obfuscated task names is shown in Figure 6. The difference in number of nodes and edges between the graph mined from the first trace and the graph mined from all the 15 traces can be explained by the fact that the target vehicle was probably tested on different driving scenarios. This way, the TPG shown in Figure 6 reflects the car's behavior common to all the tested scenarios.

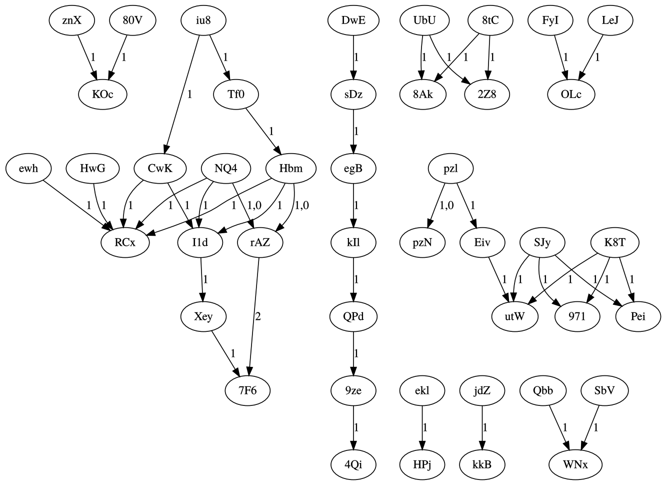

Fig. 6: A trained TPG from Case Study II

We evaluated the false positive rate of our precedence graph-based IDS using the following procedure. We mined a TPG from the first trace, then ran Algorithm 4 on the second trace using the mined graph and recorded the number of detected anomalies. The reported anomalies are, of course, false positives, since all 15 traces were captured during normal car operation. We then used the TPG trained on the first two traces, and repeated the same procedure on the third trace. Table II shows the number of false positives returned by our TPG-based IDS when it was applied on the i^{th} trace using the TPG trained on the first $i-1$ traces. As we can see in Table II, there were no false positives after the 8^{th} trace. This supports the observation made in previous works [27] [34] that CAN traffic is highly regular since it is machine-generated. This allows us to posit that a task precedence graph-based IDS requires only a limited amount of training data to reach a low or even perfect false positive rate.

IX. Limitations

Our task precedence graph mining method works only with *perfect* traces [41], that is, traces that capture all the executed events that the software developer has chosen to trace. Indeed, given a pair of real-time tasks τ_i and τ_j such that τ_j reads τ_i's output, if an execution trace W misses a single execution of τ_i (or τ_j), then Algorithm 1 will not mine the occurrence pattern $op(\tau_i, \tau_j)$ from W.

It follows from the previous limitation that our method requires the tracing to start when the system begins its execution. Consider the pair of real-time tasks from the previous paragraph. If a trace W starts with an invocation of task τ_j, then the first element of the occurrence string $os(\tau_i, \tau_j)$ will be 0. Therefore, Algorithm 1 will not mine $op(\tau_i, \tau_j)$ from the provided trace W.

Lastly, a TPG-based IDS does not allow to detect an anomalous event ε in a trace stream W if the corresponding task ε.task does not have at least one incoming and/or outgoing edge in the precedence graph. Indeed, the proposed IDS detects temporal violations of real-time tasks with respect to other tasks in the system. If a task is independent and does not have any precedence constraints with other tasks, violations of its temporal behavior must be detected using other anomaly detection methods. One idea of the future work in this direction could be verification of tasks' periods and response time profiles mined, for example, with the PeTaMi tool [20].

X. Conclusion

In this paper, we propose an approach to mine a task precedence graph from a set of system traces. The main idea behind our approach is that a real-time task τ_i that sends its output to a real-time task τ_j executes a deterministic number of times between a given pair of consecutive executions of τ_j. The proposed method firstly mines a TPG from a single execution trace and then trains the graph on the remaining traces. We also show how to use a TPG to find anomalies in incoming trace streams of an embedded system. The proposed anomaly detection method returns the first event that violates the TPG, facilitating the diagnostics of the detected problem; it also allows the software developer to mark the anomaly as a normal system behavior and resume anomaly detection with the updated TPG.

Trace	# anomalies
1	—
2	140
3	220
4	11
5	1
6	38
7	2
8	9
9	0
10	0
11	0
12	0
13	0
14	0
15	0

TABLE II: The number of anomalies in the i^{th} trace detected by our IDS using a task precedence graph trained on the previous $i-1$ traces from Case Study II

We also present two case studies where we apply our algorithms on CAN bus traces captured on different production cars. We show that a TPG-based intrusion detection system has good accuracy even when trained on a limited amount of data. Moreover, we explain why our approach can be superior to the methods available in the literature in detecting spoofing attacks on automobiles.

References

[1] Rakesh Agrawal, Dimitrios Gunopulos, and Frank Leymann. Mining process models from workflow logs. In *International Conference on Extending Database Technology*, pages 467–483. Springer, 1998.

[2] Glenn Ammons, Rastislav Bodík, and James R Larus. Mining specifications. *ACM Sigplan Notices*, 37(1):4–16, 2002.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16. ACM, 2002.

[4] Ivan Beschastnikh, Yuriy Brun, Michael D Ernst, and Arvind Krishnamurthy. Inferring models of concurrent systems from logs of their behavior with CSight. In *Proceedings of the 36th International Conference on Software Engineering*, pages 468–479. ACM, 2014.

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

[6] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering*, pages 33–42. ACM, 2006.

[7] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 24(5):823–839, 2012.

[8] Robert N Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.

[9] Yingying Chen, Wade Trappe, and Richard P Martin. Detecting and localizing wireless spoofing attacks. In *Sensor, Mesh and Ad Hoc Communications and Networks, 2007. SECON'07. 4th Annual IEEE Communications Society Conference on*, pages 193–202. IEEE, 2007.

[10] Jonathan E Cook and Alexander L Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(3):215–249, 1998.

[11] Francis Cottet, Jolle Delacroix, Claude Kaiser, and Zoubir Mammeri. *Scheduling in Real-Time Systems*. John Wiley & Sons, Ltd, 2003.

[12] Greta Cutulenco, Yogi Joshi, Apurva Narayan, and Sebastian Fischmeister. Mining timed regular expressions from system traces. In *Proceedings of the 5th International Workshop on Software Mining*, pages 3–10. ACM, 2016.

[13] Armaghan Darbandi, Seokhoon Yoon, and Myung Kyun Kim. Schedule construction under precedence constraints in FlexRay in-vehicle networks. *International Journal of Automotive Technology*, 18(4):671–683, 2017.

[14] Colin De la Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[15] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[16] David Gries, Alain J Martin, Jan LA van de Snepscheut, and Jan Tijmen Udding. An algorithm for transitive reduction of an acyclic graph. *Science of Computer Programming*, 12(2):151–155, 1989.

[17] Alireza Haghighatkhah, Ahmad Banijamali, Olli-Pekka Pakanen, Markku Oivo, and Pasi Kuvaja. Automotive software engineering: A systematic mapping study. *Journal of Systems and Software*, 128:25–55, 2017.

[18] Lennart Hofland and Joop van der Linden. Software in MRI scanners. *IEEE software*, 27(4):87, 2010.

[19] Gerard J Holzmann. Landing a spacecraft on Mars. *IEEE Software*, 30(2):83–86, 2013.

[20] Oleg Iegorov, Reinier Torres, and Sebastian Fischmeister. Periodic task mining in embedded system traces. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE*, pages 331–340. IEEE, 2017.

[21] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 447–462. IEEE, 2010.

[22] Johannes A La Poutré and Jan van Leeuwen. Maintenance of transitive closures and transitive reductions of graphs. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 106–120. Springer, 1987.

[23] Hyunsung Lee, Seong Hoon Jeong, and Huy Kang Kim. Otids: A novel intrusion detection system for in-vehicle network by using remote frame. In *Privacy, Security and Trust (PST) 2017*, 2017.

[24] Qing Li and Caroline Yao. *Real-time concepts for embedded systems*. CRC Press, 2003.

[25] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, pages 501–510. ACM, 2008.

[26] Charlie Miller and Chris Valasek. Adventures in automotive networks and control units. *DEF CON*, 21:260–264, 2013.

[27] Charlie Miller and Chris Valasek. A survey of remote automotive attack surfaces. *Black Hat USA*, 2014.

[28] Charlie Miller and Chris Valasek. Remote exploitation of an unaltered passenger vehicle. *Black Hat USA*, 2015.

[29] Michael Müter and Naim Asaj. Entropy-based anomaly detection for in-vehicle networks. In *Intelligent Vehicles Symposium (IV), 2011 IEEE*, pages 1110–1115. IEEE, 2011.

[30] Nicolas Navet and Françoise Simonot-Lion. A review of embedded automotive protocols. *Automotive Embedded Systems Handbook, Industrial Information Technology Series, pages*, pages 4–1, 2008.

[31] Keen Security Lab of Tecent. Car hacking sesearch: remote attack Tesla Motors. http://keenlab.tencent.com/en/2016/09/19/Keen-Security-Lab-of-Tencent-Car-Hacking-Research-Remote-Attack-to-Tesla-Cars/, 2016.

[32] Steven P Reiss and Manos Renieris. Encoding program executions. In *proceedings of the 23rd International Conference on Software Engineering*, pages 221–230. IEEE Computer Society, 2001.

[33] Hyun Min Song, Ha Rang Kim, and Huy Kang Kim. Intrusion detection system based on the analysis of time intervals of CAN messages for in-vehicle network. In *Information Networking (ICOIN), 2016 International Conference on*, pages 63–68. IEEE, 2016.

[34] Adrian Taylor. *Anomaly-based detection of malicious activity in in-vehicle networks*. PhD thesis, University of Ottawa, 2017.

[35] Adrian Taylor, Nathalie Japkowicz, and Sylvain Leblanc. Frequency-based anomaly detection for the automotive CAN bus. In *Industrial Control Systems Security (WCICSS), 2015 World Congress on*, pages 45–49. IEEE, 2015.

[36] Wil Van der Aalst, Ton Weijters, and Laura Maruster. Workflow mining: Discovering process models from event logs. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1128–1142, 2004.

[37] Wil MP van der Aalst. *Process mining: data science in action*. Springer, 2016.

[38] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[39] Anton JMM Weijters and Wil MP Van der Aalst. Rediscovering workflow models from event-based data using little thumb. *Integrated Computer-Aided Engineering*, 10(2):151–162, 2003.

[40] Dingkun Yang and Fei Hu. Schedulability of real-time systems with enhanced safety. In *Multimedia and Ubiquitous Engineering*, pages 391–398. Springer, 2014.

[41] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, pages 282–291. ACM, 2006.