Behavior Query Discovery in System-Generated Temporal Graphs

Bo Zong[1*] Xusheng Xiao[2] Zhichun Li[2] Zhenyu Wu[2] Zhiyun Qian[3]
Xifeng Yan[1] Ambuj K. Singh[1] Guofei Jiang[2]

[1]UC Santa Barbara [2]NEC Labs America, Inc. [3]UC Riverside

{bzong, xyan, ambuj}@cs.ucsb.edu {xsxiao, zhichun, adamwu, gfj}@nec-labs.com zhiyunq@cs.ucr.edu

ABSTRACT

Computer system monitoring generates huge amounts of logs that record the interaction of system entities. How to query such data to better understand system behaviors and identify potential system risks and malicious behaviors becomes a challenging task for system administrators due to the dynamics and heterogeneity of the data. System monitoring data are essentially heterogeneous temporal graphs with nodes being system entities and edges being their interactions over time. Given the complexity of such graphs, it becomes time-consuming for system administrators to manually formulate useful queries in order to examine abnormal activities, attacks, and vulnerabilities in computer systems.

In this work, we investigate how to query temporal graphs and treat *query formulation* as a discriminative temporal graph pattern mining problem. We introduce TGMiner to mine discriminative patterns from system logs, and these patterns can be taken as templates for building more complex queries. TGMiner leverages temporal information in graphs to prune graph patterns that share similar growth trend without compromising pattern quality. Experimental results on real system data show that TGMiner is 6-32 times faster than baseline methods. The discovered patterns were verified by system experts; they achieved high precision (97%) and recall (91%).

1. INTRODUCTION

Computer systems are widely deployed to manage the business in industry and government. Ensuring the proper functioning of these systems is critical to the execution of the business. For example, if a system is compromised, the security of the customer data cannot be guaranteed; if certain components of a system have failures, the services hosted in the system may be interrupted. Maintaining the proper functioning of computer systems is a challenging task. System experts have limited visibility into systems, as the tools they use often give a partial view of the complex systems. This motivates the recent trend of leveraging system monitoring logs to offer intelligence in system management.

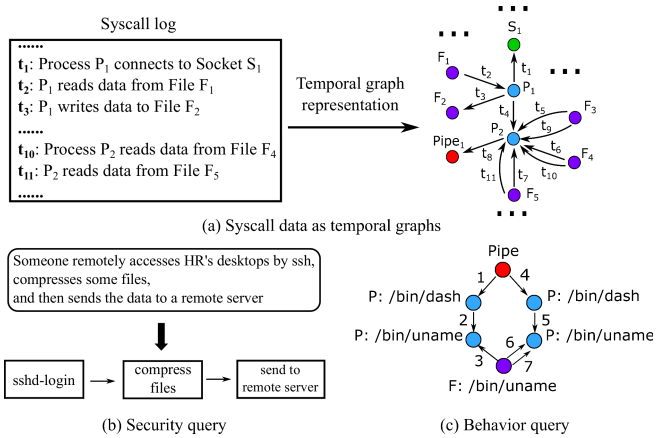


Figure 1: Capture system activities: (a) syscall data represented as temporal graphs, (b) a security query, and (c) a discriminative subgraph pattern representing an sshd-login activity.

Temporal graphs are key data structures used to fuse information from heterogeneous sources [33, 36]. In computer systems, system monitoring data are represented as temporal graphs. For example, in cybersecurity, system call (syscall) logs provide a comprehensive way to capture system activities [12]. Unlike its alternatives (*e.g.,* file access logs [1], firewall [4], and network monitoring [23]) which provide partial information and are application-specific, syscall logs cover all interactions among system entities (*e.g.,* processes, files, sockets, and pipes) over time. In Figure 1(a), a syscall log contains a sequence of events each of which describes at which time what kind of interactions happened between which system entities. Note that this syscall log also forms an equivalent temporal graph.

While temporal graphs provide an intuitive way to visualize system behaviors, another important step is to query such data and leverage the results to better comprehend system status. The question is how! It is usually burdensome to formulate useful queries to search system-generated temporal graphs, as they are complex with many tedious low-level entities. Consider the following scenarios.

EXAMPLE 1 (CYBERSECURITY). *To ensure the security of an enterprise system, a system expert wants to know if*

*Bo Zong is currently a researcher at NEC Labs America.

Proceedings of the VLDB Endowment, Vol. 9, No. 4

there exists any information stealthy activity in the human resource department over the weekend. A hypothetical activity could involve three steps: someone remotely accessed an HR desktop by ssh, compressed several files, and transferred them to a remote server. In order to find such activity, one can submit a query like Figure 1(b) consisting of three components: "sshd-login", "compress-files", and "send-to-remote-server", and perform search over syscall logs like Figure 1(a). However, such query cannot retrieve any useful information since the low-level entities (e.g., processes and files) recorded in the syscall logs cannot be directly mapped to any high-level activity like "sshd-login" or "compress-files". In order to locate all "sshd-login" activities, one has to know which processes or files are involved in "sshd-login" and in what order over time these low-level entities are involved in order to write a query. This becomes very time consuming.

Besides querying risky behaviors, the formulated behavior queries can also be applied on the real-time monitoring data for surveillance and policy compliance checking.

EXAMPLE 2 (DATACENTER MONITORING). *State-of-the-art system monitoring tools generate large-scale monitoring data as temporal graphs [36], where nodes are system performance alerts, and edges indicate dependencies between alerts. While these alerts suggest low-level anomalies (*e.g., *CPU usage is too high on server A, or there are too many full table joins on server B), system experts desire high-level knowledge about system behaviors: do these alerts result from* disk failure *or* abnormal database workload*? To search such high-level system behaviors, we have to know how alerts trigger each other and their temporal order [25]. It is a daunting task for system experts to manually formulate such queries.*

In addition, such query formulation problems also exist in other complex systems in the big data era.

EXAMPLE 3 (URBAN COMPUTING). *Modern cities generate a diverse array of data from heterogeneous sources, such as traffic flow, reports of sickness in cities, reports of food production, and so on [33]. Information inferred from these sources are fused into temporal graphs, where nodes are events detected from different sources (*e.g., *traffic jam, high sickness rate, and decrease in food production yield), edges indicate relationships between events (*e.g., *two events are geographically close), and timestamps record when such relationships are detected. Domain experts are interested in high-level knowledge in urban systems: are these unusual events caused by river or air pollution? To formulate queries and search such knowledge, domain experts have to understand detailed temporal dependency patterns between events, which is extremely difficult for them.*

Querying high-level system behaviors significantly reduces the complexity of evaluating system status, but it is quite difficult to formulate useful system behavior queries, referred to as *behavior queries* in this paper, because of the big semantic gap between the high-level abnormal activities and the low-level footprints of such activities. To address this problem, one approach is to collect monitoring data of target behaviors (*e.g.,* "sshd-login"), model the raw monitoring data by heterogeneous temporal graphs to , and use the full graphs to formulate queries. Unfortunately, the raw data can be large and noisy. To overcome this challenge, instead of using the full graphs, we identify the most discriminative patterns for target behaviors and treat them as queries. Such queries (*e.g.,* a few edges) are easier to interpret and modify, and are robust to noise. A discriminative pattern should frequently occur in the target activities, and rarely exist in other activities. One of the discriminative subgraph patterns for "sshd-login" is shown in Figure 1(c), which includes a few nodes/edges and is more promising for querying "sshd-login" from syscall logs.

To this end, we formulate the behavior query construction problem as a *discriminative temporal graph mining problem*: Given a positive set and a negative set of temporal graphs, the goal is to find the temporal graph patterns with maximum discriminative score. It is difficult to extend the existing mining techniques [10, 27] to solve this problem, since they mainly focus on non-temporal graphs, not temporal graphs (detailed discussion in Section 7.1).

In this paper, we propose TGMiner that addresses the challenges to discriminative temporal graph pattern mining.

1. We have to consider both topology and edge temporal order while searching temporal graph space. To avoid redundant search, do we need another complex canonical labeling method like [10, 28] for mining temporal graphs? In our study, we find the temporal information in graphs allows us to explore temporal graph space in a more efficient manner. In particular, we propose a pattern growth algorithm without any complex canonical labeling. It guarantees that all promising patterns are covered, and no redundant search.

2. Since temporal graph space is huge, a naive exhaustive search is slow even for small temporal graphs. To speed up search, we first identify general cases where we can conduct pruning. Then we propose algorithms to minimize the overhead for discovering the pruning opportunities: (1) By encoding temporal graphs into sequences, a light-weight algorithm based on subsequence tests is proposed to enable fast temporal subgraph tests; and (2) we compress residual graph sets into integers such that residual graph set equivalence tests are performed in constant time.

Our major contributions are as follows. First, motivated by the need of queries in system monitoring applications, we identify a challenging query formulation problem in complex temporal graphs. Second, we propose the idea of using discriminative subgraph pattern mining to automatically formulate behavior queries, significantly easing the query formulation. Third, we develop TGMiner that leverages temporal information to enable fast pattern mining in temporal graphs. Experimental results on real data show that the behavior queries constructed by TGMiner are effective for behavior analysis in cybersecurity applications, with high precision 97% and recall 91%, better than a non-temporal graph pattern based approach whose precision and recall are 83% and 91%, respectively. For mining speed, TGMiner is 6-32 times faster than baseline methods.

2. PROBLEM FORMULATION

As described earlier, our goal is to mine skeletons for behavior queries from temporal graphs derived from system monitoring data. In the following, we focus on cybersecurity applications for the ease of presentation, but the proposed

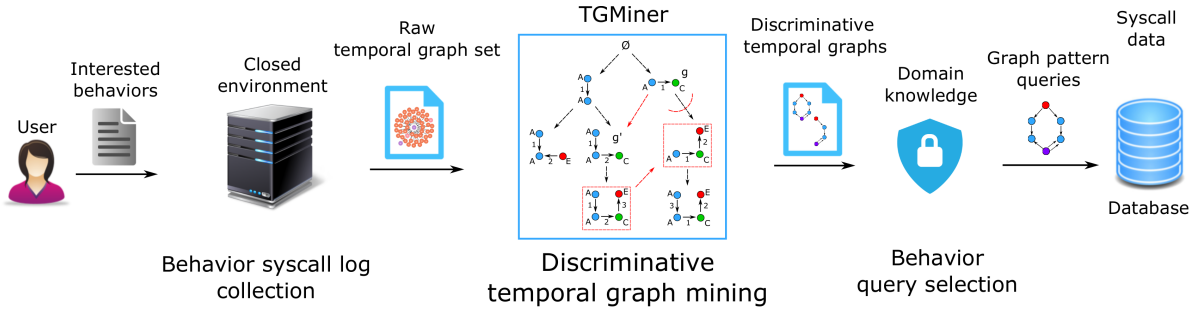


Figure 2: Behavior Query Formulation Pipeline

ideas and techniques can also be applied to applications in other domains.

Temporal graph. A temporal graph G is represented by a tuple (V, E, A, T), where (1) V is a node set; (2) $E \subset V \times V \times T$ is a set of directed edges that are totally ordered by their timestamps; (3) $A : V \to \Sigma$ is a function that assigns labels to nodes (Σ is a set of node labels); and (4) T is a set of possible timestamps, non-negative integers on edges.

In practice, the syscall data of a behavior instance is collected from a controlled environment, where only one target behavior is performed. In most cases, the syscall data of a target behavior forms a temporal graph of no more than a few thousand of nodes/edges.

We target temporal graphs with total edge order in this work. Our empirical results show that this model performs quite well in identifying basic security behaviors, as these behaviors are usually finished by one thread. It also possesses computation advantages, in comparison with more complex temporal graph models. For the cases of concurrent edges, we provide a discussion in Section 5.

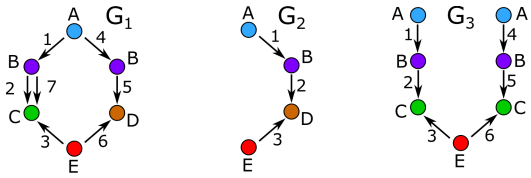


Figure 3: Temporal graphs: (1) G_2 is a temporal subgraph of G_1 ($G_2 \subseteq_t G_1$); and (2) G_1 and G_2 are T-connected, while G_3 is non T-connected.

Figure 3 shows three temporal graphs. Note that multi-edges are allowed in temporal graphs as shown in G_1. For simplicity, we examine temporal graphs with only node labels and edge timestamps in this paper. Our algorithms also work for temporal graphs with edge labels.

Temporal graph pattern. A temporal graph pattern $g = (V, E, A, T)$ is a temporal graph, where $\forall t \in T, 1 \leq t \leq |E|$.

Unlike general temporal graphs where timestamps could be arbitrary non-negative integers, timestamps in temporal graph patterns are aligned (from 1 to $|E|$) and only total edge order is kept. In the following discussion, we use upper case letters (such as G) to represent temporal graph data, and use lower case letters (such as g) to represent abstract temporal graph patterns.

As it is ineffective and inefficient to take the entire raw graph as a behavior query, it will be better to use its discriminative subgraphs to capture the footprint of a behavior.

Temporal subgraph. Given two temporal graphs $G = (V, E, A, T)$ and $G' = (V', E', A', T')$, $G \subseteq_t G'$ if and only if there exist two *injective* functions $f : V \to V'$ and $\tau : T \to T'$ such that (1) *node mapping*: $\forall u \in V$, $A(u) = A'(f(u))$; (2) *edge mapping*: $\forall (u, v, t) \in E$, $\big(f(u), f(v), \tau(t)\big) \in E'$; and (3) *edge order preserved*: $\forall (u_1, v_1, t_1), (u_2, v_2, t_2) \in E$, $\mathsf{sign}(t_1 - t_2) = \mathsf{sign}(\tau(t_1) - \tau(t_2))$.

G' is a *match* of G denoted as $G' =_t G$, when f and τ are *bijective* functions.

Figure 3 shows an example of a temporal subgraph where $G_2 \subseteq_t G_1$. In particular, the subgraph in G_1 formed by edges of timestamps 4, 5, and 6 is a match of G_2.

Pattern frequency. Given a set of temporal graphs $\mathbb{G}$ and a temporal graph pattern g, the frequency of g with respect to $\mathbb{G}$ is defined as

$$\mathsf{freq}(\mathbb{G}, g) = \frac{|\{G \mid g \subseteq_t G \wedge G \in \mathbb{G}\}|}{|\mathbb{G}|}.$$

Moreover, we differentiate two types of connected graphs among temporal graphs.

T-connected temporal graph. A temporal graph $G = (V, E, A, T)$ is T-connected if $\forall (u, v, t) \in E$, the edges whose timestamps are smaller than t form a connected graph.

In Figure 3, G_1 and G_2 are T-connected temporal graphs while G_3 is not, because the graph formed by edges with timestamps smaller than 5 is disconnected.

In this work, we focus on mining T-connected temporal graph patterns for the following reasons. First, in pattern growth, T-connected patterns remain connected, while non T-connected patterns might be disconnected during the growth process, resulting in formidable explosion of pattern search space. Second, any non T-connected temporal graph is formed by a set of T-connected temporal graphs. In practice, we can use a single T-connected pattern or a set of T-connected patterns that in all could be a non T-connected pattern to form a behavior query. In the rest of this paper, T-connected temporal graphs are referred to as connected temporal graphs without ambiguity. Next we define the discriminative temporal graph pattern mining problem.

PROBLEM 1. *Given a set of positive temporal graphs $\mathbb{G}_p$ and a set of negative temporal graphs $\mathbb{G}_n$, the goal is to find the connected temporal graph patterns g^* with maximum $F\big(\mathsf{freq}(\mathbb{G}_p, g^*), \mathsf{freq}(\mathbb{G}_n, g^*)\big)$, where $F(x, y)$ is a discriminative score function with partial (anti-)monotonicity:*

(1) when x is fixed, y is smaller, $F(x,y)$ is larger; and (2) when y is fixed, x is larger, $F(x,y)$ is larger.

$F(x,y)$ covers many widely used score functions including G-test, information gain, and so on [27]. In practice, one could pick a discriminative score function that satisfies partial (anti-)monotonicity and best fits his/her query formulation task. Note that for the ease of presentation, the discriminative score of a pattern g is also denoted as $F(g)$ in the following discussion.

Behavior query formulation pipeline. Figure 2 shows a pipeline of collecting syscall logs for behaviors, finding patterns, and using them to construct graph pattern queries for searching behaviors from syscall data (temporal graphs) and retrieving interesting security knowledge. We take the behavior of sshd-login as an example.

The first step is to form input data. Relatively clean syscall logs for sshd-login are crawled from a closed environment, where sshd-login is independently run multiple times. Additionally, we also collect syscall logs where sshd-login is not performed and treat them as background syscall logs. The input of mining sshd-login behavior patterns is formed as follows: (1) the raw syscall logs of sshd-login are treated as a set of positive temporal graphs $\mathbb{G}_p$ and (2) the raw background syscall logs are treated as a set of negative temporal graphs $\mathbb{G}_n$. In practice, we can also use the syscall logs for normal or abnormal sshd-login (*e.g.,* intrusion) as positive datasets, which will generate graph pattern queries for normal and abnormal behaviors.

Given $\mathbb{G}_p$ and $\mathbb{G}_n$, TGMiner finds the most discriminative temporal graph patterns for sshd-login. To identify the patterns that best serve the purpose of behavior search, the patterns discovered by TGMiner are further ranked by domain knowledge, including semantic/security implication on node labels and node label popularity among monitoring data. Top ranked patterns are then selected as queries to search sshd-login activities from a repository of syscall log data and see if there are abnormal/suspicious activities, *e.g.,* too many times of sshd-login over a Saturday night.

TGMiner **overview**. Our mining algorithm includes two key components: pattern growth and pattern space pruning.

Pattern growth guides the search in pattern space. It conducts depth-first search: starting with an empty pattern, growing it into a one-edge pattern, and exploring all possible patterns in its branch. When one branch is completely searched, the algorithm continues to explore the branches initiated by other one-edge patterns. The key challenges in this component is how to avoid repeated pattern search and how to cover the whole pattern space.

Pattern space pruning is a crucial step to speed mining processes. The underlying pattern space could be large, and a naive search algorithm cannot scale. Therefore, effective pruning algorithms are desired. The key problems in this component is how to identify general pruning opportunities and how to minimize the overhead in pruning.

3. TEMPORAL GRAPH GROWTH

In this section, we discuss the pattern growth algorithm in TGMiner. In particular, we demonstrate (1) how temporal information in graphs enables efficient pattern growth without repetition, and (2) the general principles to grow temporal graph patterns so that all possible connected temporal graph patterns will be covered.

3.1 Growth without Repetition

Pattern growth is more efficient in temporal graphs, compared with its counterpart in non-temporal graphs.

It is costly to conduct pattern growth for non-temporal graphs. To grow a non-temporal pattern to a specific larger one, there exist a combinatorial number of ways. In order to avoid repeated computation, we need extra computation to confirm whether one pattern is a new pattern or is a discovered one. This results in high computation cost, as graph isomorphism is inevitably involved. To reduce the overhead, various canonical labeling techniques along with their sophisticated pattern growth algorithms [8, 10, 28] have been proposed, but the cost is still very high because of the intrinsic complexity in graph isomorphism.

Unlike non-temporal graphs, we can develop efficient algorithms for temporal graph pattern growth. First, we can decide whether two temporal graph patterns are identical in linear time. Second, these is at most one possible way to grow a temporal graph pattern to a specific larger one. Next, we show why these properties stand.

The computation advantages of temporal graphs originate from the following property.

LEMMA 1. *Let g_1 and g_2 be temporal graph patterns. If $g_1 =_t g_2$, the mappings f and τ between them are unique.*

PROOF. The proof is detailed in Appendix A [37]. □

With Lemma 1, we further prove the efficiency of determining whether two temporal graph patterns are matched.

LEMMA 2. *If g_1 and g_2 are temporal graph patterns, then $g_1 =_t g_2$ can be determined in linear time.*

PROOF. The proof is detailed in Appendix B [37]. □

Pattern growth for temporal graphs will be more efficient, when it is guided by *consecutive growth*.

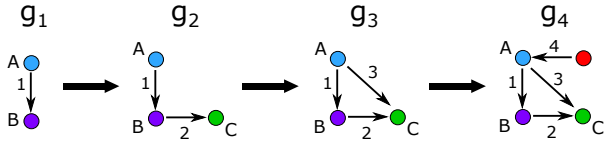


Figure 4: An example of consecutive growth

Consecutive growth. Given a connected temporal graph pattern g of edge set E and an edge $e' = (u', v', t')$, adding e' into g is consecutive growth, if (1) it results in another connected temporal graph pattern; and (2) $t' = |E|+1$.

Figure 4 demonstrates how g_1 grows to g_4 by consecutive growth. Consecutive growth guarantees a connected temporal graph pattern will form another connected temporal graph pattern without repetition.

LEMMA 3. *Let g_1 and g_2 be connected temporal graph patterns with $g_1 \subseteq g_2$. If pattern growth is guided by consecutive growth, then (1) either there exists a unique way to grow g_1 into g_2, (2) or there is no way to grow g_1 into g_2.*

PROOF. The proof is detailed in Appendix C [37]. □

Unlike mining non-temporal graphs, by Lemma 2 and 3, we can avoid repeated pattern search without using any sophisticated canonical labeling or complex pattern growth algorithms [10, 27]. Next, we show the three growth options that one needs to cover the whole pattern space.

3.2 Growth Options

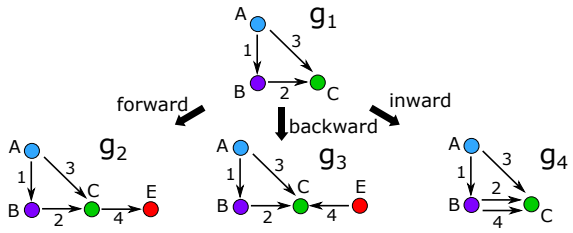


Figure 5: Three growth options

To guarantee the quality of discovered patterns, we need to ensure our search algorithm can cover the whole pattern space. In this work, we identify three growth options to achieve the completeness. Let g be a connected temporal graph pattern with node set V. We can grow g by consecutive growth as follows.

- Forward growth: growing g by an edge (u, v, t) is a forward growth if $u \in V$ and $v \notin V$.
- Backward growth: growing g by an edge (u, v, t) is a backward growth if $u \notin V$ and $v \in V$.
- Inward growth: growing g by an edge (u, v, t) is an inward growth if $u \in V$ and $v \in V$.

Figure 5 illustrates the three growth options. Note that inward growth allows multi-edges between node pairs.

The three growth options provide a guidance to conduct a complete search over pattern space.

THEOREM 1. *Let $\mathcal{A}$ be a search algorithm following consecutive growth with forward, backward, and inward growth. Algorithm $\mathcal{A}$ guarantees (1) a complete search over pattern space, and (2) no pattern will be searched more than once.*

PROOF. The proof is detailed in Appendix D [37]. □

Theorem 1 provides a naive exhaustive approach to mining discriminative temporal graph patterns. However, the underlying pattern space is usually huge, and the naive method suffers from poor mining speed. To speed up the mining process, we propose pruning algorithms in Section 4.

4. PRUNING TEMPORAL GRAPH SPACE

In this section, we investigate how to prune search space by the unique properties in temporal graphs. First, we explore the general cases where we can prune unpromising branches. Next, we identify the major overhead for discovering pruning opportunities, and leverage the temporal information in graphs to minimize the overhead.

4.1 Naive Pruning Conditions

A straightforward pruning condition is to consider the upper bound of a pattern's discriminative score. Given a temporal graph pattern g, the upper bound of g indicates the largest possible discriminative score that could be achieved by g's supergraphs. Let $\mathbb{G}_p$ and $\mathbb{G}_n$ be positive graph set and negative graph set, respectively. Since $\forall g \subseteq_t g'$, $\mathsf{freq}(\mathbb{G}_p, g') \leq \mathsf{freq}(\mathbb{G}_p, g)$ and $\mathsf{freq}(\mathbb{G}_n, g') \geq 0$, we can derive the following upper bound,

$$F\big(\mathsf{freq}(\mathbb{G}_p, g'), \mathsf{freq}(\mathbb{G}_n, g')\big) \leq F\big(\mathsf{freq}(\mathbb{G}_p, g), 0\big).$$

This upper bound is theoretically tight; however, it is ineffective for pruning in practice [27]. In the rest of this section, we discuss general pruning opportunities inspired by temporal sub-relations.

4.2 Pruning by Temporal Sub-relations

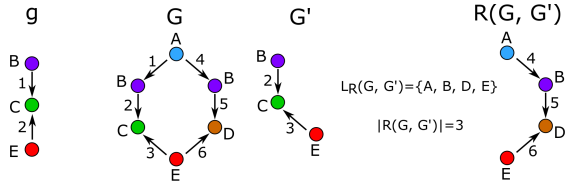


Figure 6: An example of residual graph

For a temporal graph pattern, we denote the graph data that need to be considered for growing this pattern as a set of *residual graphs*.

Let G' be a subgraph of G. If we remove the edges in G whose timestamps are no larger than the largest edge timestamp in G', then we form a residual graph.

Residual graph. Given a temporal graph $G = (V, E, A, T)$ and its subgraph $G' = (V', E', A', T')$, $R(G, G') = (V_R, E_R, A_R, T_R)$ is G's residual graph with respect to G', where (1) $E_R \subset E$ satisfies $\forall (u_1, v_1, t_1) \in E_R, (u_2, v_2, t_2) \in E'$, $t_1 > t_2$; and (2) V_R is the set of nodes that are associated with edges in E_R. The size of $R(G, G')$ is defined as $|R(G, G')| = |E_R|$ (*i.e.,* the number of edges in $R(G, G')$).

Given a residual graph $R(G, G')$, its residual node label set is defined as $L_R(G, G') = \{A_R(u) \mid \forall u \in V_R\}$.

Figure 6 demonstrates an example of residual graph, where G' is a subgraph of G, $R(G, G')$ is G's residual graph with respect to G', and $L_R(G, G')$ is its residual node set.

Let $\mathbb{M}(G, g)$ be a set including all the subgraphs in G that match a temporal graph pattern g. Given $\mathbb{G}_p$ and g, we define positive residual graph set $\mathbb{R}(\mathbb{G}_p, g)$ as

$$\mathbb{R}(\mathbb{G}_p, g) = \bigcup_{G \in \mathbb{G}_p} \{R(G, G') \mid G' \in \mathbb{M}(G, g)\}.$$

Given $\mathbb{R}(\mathbb{G}_p, g)$, its residual node label set $\mathbb{L}(\mathbb{G}_p, g)$ is defined as

$$\mathbb{L}(\mathbb{G}_p, g) = \bigcup_{G \in \mathbb{G}_p} \bigcup_{G' \in \mathbb{M}(G,g)} L_R(G, G').$$

Similarly, we can define negative residual graph set $\mathbb{R}(\mathbb{G}_n, g)$ and its residual node label set $\mathbb{L}(\mathbb{G}_n, g)$.

Next, we introduce another important property that helps us identify pruning opportunities.

PROPOSITION 1. *Given a temporal graph set $\mathbb{G}$ and two temporal graph patterns $g_1 \subseteq_t g_2$, if $\mathbb{R}(\mathbb{G}, g_1) = \mathbb{R}(\mathbb{G}, g_2)$, then the node mapping between g_1 and g_2 is unique.*

PROOF. We sketch the proof as follows. First, we prove if the node mapping between g_1 and g_2 is not unique, any match for g_2 includes multiple matches for g_1. Then we prove if any match for g_2 includes multiple matches for g_1,

$\mathbb{R}(\mathbb{G}, g_1) = \mathbb{R}(\mathbb{G}, g_2)$ will never be true. The detailed proof is stated in Appendix E [37]. □

In the following, we present *subgraph pruning* and *supergraph pruning*. In the following discussion, for a temporal graph pattern g, we use g's branch to refer to the space of patterns that are grown from g, and use F^* to denote the largest discriminative score discovered so far.

Let g_1 and g_2 be temporal graph patterns where g_1 is discovered before g_2. If (1) g_2 is a temporal subgraph of g_1; (2) they share identical positive residual graph sets; and (3) for those nodes in g_1 that cannot match to any nodes in g_2, their labels never appear in g_2's residual node label set, then we can conduct subgraph pruning on g_2.

Subgraph pruning. Given a discovered pattern $g_1 = (V_1, E_1, A_1, T_1)$ and a pattern g_2 of node set V_2, if (1) $g_2 \subseteq_t g_1$, (2) $\mathbb{R}(\mathbb{G}_p, g_2) = \mathbb{R}(\mathbb{G}_p, g_1)$, and (3) $\mathbb{L}(\mathbb{G}_p, g_2) \cap L_{g_1 \setminus g_2} = \emptyset$ (where $L_{g_1 \setminus g_2} = \{A_1(u) \mid \forall u \in V_1 \setminus V_1'\}$ and $V_1' \subseteq V_1$ is the set of nodes that map to nodes in V_2), we can prune the search on g_2's branch, if the largest discriminative score for patterns in g_1's branch is smaller than F^*.

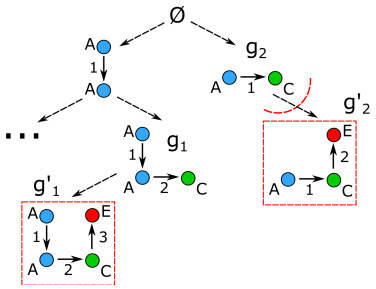


Figure 7: Subgraph pruning

Figure 7 illustrates the idea in subgraph pruning. In the mining process, we reach a pattern g_2, and we also notice that there exists a discovered pattern g_1, which satisfies the conditions in subgraph pruning. Therefore, pattern growth in g_1's branch suggests how to grow g_2 to larger patterns (*e.g.,* growing g_1 to g_1' indicates we can grow g_2 to g_2'). Since none of the patterns in g_1's branch have the score F^*, the patterns in g_2's branch cannot be the most discriminative ones as well, which can be safely pruned.

LEMMA 4. *Subgraph pruning prunes pattern space without missing any of the most discriminative patterns.*

PROOF. The proof is detailed in Appendix F [37]. □

Similar to subgraph pruning, we can perform supergraph pruning. Let g_1 and g_2 be temporal graph patterns where g_1 is discovered before g_2. If (1) g_1 is a temporal subgraph of g_2, (2) they share identical positive and negative residual graph sets, and (3) they have the same number of nodes, we can conduct supergraph pruning on g_2.

Supergraph pruning. Given two patterns g_1 and g_2, where g_1 is discovered before g_2 and g_2 is not grown from g_1, if (1) $g_2 \supseteq_t g_1$, (2) $\mathbb{R}(\mathbb{G}_p, g_2) = \mathbb{R}(\mathbb{G}_p, g_1)$, (3) $\mathbb{R}(\mathbb{G}_n, g_2) = \mathbb{R}(\mathbb{G}_n, g_1)$, and (4) g_2 and g_1 have the same number of nodes, the search in g_2's branch can be safely pruned, if the largest discriminative score for g_1's branch is smaller than F^*.

Figure 8 demonstrates the idea in supergraph pruning. In the mining process, a temporal graph pattern g_2 is reached,

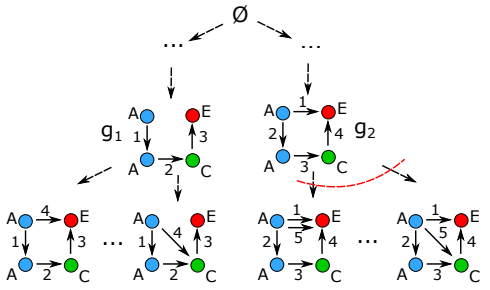


Figure 8: Supergraph pruning

and there is another pattern g_1 discovered before g_2, which satisfies the conditions in supergraph pruning. Therefore, the growth knowledge in g_1's branch suggests how to grow g_2 to larger patterns. Since none of the patterns in g_1's branch are the most discriminative, we can infer the patterns in g_2's branch are unpromising as well, and the search in g_2's branch can be safely pruned.

PROPOSITION 2. *Supergraph pruning prunes pattern space without missing the most discriminative patterns.*

PROOF. The proof is detailed in Appendix G [37]. □

Lemma 4 and Proposition 2 lead to the following theory.

THEOREM 2. *Performing subgraph pruning and supergraph pruning guarantees the most discriminative patterns will still be preserved.*

Theorem 2 identifies general cases where we can conduct pruning in temporal graph space; however, it works only if the overhead for discovering these pruning opportunities is small enough. The major overhead of subgraph pruning and supergraph pruning comes from two sources: (1) temporal subgraph tests (*e.g.,* $g_2 \subseteq_t g_1$), and (2) residual graph set equivalence tests (*e.g.,* $\mathbb{R}(\mathbb{G}_p, g_2) = \mathbb{R}(\mathbb{G}_p, g_1)$). For example, to mine patterns for "sshd-login" behavior, the mining process involves more than 70M temporal subgraph tests and 400M residual graph set equivalence tests. These overhead is not negligible, and it significantly degrades the efficiency of the pruning algorithm. Next, we investigate how to minimize these overhead.

4.3 Temporal Subgraph Test

In this section, we discuss how to leverage the temporal information in graphs to minimize the overhead from temporal subgraph tests. First, we propose an encoding scheme that represent temporal graphs by sequences. Second, we develop a light-weight algorithm based on subsequence tests.

Similar to subgraph test for non-temporal graphs, it is difficult to efficiently perform temporal subgraph tests.

PROPOSITION 3. *Given two temporal graphs g and g', it is NP-complete to decide $g \subseteq_t g'$.*

PROOF. We sketch the proof as follows. First, we prove the NP-hardness by reducing clique problem [15] to temporal subgraph test problem. By transforming non-temporal graphs into temporal graphs in polynomial time, we show we solve clique problem by temporal subgraph tests, which implies temporal subgraph test problem is at least as hard

as clique problem. Second, we prove temporal subgraph test problem is NP by showing we can verify its solution in polynomial time. The proof is detailed in Appendix H [37]. □

While existing algorithms [5, 21, 32] for non-temporal subgraphs provide possible solutions, temporal information in graphs suggests the existence of a faster solution.

1. Since edges are totally ordered in temporal graphs, it is possible to encode temporal graphs into sequences.

2. After temporal graphs are represented as sequences, it is possible to enable faster temporal subgraph tests using efficient subsequence tests.

Based on these insights, we propose a light-weight temporal subgraph test algorithm. In particular, this algorithm consists of two components: (1) a sequence-based temporal graph representation, and (2) a temporal subgraph test algorithm based on subsequence tests.

Before we dive into the technical details, we review the definition of subsequence. Let $s_1 = (a_1, a_2, ..., a_n)$ and $s_2 = (b_1, b_2, ..., b_m)$ be two sequences. If there exist $1 \leq i_1 < i_2 < ... < i_n \leq m$ such that $\forall 1 \leq j \leq n$, $a_j = b_{i_j}$, s_1 is a subsequence of s_2, denoted as $s_1 \sqsubseteq s_2$.

Sequence-based representation. A temporal graph pattern g can be represented by two sequences.

- Node sequence $\mathsf{nodeseq}(g)$ is a sequence of labeled nodes. Given g is traversed by its edge temporal order, nodes in $\mathsf{nodeseq}(g)$ are ordered by their first visited time. Any node of g appears only once in $\mathsf{nodeseq}(g)$.

- Edge sequence $\mathsf{edgeseq}(g)$ is a sequence of edges in g, where edges are ordered by their timestamps;

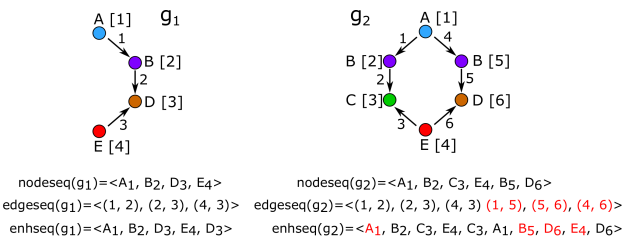


Figure 9: Sequence-based temporal graph representation and temporal subgraph tests

Figure 9 illustrates examples of sequence-based representation. In g_1 and g_2, node labels are represented by letters, and nodes of the same labels are differentiated by their node IDs represented by integers in brackets. Node labels in $\mathsf{nodeseq}$ are associated with node IDs as subscripts. Note that when we compare node labels, their subscripts will be ignored (*i.e.*, $\forall i, j, B_i = B_j$). Each edge in $\mathsf{edgeseq}$ is represented by the following format $(id(u), id(v))$, where $id(u)$ is the source node ID and $id(v)$ is the destination node ID.

Given two temporal graphs g_1 and g_2, if $g_1 \subseteq_t g_2$, we expect $\mathsf{nodeseq}(g_1) \sqsubseteq \mathsf{nodeseq}(g_2)$ and $\mathsf{edgeseq}(g_1) \sqsubseteq \mathsf{edgeseq}(g_2)$. However, when $g_1 \subseteq_t g_2$, $\mathsf{nodeseq}(g_1) \sqsubseteq \mathsf{nodeseq}(g_2)$ may not be true. As shown in Figure 9, $\mathsf{nodeseq}(g_1) \not\sqsubseteq \mathsf{nodeseq}(g_2)$ because the first visited time of the node with label E is inconsistent in g_1 and g_2.

To address this issue, we propose enhanced node sequence enhseq. Let g be a temporal graph. $\mathsf{enhseq}(g)$ is a sequence of labeled nodes in g. Given g is traversed by its edge temporal order, $\mathsf{enhseq}(g)$ is constructed by processing each edge (u, v, t) as follows. (1) If u is the last added node in the current $\mathsf{enhseq}(g)$, or u is the source node of the last processed edge, u will be skipped; otherwise, u will be added into $\mathsf{enhseq}(g)$. (2) Node v will be always added into $\mathsf{enhseq}(g)$. Note that nodes in g might appear multiple times in $\mathsf{enhseq}(g)$. Figure 9 shows the enhanced node sequences of g_1 and g_2.

With the support from enhanced node sequence, we are ready to build the connection between temporal subgraph tests and subsequence tests.

LEMMA 5. *Two temporal graphs $g_1 \subseteq_t g_2$ if and only if*

1. $\mathsf{nodeseq}(g_1) \sqsubseteq \mathsf{enhseq}(g_2)$*, where the underlying match forms an injective node mapping f_s from nodes in g_1 to nodes in g_2;*

2. $f_s(\mathsf{edgeseq}(g_1)) \sqsubseteq \mathsf{edgeseq}(g_2)$*, where $f_s(\mathsf{edgeseq}(g_1))$ is an edge sequence where the nodes in g_1 are replaced by the nodes in g_2 via the node mapping f_s.*

PROOF. The proof is detailed in Appendix I [37]. □

As shown in Figure 9, g_1 and g_2 are two temporal graphs satisfying $g_1 \subseteq_t g_2$. The node sequence of g_1 is a subsequence of the enhanced node sequence of g_2 with the injective node mapping $f_s(1) = 1$, $f_s(2) = 5$, $f_s(3) = 6$, and $f_s(4) = 4$. Therefore, we obtain $f_s(\mathsf{edgeseq}(g_1)) = \langle(1, 5), (5, 6), (4, 6)\rangle$ so that $f_s(\mathsf{edgeseq}(g_1)) \sqsubseteq \mathsf{edgeseq}(g_2)$.

Subsequence-test based algorithm. A temporal subgraph test algorithm is derived from Lemma 5. Given temporal graphs g_1 and g_2, the algorithm performs as follows.

1. Search an injective node mapping f_s between g_1 and g_2 such that $\mathsf{nodeseq}(g_1) \sqsubseteq \mathsf{enhseq}(g_2)$;

2. Test whether $f_s(\mathsf{edgeseq}(g_1)) \sqsubseteq \mathsf{edgeseq}(g_2)$. If yes, the algorithm terminates and returns $g_1 \subseteq_t g_2$; otherwise, the algorithm searches next qualified node mapping. If such a node mapping exists, repeat step 1 and 2; otherwise, the algorithm terminates with $g_1 \not\subseteq_t g_2$.

Although we can perform subsequence tests in linear time, many possible node mappings may exist. Among all the possible mappings, a large amount of them are false mappings, which are not injective. To further improve the speed, we adapt existing pruning techniques for subsequence matching to temporal subgraph tests. The idea is to identify false mappings as early as possible by leveraging node labels, local neighborhood information, and processed prefixes as conditions to prune unpromising search branches. We detail these pruning techniques in Appendix J [37].

4.4 Residual Graph Set Equivalence

In this section, we discuss how to efficiently test equivalent residual graph sets by leveraging temporal information in graphs. A naive approach applies a linear scan algorithm. Since residual graph set equivalence tests are frequently employed by subgraph and supergraph pruning, repeated linear scans causes significant overhead and suffers poor efficiency.

Let g_1 and g_2 be temporal graph patterns. Consider G_1' and G_2' are the matches of g_1 and g_2 in G, respectively. Since

edges in temporal graphs are totally ordered, we can derive the following result: $R(G, G'_1)$ is equivalent to $R(G, G'_2)$ if and only if $|R(G, G'_1)| = |R(G, G'_2)|$. Thus, we can efficiently conduct residual graph set equivalence tests as follows.

LEMMA 6. *Given temporal graph patterns g_1 and g_2 with $g_1 \subseteq_t g_2$, and a set of graphs $\mathbb{G}$, $\mathbb{R}(\mathbb{G}, g_1) = \mathbb{R}(\mathbb{G}, g_2)$ if and only if $I(\mathbb{G}, g_1) = I(\mathbb{G}, g_2)$, where*

$$I(\mathbb{G}, g_i) = \sum_{R(G,G') \in \mathbb{R}(\mathbb{G}, g_i)} |R(G, G')|.$$

PROOF. The proof is detailed in Appendix K [37]. □

Remark. We only need to pre-compute $I(\mathbb{G}, g)$ once by a linear scan over $\mathbb{R}(\mathbb{G}, g)$, and the following residual graph set equivalence tests are performed in constant time.

5. DISCUSSION: CONCURRENT EDGES

In a system with parallelism and concurrency, its monitoring data may generate concurrent edges (*i.e.,* edges sharing identical timestamps). In the following, we discuss how our technique can handle such cases.

First, we can extend TGMiner to mine patterns of concurrent edges by the following modifications.

- For temporal graph representation, instead of using a sequence of edges, we use a sequence of concurrent subgraphs, each of which includes all the edges sharing identical timestamps.
- In terms of pattern growth, if the added edge has a larger timestamp compared with the last added edge, follow the growth algorithm in TGMiner; if the added edge share the same timestamp with the last added edge, follow the growth algorithm in gSpan [28]; and we never add edges with smaller timestamps. To avoid repeated patterns, each concurrent subgraph is encoded by canonical labeling.
- For subgraph tests in pruning, we need to replace node matching based on labels with concurrent subgraph matching based on subgraph isomorphism, and ensure node mappings are injective.

Note that the computation complexity in the extended TGMiner will be increased, as the costly subgraph isomorphism is unavoidable for dealing with the non-temporal graphs formed by concurrent edges.

Second, instead of modifying the mining algorithm, we can transform concurrent edges into total-ordered edges. Despite of the existence of concurrency, data collectors can sequentialize concurrent events based on pre-defined policies [25, 36] (*e.g.,* randomly assigning a total time order for concurrent edges). In this way, we use the monitoring data with an artificial total order to approximate the original data, and apply TGMiner without modification. When there are a small portion of concurrent edges with minor accuracy loss, this method benefits the efficiency of TGMiner.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the proposed algorithms for behavior query discovery using real system activity data (*i.e.,* syscall logs). In particular, we focus on two aspects: (1) the effectiveness of the behavior queries found by our algorithms, and (2) the efficiency of the proposed algorithms.

6.1 Setup

We start with the description for the datasets investigated in our experimental study.

Behavior	Avg. #nodes	Avg. #edges	Total #labels	Size
bzip2-decompress	11	12	15	small
gzip-decompress	10	12	7	small
wget-download	33	40	92	small
ftp-download	30	61	39	small
scp-download	50	106	68	medium
gcc-compile	65	122	94	medium
g++-compile	67	117	100	medium
ftpd-login	28	103	119	medium
ssh-login	66	161	94	medium
sshd-login	281	730	269	large
apt-get-update	209	994	203	large
apt-get-install	1006	1879	272	large
background	172	749	9065	–

Table 1: Statistics in training data

Training data. We collect 13 datasets with in total 11,200 temporal graphs to mine behavior queries for 12 different behaviors. In total, it contains 1,905,621 nodes and 7,923,788 edges derived from the syscall logs including behaviors of interest and background system activities.

We focus on 12 behaviors as representatives for the basic behaviors that have drawn attention in cybersecurity study [2, 3, 30]. For each behavior, we collect 100 temporal graphs based on the syscall logs generated from 100 independent executions of the behavior, as we find the sample size of 100 already achieves good precision (97%) and recall (91%). For background system activities, 10,000 temporal graphs are sampled from 7 days' syscall logs generated by a server without performing any target behaviors.

The statistics of the training data are shown in Table 1. Although the size of a single temporal graph is not large, we find discriminative patterns within the data can involve up to 45 edges, which is a large number for pattern mining problems because of the exponential number of sub-patterns.

To evaluate how the effectiveness and efficiency are affected by different amounts of training data, we vary the amount of used training data from 0.01 to 1.0, where 0.01 means 1% of training data are used for behavior query discovery, and 1.0 means we consider all the data.

In addition, we generate synthetic datasets to evaluate the scalability of TGMiner. The synthetic datasets are created based on the training data: we generate datasets SYN-2, SYN-4, SYN-6, SYN-8, and SYN-10 by replicating each graph in the training data 2, 4, 6, 8, and 10 times, respectively.

Test data. Test data are used to evaluate the accuracy of the behavior queries found by our algorithms. They are obtained from an independent data collection process: we collect another seven days' syscall log data, which forms a large temporal graph with 2,352,204 nodes and 15,035,423 edges. The test data contain 10,000 behavior instances of the 12 target behaviors. Note that our focus in this paper is query formulation instead of pattern query processing. Based on the patterns discovered from training data, we formulate behavior queries, and search their existence from the test data by existing techniques [35].

More details about training/test data collection are presented in Appendix L [37].

Implementation. For effectiveness, we consider Ntemp and NodeSet as baselines. (1) Ntemp employs non-temporal graph patterns for behavior query discovery. In particular, we remove all the temporal information in the train-

ing data, apply existing algorithms [10] to mine discriminative non-temporal graph patterns for each behavior, and use the discovered patterns to formulate non-temporal behavior queries. (2) NodeSet searches behavior instances by keyword queries using a set of discriminative node labels. The discriminativeness of a node label is measured by the same score function $F(x, y)$ for temporal graph patterns. Top-k discriminative node labels are selected for a query of k node labels. A match of a query is a set of k nodes, where its node label set is identical to the node label set specified in the query, and its spanned time interval is no longer than the longest observed lifetime of the target behavior.

For efficiency, we implement five baseline algorithms to demonstrate the contributions of each component in TGMiner. All the baselines apply the proposed pattern growth algorithm and the naive pruning condition stated in Section 4.1. (1) SubPrune employs the pruning condition in Lemma 4. (2) SupPrune considers the pruning condition in Proposition 2. (3) PruneGI uses all the pruning conditions but applies a graph index based algorithm for temporal subgraph tests. In particular, we index one-edge substructures, and use efficient algorithms to join partial matches into full matches [35]. (4) PruneVF2 considers all the pruning conditions but performs temporal subgraph tests by a modified VF2 algorithm [5]. (5) LinearScan uses all the pruning conditions but performs residual graph set equivalence tests via a linear scan algorithm.

In addition, we employ information gain, G-test [27], as well as the function $F(x, y) = \log(x/(y + \epsilon))$ (ϵ is set to 10^{-6}) adopted in [10] as discriminative score functions. In our experiment, we find these score functions deliver a common set of discriminative patterns.

All the algorithms are implemented in C++ with GCC 4.8.2, and all the experiments are performed on a server with Ubuntu 14.04, powered by an Intel Core i7-2620M 2.7GHz CPU and 32GB of RAM. Each experiment is repeated 10 times, and their average results are presented.

6.2 Effectiveness of Behavior Queries

We evaluate the effectiveness of TGMiner from three aspects. (1) For different behaviors, how accurately can behavior queries suggested by TGMiner search behavior instances from system activity data? (2) How does query accuracy vary when pattern size in queries changes? (3) How does the amount of training data affect query accuracy?

Precision and recall are used as the metrics to evaluate the accuracy. Given a target behavior and its behavior query, a match of this behavior query is called an identified instance. An identified instance is correct, if the time interval during which the match happened is fully contained in a time interval during which one of the true behavior instances was under execution. A behavior instance is discovered, if the behavior query can return at least one correct identified instance with respect to this behavior instance. The precision and recall of a behavior query are defined as follows.

$$\text{precision} = \frac{\#\text{correctly identified instances}}{\text{total } \#\text{identified instances}},$$

$$\text{recall} = \frac{\#\text{discovered instances}}{\#\text{behavior instances}}.$$

Note that when TGMiner returns multiple discriminative patterns that have the same highest discriminative score, the returned patterns are further ranked by a score function based on domain knowledge. From all the discriminative patterns, top-5 patterns are used to build behavior queries. The details about the domain knowledge based score function is discussed in Appendix M [37].

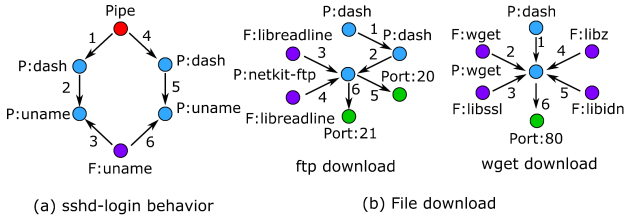


Figure 10: Discovered discriminative patterns

Figure 10 illustrates a few discovered discriminative patterns. For sshd-login behavior, the unique interaction pattern among system entities makes it accurate for search purpose. Note that this pattern does not include any node with "sshd" in the label. This indicates that keyword-based techniques that simply use application names as keywords (*e.g.,* sshd) cannot find such highly accurate patterns. For file download behaviors, it is the distinct patterns of how to access libraries and sockets that differentiate wget-based download from ftp-based download.

Metric	Precision (%)			Recall (%)		
Algorithm	NodeSet	Ntemp	TGMiner	NodeSet	Ntemp	TGMiner
bzip2-decompress	100	100	100	100	100	100
gzip-decompress	96.6	100	100	100	100	100
wget-download	96.5	100	100	93.6	93.4	93.4
ftp-download	100	100	100	100	96.1	96.1
scp-download	13.8	59.4	**100**	11.2	91.3	91.3
gcc-compile	69.7	81.2	**94.3**	89.2	89.4	87.6
g++-compile	73.4	91.3	**95.2**	84.5	85.3	85.3
ftpd-login	76.6	81.8	**94.1**	100	89.7	86.8
ssh-login	33.8	64.3	**93.9**	78.7	87.2	85.9
sshd-login	43.4	59.6	**99.9**	99.8	99.9	99.9
apt-get-update	50.3	79.3	**95.9**	47.6	84.5	82.4
apt-get-install	68.3	81.7	**95.7**	35.6	86.3	83.9
Average	68.5	83.2	**97.4**	78.4	91.9	91.1

Table 2: Query accuracy on different behaviors

Table 2 shows the precision and recall of behavior queries on all the 12 behaviors. The size (*i.e.,* the number of edges) of the behavior queries suggested by TGMiner and Ntemp is fixed as 6, using all the training data. NodeSet employs the top-6 discriminative node labels to query each behavior. First, the behavior queries suggested by TGMiner accurately discover behavior instances. Over all the behaviors, its average precision and recall are 97.4% and 91.1%, respectively. Second, the queries provided by Ntemp only achieve 83.2% of precision, suffering significantly higher false positive rate. This result indicates the importance of temporal information in searching system behaviors. Third, the queries suggested by TGMiner consistently outperform the queries formed by NodeSet. These results confirm that temporal graph patterns discovered by TGMiner provide high-quality skeletons to formulate accurate behavior queries.

Figure 11 demonstrates how the precision and recall of behavior queries suggested by TGMiner vary while their query size ranges from 1 to 10. All the training data were used in this experiment, and the average precision and recall over all the behaviors are reported. First, when the query size increases, the precision increases, but the recall decreases. In general, increasing behavior query size improves query

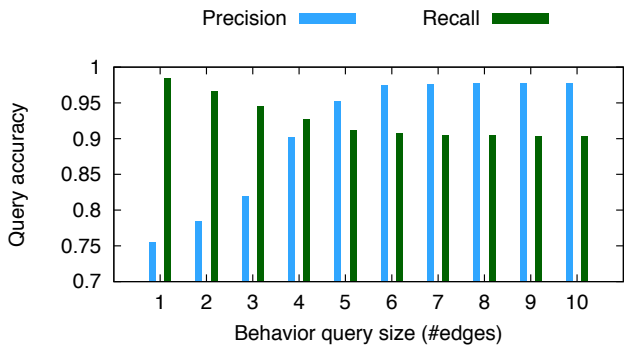


Figure 11: Query accuracy with different query sizes

precision at the cost of a slightly higher false negative rate. Second, when the query size goes beyond 6, we observe little improvement on precision or loss on recall.

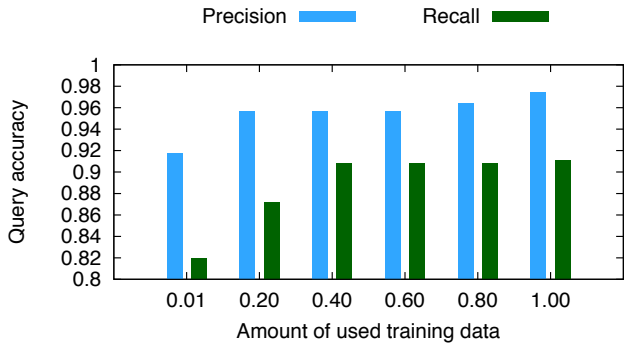


Figure 12: Query accuracy with different amounts of used training data

Figure 12 suggests how precision and recall of behavior queries vary as the amount of used training data is changed from 0.01 to 1.0. Note that behavior query size is fixed as 6, and the average precision and recall over all the behaviors are presented. First, more training data bring higher precision and recall. For example, when the amount of used training data increases from 0.01 to 1.0, the precision of the constructed behavior queries increase from 91% to 97%. Second, when the amount of used training data increases, we observe a diminishing return for both precision and recall.

6.3 Efficiency in Behavior Query Discovery

The efficiency of TGMiner is evaluated from two aspects. (1) How is the efficiency of TGMiner compared with baseline algorithms? (2) How is the efficiency of TGMiner affected by different amounts of used training data? Note that behavior query discovery is an offline step: we only need to mine patterns once, and then use the patterns to formulate queries serving online search demands.

Response time is used as the metric to evaluate the efficiency. Given input temporal graph sets, the response time of an algorithm is the amount of time the algorithm spends in mining all discriminative patterns.

Figure 13 demonstrates response time of all algorithms over small, medium, and large size behaviors as categorized in Table 1. All the training data are used in this experiment, and the discovered patterns have up to 45 edges.

First, TGMiner consistently outperforms the baseline algorithms over all the target behaviors. TGMiner is up to 50 and 4 times faster than SubPrune and SupPrune. SupPrune cannot finish the mining tasks for medium and large behaviors within 2 days. We notice most of the pruning opportunities come from subgraph pruning, while supergraph pruning brings additional performance improvement.

Second, TGMiner performs up to 6, 17, and 32 times faster than PruneGI, LinearScan, and PruneVF2, respectively. PruneGI has to frequently build graph indexes for each discovered patterns during the whole mining process, which involves high overhead. Indeed, graph indexing is more suitable for querying large graphs, where we can build graph indexes offline. In our case, a light-weighted temporal subgraph test algorithm performs better. In sum, the performance improvement highlights the importance of minimizing the overhead in temporal subgraph tests and residual graph set equivalence tests.

Third, for small, medium, and large behaviors, TGMiner can complete mining tasks within 6 seconds, 4 minutes, and 26 minutes, respectively. We also noticed that TGMiner mines all discriminative patterns with no more than 6 edges within one minute for all the behaviors, while 6-edge patterns can achieve good search accuracy as shown before.

Figure 14 presents the response time of TGMiner as the size of the largest patterns that are allowed to explore is set to 5, 15, 25, 35, and 45. When the size increases, the response time of TGMiner increases. When the size is set as 5, TGMiner can finish the mining tasks within 10 seconds for all the behaviors,.

Pruning Condition	Small	Medium	Large
Subgraph pruning	71.8%	61.0%	62.2%
Supergraph pruning	1.1%	8.3%	4.2%

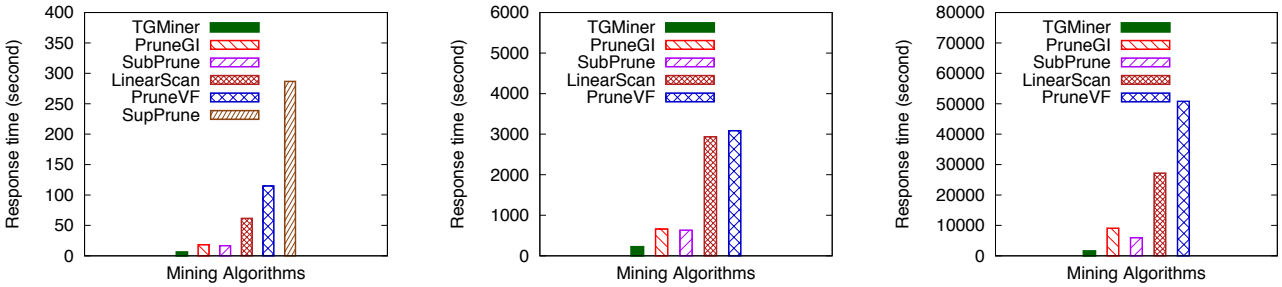
Table 3: Empirical probabilities that pruning conditions are triggered on behaviors of different sizes

Table 3 shows the empirical probabilities that subgraph and supergraph pruning are triggered when TGMiner is processing a pattern for behaviors of different sizes. The high trigger rate of subgraph pruning is consistent over all the behaviors, which explains its high pruning power.

Figure 15 shows how the response time of TGMiner is affected by using different amounts of training data. Ranging from 0.01 to 1.0, the response time of TGMiner increases linearly when more training data is considered. The scalability test on synthetic datasets (detailed in Appendix N [37]) also shows that the response time of TGMiner linearly scales with the size of training data. From the training data with up to 20M nodes and 80M edges, TGMiner can mine all discriminative patterns of up to 45 edges within 3 hours.

6.4 Summary

The experimental results are summarized as follows. First, behavior queries discovered by TGMiner achieve high precision (97%) and recall (91%), while the baseline algorithms Ntemp and NodeSet suffers poor accuracy. Second, TGMiner consistently outperforms the baseline algorithms in terms of efficiency, and is up to 6, 17, and 32 times faster than PruneGI, LinearScan, and PruneVF2, respectively.



(a) Small behavior traces (b) Medium behavior traces (c) Large behavior traces

Figure 13: Response time for behaviors of different sizes

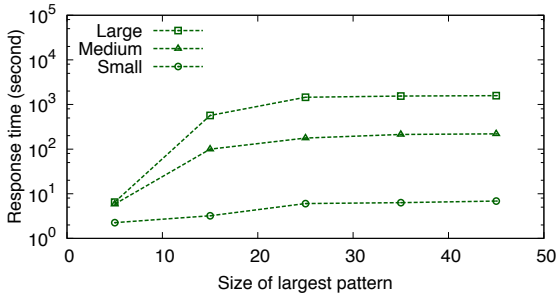


Figure 14: Response time by varying the size of the largest patterns that are allowed to explore

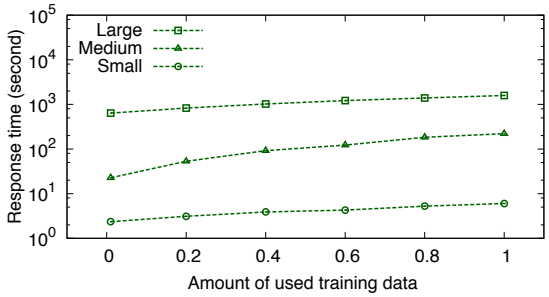


Figure 15: Response time by varying the amount of used training data

7. RELATED WORK

7.1 Discriminative Graph Pattern Mining

Discriminative graph pattern mining is one of the feature selection methods that is widely applied in a variety of graph classification tasks [18, 22, 27].

In general, two directions have been investigated to speed up mining discriminative non-temporal graph patterns. One direction is to find discriminative patterns early such that unpromising search branches can be pruned early [27]. The other direction relies on approximate search that finds good-enough graph patterns [10]. In addition, a few studies [17, 24] focus on mining patterns from graph snapshots.

It is difficult to extend existing work on non-temporal graphs to mine temporal graph patterns. The key problem is how to deal with timestamps in the mining process.

One possibility is to ignore timestamps: mine discriminative non-temporal patterns by existing approaches, and then use timestamps to find discriminative temporal patterns from the non-temporal patterns. This method has two drawbacks. First, since canonical labeling on non-temporal graphs [10, 28] have difficulties in dealing with multi-edges, we have to collapse multi-edges into a single edge. In this way, the final result will be partial, as it excludes patterns with multi-edges. Second, a large number of temporal patterns may share the same non-temporal patterns, and a discriminative non-temporal pattern may result in no discriminative temporal pattern. The redundancy in non-temporal patterns will bring potential scalability problems.

Another possibility is to consider timestamps as labels. The temporal patterns discussed in this paper focus on the temporal order instead of exact timestamps. Therefore, it is difficult to mine the desired patterns by the approaches for non-temporal patterns.

Our work is different from existing works. First, unlike existing studies that mainly deal with non-temporal graphs, we propose a solution to temporal graphs. Second, compared with existing works on graph snapshots, we study more flexible temporal graph patterns. The graph patterns defined in [17, 24] are too rigid to support system management.

7.2 Temporal Graph Management

Recent studies on temporal graphs strive to develop cost-effective solutions to management problems.

There have been a few works on developing more efficient index-free algorithms for time-dependent shortest-path [29], minimum temporal path [26], and anomaly detection [20].

Graph indexing, compression, and partitioning have been proposed to accelerate query processing for subgraph matching [35], reachability [34], community searching [9], and neighborhood aggregation [16]. Incremental computation has been investigated for queries including shortest-path [19], SimRank [31], and cluster searching [14]. In addition, Gao et al. [6] exploited distributed computation to monitor subgraph matching queries.

Data storage is a critical component in temporal graph management. Chronos [7] investigated temporal data locality for query processing. Efficient ways to store and retrieve graph snapshots have been studied in [11, 13].

Unlike these works, our focus is to develop cost-effective algorithms that help users formulate meaningful temporal graph queries, bridging the gap between users' knowledge and temporal graph data.

8. CONCLUSION

Computer system monitoring generates huge temporal graphs that record the interaction of system entities. While these graphs are promising for system experts to query behaviors for system management, it is also difficult for them to compose queries as it involves many tedious low-level system entities. We formulated this query formulation problem as a discriminative temporal graph mining problem, and introduced TGMiner to mine discriminative patterns that can be taken as query templates for building more complex queries. TGMiner leverages temporal information in graphs to enable efficient pattern space exploration and prune unpromising search branches. Experimental results on real system data show that TGMiner is 6-32 times faster than baseline methods. Moreover, the discovered patterns were verified by system experts: they achieved high precision (97%) and recall (91%).

Acknowledgement. We would like to thank the anonymous reviewers for the helpful comments on earlier versions of the paper. This research was partially supported by the Army Research Laboratory under cooperative agreements W911NF-09-2-0053 (NS-CTA), NSF IIS-1219254, and NSF IIS-0954125. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notice herein.

9. REFERENCES

[1] Splunk. `http://www.splunk.com/`.

[2] Ssh brute force - the 10 year old attack that still persists. `http://blog.sucuri.net/2013/07/`.

[3] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, and C. Kruegel. A view on current malware behaviors. In *LEET*, 2009.

[4] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. 2003.

[5] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub) graph isomorphism algorithm for matching large graphs. *TPAMI*, pages 1367–1372, 2004.

[6] J. Gao, C. Zhou, J. Zhou, and J. X. Yu. Continuous pattern detection over billion-edge graph using distributed framework. In *ICDE*, pages 556–567, 2014.

[7] W. Hant, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen. Chronos: a graph engine for temporal graph analysis. In *EuroSys*, pages 1–14, 2014.

[8] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.

[9] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu. Querying k-truss community in large and dynamic graphs. In *SIGMOD*, pages 1311–1322, 2014.

[10] N. Jin, C. Young, and W. Wang. Gaia: graph classification using evolutionary computation. In *SIGMOD*, pages 879–890, 2010.

[11] U. Khurana and A. Deshpande. Efficient snapshot retrieval over historical graph data. In *ICDE*, pages 997–1008, 2013.

[12] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[13] A. G. Labouseur, P. W. Olsen, and J.-H. Hwang. Scalable and robust management of dynamic graph data. In *BD3 at VLDB*, pages 43–48, 2013.

[14] P. Lee, L. V. Lakshmanan, and E. E. Milios. Incremental cluster evolution tracking from highly dynamic network data. In *ICDE*, pages 3–14, 2014.

[15] R. G. Michael and S. J. David. Computers and intractability: a guide to the theory of np-completeness. 1979.

[16] J. Mondal and A. Deshpande. Eagr: Supporting continuous ego-centric aggregate queries over large dynamic graphs. In *SIGMOD*, pages 1335–1346, 2014.

[17] S. Ranu, M. Hoang, and A. Singh. Mining discriminative subgraphs from global-state networks. In *KDD*, pages 509–517, 2013.

[18] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *ICDE*, pages 844–855, 2009.

[19] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On querying historical evolving graph sequences. In *VLDB*, pages 726–737, 2011.

[20] K. Sricharan and K. Das. Localizing anomalous changes in time-evolving graphs. In *SIGMOD*, pages 1347–1358, 2014.

[21] Z. Sun, H. Wang, H. Wang, B. Shao, and J. Li. Efficient subgraph matching on billion node graphs. In *VLDB*, pages 788–799, 2012.

[22] M. Thoma, H. Cheng, A. Gretton, J. Han, H.-P. Kriegel, A. J. Smola, L. Song, S. Y. Philip, X. Yan, and K. M. Borgwardt. Near-optimal supervised feature selection among frequent subgraphs. In *SDM*, pages 1076–1087, 2009.

[23] W. Venema. TCP wrapper: Network monitoring, access control, and booby traps. In *USENIX Security*, 1992.

[24] B. Wackersreuther, P. Wackersreuther, A. Oswald, C. Böhm, and K. M. Borgwardt. Frequent subgraph discovery in dynamic networks. In *MLG*, pages 155–162, 2010.

[25] P. Wang, H. Wang, M. Liu, and W. Wang. An algorithmic approach to event summarization. In *SIGMOD*, pages 183–194, 2010.

[26] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu. Path problems in temporal graphs. In *VLDB*, pages 721–732, 2014.

[27] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD*, pages 433–444, 2008.

[28] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.

[29] Y. Yang, H. Gao, J. X. Yu, and J. Li. Finding the cost-optimal path with time constraint over time-dependent graphs. In *VLDB*, pages 673–684, 2014.

[30] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS*, pages 116–127, 2007.

[31] W. Yu, X. Lin, and W. Zhang. Fast incremental simrank on link-evolving graphs. In *ICDE*, pages 304–315, 2014.

[32] S. Zhang, J. Yang, and W. Jin. Sapper: subgraph indexing and approximate matching in large graphs. In *VLDB*, pages 1185–1194, 2010.

[33] Y. Zheng, H. Zhang, and Y. Yu. Detecting collective anomalies from multiple spatio-temporal datasets across different domains. In *SIGSPATIAL*, 2015.

[34] A. D. Zhu, W. Lin, S. Wang, and X. Xiao. Reachability queries on large dynamic graphs: A total order approach. In *SIGMOD*, pages 1323–1334, 2014.

[35] B. Zong, R. Raghavendra, M. Srivatsa, X. Yan, A. K. Singh, and K.-W. Lee. Cloud service placement via subgraph matching. In *ICDE*, pages 832–843, 2014.

[36] B. Zong, Y. Wu, J. Song, A. K. Singh, H. Cam, J. Han, and X. Yan. Towards scalable critical alert mining. In *KDD*, pages 1057–1066, 2014.

[37] B. Zong, X. Xiao, Z. Li, Z. Wu, Z. Qian, X. Yan, A. K. Singh, and G. Jiang. Behavior query discovery in system-generated temporal graphs. arXiv:1511.05911.