Behavior Query Discovery in System-Generated Temporal Graphs

Bo Zong¹ Xusheng Xiao² Zhichun Li² Zhenyu Wu² Zhiyun Qian³
Xifeng Yan¹ Ambuj K. Singh¹ Guofei Jiang²
¹UC Santa Barbara ²NEC Labs America, Inc. ³UC 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 log data that record the interaction of system entities such as processes, files, and sockets. How to query such data to better understand system behaviors and identify potential security risks and malicious behaviors becomes a challenging task for system administrators due to the dynamics and heterogeneity of the data. System monitoring data is essentially a heterogeneous temporal graph with nodes being system entities and edges being their interactions over time. Given the complexity of such graphs and intrusion tactics, it becomes time-consuming for system administrators to manually formulate useful queries in order to examine suspicious 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 traces, 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

Cybersecurity is one of the top priorities in industry and government. The goal is to ensure the security of customer data and proper execution of business. However, defending computer systems is an extremely challenging task. System administrators often have limited visibility into systems. The tools they have access to often give only a partial view of the complex systems. This motivates the recent trend of leveraging big data to offer security intelligence. In this paper, we propose to offer system administrators a powerful tool to ask high-level security queries by analyzing large amounts of data, applying data analytics to extract security knowledge, and then exploiting such knowledge to harden systems and defend against security intrusions.

Among the possible solutions, system call (syscall) logs provide a comprehensive way to capture system activities [20, 21]. Unlike its alternatives [2] such as file access logs [18], firewall [7], and network monitoring [23, 42] which usually provide partial information and are application-specific, syscall logs cover all interactions among system entities such as processes, files, sockets, and pipes over time. Figure 1(a) shows a syscall log containing a sequence of entries each of which describes at which time what kind of interactions happened between which system entities. Syscall logs are essentially temporal graphs recording the dynamic interactions among system entities.

One can query syscall data and leverage it to better understand the security status of a system. The question is how! It is usually burdensome to formulate useful queries to search syscall graphs as they involve many tedious low-level entities. Consider the following scenario.

EXAMPLE 1. A system administrator wants to know if there exists any information stealing activity in a 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-
Although this query is meaningful, we cannot retrieve any useful information as we cannot directly find anything named “sshd-login” or “compress-files” from syscall logs. In order to find all sshd-login activities, one has to know which processes or files are involved in sshd-login, in what order over time and then write a query to search them. This becomes very time consuming.

As shown in the above example, there is a big semantic gap between the high-level suspicious activities and the low-level footprints of such activities. To address this problem, a straightforward approach is to collect syscalls of target behaviors (e.g., “sshd-login”), and use the raw syscall sequences as queries. Unfortunately, the raw syscalls can be very long (with thousands of syscalls) and their arguments and ordering can be different from time to time. To overcome this challenge, we choose to represent the raw syscalls as a heterogeneous temporal graph which is much more stable (as shown in Figure 1). The nodes represent system entities, while the edges represent their interactions with timestamps. The edge directions imply data flows.

Still, the size of such system behavior graphs can be large. Instead of using the full graph to construct a query, it is better to identify the most discriminative patterns of 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 desired pattern should frequently occur in the target activities, and rarely exist in other activities.

A discriminative subgraph pattern of “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 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 discriminative graph pattern mining techniques [15, 16, 46] to solve this problem, since they mainly focus on static graphs, not temporal graphs.

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 [16, 47] 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 security-related logs, 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 security applications, with high precision 97% and recall 91%, better than a static 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 query skeletons for basic security-related behaviors (behaviors for short) in temporal graphs derived from syscall data. In this paper, we focus on cybersecurity applications, but the proposed ideas and techniques can also be applied to other applications.

**Temporal graph.** A temporal graph $G$ is represented by a tuple $(V,E,A,T)$, where (1) $V$ is a set of nodes; (2) $E \subseteq V \times V \times T$ is a set of directed edges that are totally ordered by their timestamps; (3) $A : V \rightarrow \Sigma$ is a function that assigns labels to nodes ($\Sigma$ 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 focus on 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.

**Figure 3:** Temporal graphs: (1) $G_2$ is a temporal subgraph of $G_1$ ($G_2 \subseteq_1 G_1$); 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 multiple 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.
The raw syscall log of a behavior could form a temporal graph with thousands of nodes/edges. It is ineffective to take the entire raw graph as a query graph since it could be slow for search. In contrast, it might be better to use its discriminative subgraphs as they best capture the footprint of a behavior.

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

\( G' \) is a match of \( G \) denoted as \( G' =_1 G \), when \( f \) and \( \tau \) are bijective functions.

**Pattern frequency.** Given a set of temporal graphs \( G \) and a temporal graph pattern \( g \), the frequency of \( g \) with respect to \( G \) is defined as

\[
\text{freq}(G, g) = \frac{|\{ G \mid g \subseteq G \land G \in G \}|}{|G|}.
\]

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 this work, we focus on mining T-connected temporal graph patterns for the following reasons. First, as one grows patterns, T-connected graph patterns remain connected. Meanwhile, non T-connected graph patterns might be disconnected during the growth process, resulting in formidable explosion of pattern search space. Second, T-connected temporal graphs are more fundamental as a non T-connected temporal graph can be represented by a set of T-connected temporal graphs. 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 \( G_p \) and a set of negative temporal graphs \( G_n \), the goal is to find the connected temporal graph patterns \( g^* \) with maximum \( F(\text{freq}(G_p, g^*), \text{freq}(G_n, g^*)) \), 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.

\( F(x, y) \) with the partial (anti-)monotonicity property actually covers many widely used score functions including G-test, information gain, and so on [46]. 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 identifying 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 \( G_p \) and (2) the raw background syscall logs are treated as a set of negative temporal graphs \( G_n \). 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 \( G_p \) and \( G_n \), 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., sshd-login many times 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 challenges in this
component is (1) how to avoid repeated pattern search and (2) how to cover the whole pattern space.

Pattern space pruning is a crucial step to speed mining processes. The underlying pattern space is exponentially huge: $l$ node labels and $m$ temporal edges can form $O(n^2 m^m)$ different star patterns without considering more complex topology or multi-edges. A naive search algorithm cannot scale, so effective pruning algorithms are desired. The challenges in this component is (1) what are the general pruning opportunities and (2) how to identify these pruning cases with small overhead.

In Section 3 and Section 4, we discuss how we address the challenges in these two components in details.

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 static graphs.

It is costly to conduct pattern growth for static graphs. To grow a static graph 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 [12, 16, 47] have been proposed, but the cost is still very high because of the intrinsic complexity in graph isomorphism.

Unlike static 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 $g_1 \Rightarrow g_2$, the mappings $f$ and $\tau$ between them are unique.

**Proof.** Without loss of generality, we assume $g_1 = (V_1, E_1, A_1, T_1)$ and $g_2 = (V_2, E_2, A_2, T_2)$.

First, we prove $\tau: T_1 \rightarrow T_2$ is unique. Since $g_1$ and $g_2$ are temporal graph patterns, we have $\forall (u_1, v_1, t_1) \in E_1$, $1 \leq t_1 \leq |E_1|$ and $\forall (u_2, v_2, t_2) \in E_2$, $1 \leq t_2 \leq |E_2|$. Because $g_1 \Rightarrow g_2$, and $|E_1| = |E_2|$, we can derive $(u_1, v_1, t_1) \in E_1$ matches $(u_2, v_2, t_2) \in E_2$ only if $t_1 = t_2$ (in order to preserve total edge order). Thus, the uniqueness of $\tau$ is proved.

Second, we prove $f: V_1 \rightarrow V_2$ is unique. Since $\tau$ is unique, the edge mapping between $g_1$ and $g_2$ is unique, and then we can derive the node mapping $f$ is also unique.

Therefore, we have proved the uniqueness of $f$ and $\tau$. $\Box$

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 $g_1 \Rightarrow g_2$ can be determined in linear time.

**Proof.** We first present the linear algorithm, and then prove its correctness. Suppose $g_1 = (V_1, E_1, A_1, T_1)$, $g_2 = (V_2, E_2, A_2, T_2)$, $|V_1| = |V_2|$, and $|E_1| = |E_2|$.

**Algorithm.** We conduct a linear scan over edges in $g_1$. For each edge $(u_1, v_1, t) \in E_1$, we locate the edge $(u_2, v_2, t) \in E_2$. If such an edge exists, we verify if the mapping from $u_1$ to $u_2$ and the mapping from $v_1$ to $v_2$ are still one-to-one. If both are, we confirm $(u_1, v_1, t)$ matches $(u_2, v_2, t)$.

**Correctness.** We prove the correctness by showing the above algorithm finds two bijective functions $f: V_1 \rightarrow V_2$ and $\tau: T_1 \rightarrow T_2$. First, since the linear scan follows the unique way to match edge timestamps between $g_1$ and $g_2$ and $|E_1| = |E_2|$, $\tau$ is found and bijective. Second, the verification phase in the above algorithm guarantees the node mapping $f$ is one-to-one; moreover, $f$ is a full mapping because $|E_1| = |E_2|$ and all the nodes in $g_1$ and $g_2$ find their mapping in the verification phase. Therefore, we have proved the correctness of the linear algorithm. $\Box$
In sum, we have proved the correctness of Lemma 3. □

Unlike mining static graphs, by Lemma 2 and Lemma 3, we can avoid repeated pattern search without using any sophisticated canonical labeling or complex pattern growth algorithms [15, 16, 46]. Next, we show the three growth options that one needs to cover the whole pattern space.

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

**Proof.** We first justify the zero-redundancy in pattern search, and then prove the completeness.

Let \( g \) be a connected temporal graph pattern. By Lemma 3, we can tell consecutive growth guarantees there exists a unique way to grow an empty pattern into \( g \). Thus, there is no way to search \( g \) more than once.

For the completeness, we establish the proof by mathematical induction. Let \( m \) be the number of edges in a temporal graph pattern.

**Basis:** we prove the completeness holds for \( m = 1 \). Since 1-edge patterns are formed by distinct edges, it is easy to see all 1-edge patterns will be covered.

**Inductive step:** we prove if the completeness holds for \( m = k \), then it holds for \( m = k + 1 \).

Assume the completeness holds for \( m = k \). Then we have the complete set of \( k \)-edge connected temporal graph patterns \( H^{(k)} \). Assume \( g^{(k+1)} \) is \( g^{(k)} \) plus an edge \((e)\) is a connected pattern of \( k + 1 \) edges that is grown from \( g^{(k)} \) of \( k \) edges. Since the three growth options are all possible ways to keep patterns connected during growth, if \( g^{(k+1)} \) cannot be covered by growing patterns in \( H^{(k)} \), it implies \( g^{(k)} \notin H^{(k)} \), that is, \( g^{(k)} \) is not connected, which contradicts with the assumption that \( g^{(k+1)} \) is connected (T-connected). Therefore, the completeness also holds for \( m = k + 1 \).

Since both the basis and inductive step have been performed, by mathematical induction, the completeness holds.

In sum, we have proved Theorem 1. □

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 \( G_p \) and \( G_n \) be positive graph set and negative graph set, respectively. Since \( \forall g \subseteq g' \), \( \text{freq}(G_p, g') \leq \text{freq}(G_p, g) \) and \( \text{freq}(G_n, g') \geq 0 \), we can derive the following upper bound:

\[
F(\text{freq}(G_p, g'), \text{freq}(G_n, g')) \leq F(\text{freq}(G_p, g), 0).
\]

This upper bound is theoretically tight; however, it is in-effective for pruning in practice [46]. In the rest of this section, we discuss general pruning opportunities inspired by temporal sub-relations.

4.2 Pruning by Temporal Sub-relations

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) \( E_R \subseteq E \) 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 \( L_R(G, G') = \{A(u) \mid \forall u \in V_R \} \).
that are grown from set of nodes that map to nodes in (where (2) they share identical positive residual graph sets; and (3) in Subgraph pruning with respect to \( R \) we can conduct subgraph pruning on \( g \) and \( g' \).

Given \( R(G_p,g) \), its residual node label set \( L(G_p,g) \) is defined as

\[
L(G_p,g) = \bigcup_{G \in C_p} \bigcup_{G' \in M(G,g)} L_R(G,G').
\]

Similarly, we can define negative residual graph set \( R(G_n,g) \) and its residual node label set \( L(G_n,g) \).

**Proposition 1.** Given a temporal graph set \( G \) and two temporal graph patterns \( g_1 \subseteq g_2 \), if \( R(G,g_1) = R(G,g_2) \), then the node mapping between \( g_1 \) and \( g_2 \) is unique.

**Proof.** The proof is detailed in Appendix A [1]. □

Next, we introduce 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) \( g_2 \subseteq g_1 \); (2) \( R(G_p,g_2) = R(G_p,g_1) \); and (3) \( L(G_p,g_2) \cap L_{g_1} = \emptyset \) (where \( L_{g_1} = \{ A_1(u) \mid u \in V_1 \setminus V_2 \} \) and \( V_2 \subseteq V_1 \) 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 \( F^* \).

**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) \( g_2 \supseteq g_1 \); (2) \( R(G_p,g_2) = R(G_p,g_1) \); (3) \( R(G_n,g_2) = R(G_n,g_1) \); 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 \( F^* \).

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

**Proof.** Let \( g_1 \) and \( g_2 \) be temporal graph patterns, where \( g_1 \) is discovered before \( g_2 \) and they satisfy the conditions in subgraph pruning.

First, since the conditions in subgraph pruning are satisfied, we can derive the following facts: (1) \( \text{freq}(G_p,g_2) = \text{freq}(G_p,g_1) \) and (2) pattern growth in \( g_1 \)'s branch will never touch the nodes that cannot map to any nodes in \( g_2 \) as \( L(G_p,g_2) \cap L_{g_1} = \emptyset \).

Second, we prove none of the patterns in \( g_2 \)'s branch will be the most discriminative, if the conditions in subgraph pruning are satisfied and none of the patterns in \( g_1 \)'s branch is the most discriminative. Assume there exists a pattern \( g_2 \)'s discriminative score is no less than \( F^* \) and \( s \) is the sequence of consecutive growth that grows \( g_2 \) to \( g_2' \). Since no pattern growth in \( g_1 \)'s branch will touch the nodes that cannot map to any nodes in \( g_2 \), we can safely claim \( s \) also indicates a valid sequence of consecutive growth (with some timestamp shift) that grows \( g_1 \) into \( g_1' \) by Proposition 1. By \( \text{freq}(G_p,g_2') = \text{freq}(G_p,g_1) \), we can infer \( \text{freq}(G_p,g_2') = \text{freq}(G_p,g_1) \). It is not hard to see \( g_2' \subseteq g_1' \). Thus, we have \( \text{freq}(G_n,g_2') \geq \text{freq}(G_n,g_1) \).

Therefore, we can infer \( F(g_2') \leq F(g_1) \), which means \( g_1 \) is one of the most discriminative patterns which contradicts with the condition that none of the patterns in \( g_1 \)'s branch is the most discriminative. Therefore, we can claim any patterns in \( g_2 \)'s branch will have discriminative score less than \( F^* \), and the branch can be safely pruned.

In all, we have proved Lemma 4 holds. □

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) \( g_2 \) is a temporal subgraph of \( g_1 \); (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) \( g_2 \supseteq g_1 \); (2) \( R(G_p,g_2) = R(G_p,g_1) \); (3) \( R(G_n,g_2) = R(G_n,g_1) \); 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 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.

**Figure 8:** Supergraph pruning

Figure 8 demonstrates the idea in supergraph pruning. In the mining process, a temporal graph pattern \( g_2 \) is reached, 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 B [1].

Lemma 4 and Proposition 2 lead to the following theorem.

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., \( R(G_p, g_2) = R(G_p, g_1) \)). For example, to mine patterns for “sshdl-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 static graphs, it is difficult to efficiently perform temporal subgraph tests in general.

Proposition 3. Given two temporal graphs \( g \) and \( g' \), it is NP-complete to decide \( g \subseteq g' \).

Proof. The proof is detailed in Appendix C [1].

While existing algorithms [9, 37, 39, 51] for static 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 \( 1 \leq i_1 < i_2 < ... < i_m \leq m \) such that \( V_1 \leq j \leq n, a_j = b_{i_j} \), \( s_1 \) is a subsequence of \( s_2 \), denoted as \( s_1 \subseteq s_2 \).

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

- Node sequence \( \text{nodeseq}(g) \) is a sequence of labeled nodes. Given \( g \) is traversed by its edge temporal order, nodes in \( \text{nodeseq}(g) \) are ordered by their first visited time. Any node of \( g \) appears only once in \( \text{nodeseq}(g) \).
- Edge sequence \( \text{edgeseq}(g) \) 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 \( \text{nodeseq} \) 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 \( \text{edgeseq} \) 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 \( g_2 \), we expect \( \text{nodeseq}(g_1) \subseteq \text{nodeseq}(g_2) \) and \( \text{edgeseq}(g_1) \subseteq \text{edgeseq}(g_2) \). However, when \( g_1 \subseteq g_2 \), \( \text{nodeseq}(g_1) \subseteq \text{nodeseq}(g_2) \) may not be true. As shown in Figure 9, \( \text{nodeseq}(g_1) \not\subseteq \text{nodeseq}(g_2) \) 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 \( \text{enhseq} \). Let \( g \) be a temporal graph. \( \text{enhseq}(g) \) is a sequence of labeled nodes in \( g \). Given \( g \) is traversed by its edge temporal order, \( \text{enhseq}(g) \) is constructed by processing each edge \((u, v, t)\) as follows.

1. If \( u \) is the last node in current \( \text{enhseq}(g) \), or \( u \) is the source node of the last processed edge, \( u \) will be skipped; otherwise, \( u \) will be added into \( \text{enhseq}(g) \).
2. Node \( v \) will be always added into \( \text{enhseq}(g) \). Note that nodes in \( g \) might appear multiple times in \( \text{enhseq}(g) \). 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. \( \text{nodeseq}(g_1) \subseteq \text{enhseq}(g_2) \), where the underlying match forms an injective node mapping \( f_1 \) from nodes in \( g_1 \) to nodes in \( g_2 \);

2. \( f_s(\text{edgeseq}(g_1)) \subseteq \text{edgeseq}(g_2) \), where \( f_s(\text{edgeseq}(g_1)) \) is an edge sequence where the nodes in \( g_1 \) are replaced by the nodes in \( g_2 \) via the node mapping \( f_1 \).

Proof. The proof is detailed in Appendix G [1].

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 \( f_1(1) = 1, f_1(2) = 5, f_1(3) = 6, \) and \( f_1(4) = 4 \). Therefore, we obtain \( f_s(\text{edgeseq}(g_1)) = ((1, 5), (5, 6), (4, 6)) \) so that \( f_s(\text{edgeseq}(g_1)) \subseteq \text{edgeseq}(g_2) \).

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 \( f_1 \) between \( g_1 \) and \( g_2 \) such that \( \text{nodeseq}(g_1) \subseteq \text{enhseq}(g_2) \).
2. Test whether \( f_s(\text{edgeseq}(g_1)) \subseteq \text{edgeseq}(g_2) \). If yes, the algorithm terminates and returns \( g_1 \subseteq \tau g_2 \); otherwise, the algorithm searches next qualified node mapping. If such a node mapping exists, repeat step 1 and 2; otherwise, the algorithm terminates with \( g_1 \not\subseteq \tau g_2 \).

Although we can perform subsequence tests in linear time, many node mappings may exist. To further improve the speed, we adapt existing pruning techniques for subsequence matching to temporal subgraph tests. The details about these pruning techniques are discussed in Appendix D [1].

### 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 \( g_1 \subseteq g_2 \) and a set of graphs \( G \), \( \mathbb{R}(G, g_1) = \mathbb{R}(G, g_2) \) if and only if \( I(G, g_1) = I(G, g_2) \), where

\[
I(G, g) = \sum_{R(G, G') \in \mathbb{R}(G, g)} |R(G, G')|.
\]

**Proof.** We prove the sufficiency and necessity as follows.

First, we prove if \( g_1 \subseteq g_2 \) and \( I(G, g_1) = I(G, g_2) \), then \( \mathbb{R}(G, g_1) = \mathbb{R}(G, g_2) \). Let \( G_2 \) be the match of \( g_2 \) in \( G \). Since \( g_1 \subseteq g_2 \), there exists at least one subgraph \( G_1' \) in \( G_2' \) that matches \( g_1 \), where \( |R(G, G_1')| \geq |R(G, G_2')| \). Moreover, the subrelation \( g_1 \subseteq g_2 \) also implies \( |\mathbb{R}(G, g_1)| \geq |\mathbb{R}(G, g_2)| \).

If \( I(G, g_1) = I(G, g_2) \), then \( |\mathbb{R}(G, g_1)| = |\mathbb{R}(G, g_2)| \), and there exists a one-one mapping from the residual graphs in \( \mathbb{R}(G, g_1) \) to the residual graphs in \( \mathbb{R}(G, g_2) \). \( \forall R(G, G_1') \in \mathbb{R}(G, g_1) \), its mapped \( R(G, G_2') \in \mathbb{R}(G, g_2) \) has to satisfy \( |R(G, G_1')| = |R(G, G_2')| \) such that \( I(G, g_1) = I(G, g_2) \). Therefore, \( R(G, G_1') \) is equivalent to \( R(G, G_2') \), and moreover, \( \mathbb{R}(G, g_1) = \mathbb{R}(G, g_2) \).

Second, we prove if \( g_1 \subseteq g_2 \) and \( \mathbb{R}(G, g_1) = \mathbb{R}(G, g_2) \), then \( I(G, g_1) = I(G, g_2) \). Since \( \mathbb{R}(G, g_1) = \mathbb{R}(G, g_2) \), \( g_1 \) and \( g_2 \) share the same set of residual graphs. Therefore, we can directly infer \( I(G, g_1) = I(G, g_2) \).

In sum, we have proved Lemma 6.

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

### 5. 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.

#### 5.1 Setup

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

### Table 1: Statistics in training data

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Avg. #nodes</th>
<th>Avg. #edges</th>
<th>Total #labels</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip-decompress</td>
<td>11</td>
<td>12</td>
<td>15</td>
<td>small</td>
</tr>
<tr>
<td>gzip-compress</td>
<td>10</td>
<td>12</td>
<td>7</td>
<td>small</td>
</tr>
<tr>
<td>apt-get-update</td>
<td>1006</td>
<td>1879</td>
<td>272</td>
<td>large</td>
</tr>
<tr>
<td>apt-get-install</td>
<td>1172</td>
<td>749</td>
<td>9065</td>
<td>large</td>
</tr>
</tbody>
</table>

**Training data.** We collected 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 [3, 5, 14, 49]. For each behavior, we collected 100 temporal graphs based on the syscall logs generated from 100 independent executions of the behavior, as we found the sample size of 100 already achieves good precision (97%) and recall (91%). For background system activities, 10,000 temporal graphs were sampled from 7 days’ syscall logs generated by a server without performing any target behaviors.

The detailed statistics about the training data are shown in Table 1. Although the size of a single temporal graph is not large, we found discriminative patterns within the data can involve up to 40 edges, which is a large number for pattern mining problems because of the exponential number of sub-patterns. Our algorithm accelerates pattern mining by pruning redundant sub-patterns in search branches.

In addition, 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.

**Test data.** Test data are used to evaluate the accuracy of the behavior queried found by our algorithms. They were obtained from an independent data collection process: we collected 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 pattern queries for target behaviors, and search their existence from the test data by existing techniques [53].

More details about training/test data collection are presented in Appendix E [1].

**Implementation.** For effectiveness, we consider two baselines: Static and NodeSet. (1) Static considers static graph patterns for behavior query discovery. In particular, we remove all the temporal information in the training data, apply existing algorithms [16] to mine discriminative static graph patterns for each behavior, and use the discovered patterns to formulate static graph pattern queries to search behavior instances from the test data. (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 tem-
portional 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 the spanned time interval of the match is no longer than the longest observed lifetime of the target behavior.

For efficiency, we implemented 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 [53]. (4) PruneVF2 considers all the pruning conditions but conducts temporal subgraph tests by a modified VF2 algorithm [9]. (5) LinearScan uses all the pruning conditions but performs residual graph set equivalence tests via a linear scan algorithm.

In addition, we used the widely adopted discriminative function \( F(x, y) = \log \frac{\#correctly identified instances}{\# discovered instances} \) [15, 16] in the experiments, where \( \epsilon \) is set to \( 10^{-6} \). This function delivers good results based on our empirical study, but it does not mean this is the best choice among all the possibilities. The discussion over which discriminative function best fits security applications is beyond the scope of this paper.

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

5.2 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 employed 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.

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

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

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 F [1].

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.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm</td>
<td>NodeSet Static</td>
<td>TGMiner Static</td>
</tr>
<tr>
<td>apt-get-update</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>apt-get-install</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>bzip2-decompress</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>gzip-decompress</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>wget-download</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>ftp-download</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>scp-download</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>gcc-compile</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>g++-compile</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>rdp-login</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>sshd-login</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>ssh-login</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>apt-get-update</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>apt-get-install</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2: Query accuracy on different behaviors

Table 2 shows the precision and recall of behavior queries on all 12 behaviors. The size (i.e., the number of edges) of the behavior queries suggested by TGMiner and Static 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 Static only achieve 83.2% of precision, suffering significantly higher false positive rate. This result indicates the importance of temporal information in searching security-related 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 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.
achieve good search accuracy as shown before.

one minute for all the behaviors, while 6-edge patterns can all discriminative patterns with no more than 6 edges within with up to 40 edges. We also noticed that 26 minutes, respectively, and mine discriminative patterns large behaviors, 
is up to 50 times faster than TGMiner algorithms over all the target behaviors. For small behaviors, Table 1. All the training data are used in this experiment.

5.3 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.

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

Figure 13 demonstrates the runtime of all algorithms over small, medium, and large size behaviors as categorized in Table 1. All the training data are used in this experiment.

First, TGMiner consistently outperforms the baseline algorithms over all the target behaviors. For small behaviors, TGMiner can complete mining tasks within 6 seconds, which is up to 50 times faster than SupPrune. For medium and large behaviors, TGMiner can finish mining in 4 minutes and 26 minutes, respectively, and mine discriminative patterns with up to 40 edges. 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.

Second, compared with SubPrune and SupPrune, TGMiner obtains up to 50 and 4 times of speedup. SupPrune cannot mine all the discriminative patterns for medium and large behaviors within 2 days. We found most of the pruning opportunities come from subgraph pruning, while supergraph pruning brings additional performance improvement.

Third, compared with PruneGI, LinearScan, and PruneVF2, TGMiner obtains up to 6, 17, and 32 times of speedup, respectively. In particular, we found 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.

Figure 14 shows how the runtime of TGMiner varies as the amount of used training data is changed from 0 to 1. 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 to 1, 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.

Figure 12: Query accuracy with different query sizes

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.
6.2 Temporal Graph Management

6.1 Discriminative Graph Pattern Mining

6. RELATED WORK

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 Static 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.

6.1 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 [34, 41, 46].

In general, two directions have been investigated to speed up mining discriminative static graph patterns. First, many works [15, 16, 46] aim to find highly discriminative patterns early such that unpromising search branches can be pruned early. Second, approximate search has also been studied [16] to find good-enough graph patterns. In addition, there have been a few studies that mine graph patterns from a sequence of graph snapshots. In [32, 43], the desired patterns require identical activities happening within exact same time intervals. Ramu et al. [33] studied a special case where graph nodes are differentiated by their node ids.

Our work is different from existing works as follows. First, while it is difficult to extend existing mining techniques for static graphs to handle temporal graphs with multi-edges and temporal edge order, we propose a solution for temporal graphs. Second, compared with existing works on graph snapshots, we study more flexible temporal graph patterns. The graph patterns defined in [32, 33, 43] are too rigid to support security applications, where same behaviors may happen whenever a system is running and related nodes/edges could have diverse activities at different time.

6.2 Temporal Graph Management

Dynamics in temporal graphs have inspired many new graph queries. Recent study strives to develop cost-effective solutions that tackle query and management problems on temporal graphs.

One direction is to develop more efficient algorithms. Yang et al. [48] investigated time-dependent shortest-path queries. Wu et al. [44] studied minimum temporal path problems. In [38], Sricharan et al. developed efficient algorithms to detect abnormal edge weight changes.

Another direction is to enable efficient query processing by data management techniques. Novel graph indexes have been proposed to accelerate query processing for subgraph matching [53], reachability [52], and community searching [13]. Mondal et al. [27, 28] leveraged graph compression and partitioning to speed up query processing for neighborhood aggregation queries. Incremental computation can significantly reduce computation cost for many queries, including shortest-path queries [35], SimRank [50], and cluster searching [25]. In addition, Gao et al. [10] exploited distributed computation to monitor subgraph matching queries.

Data storage is another critical component in temporal graph management. Kineograph [6] addressed the consistency problem on storing graph snapshots, Grace [31] examined graph update transactions that incorporate dynamic changes into the latest graph snapshot, and Chronos [11] investigated the importance of temporal data locality for query processing. Khurana et al. [17] studied efficient ways to retrieve a certain or several graph snapshots. In [24], Labouseur et al. explored efficient storage and querying by leveraging commonalities among graph snapshots.

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.

6.3 Behavior-based Cybersecurity

Malware is a major threat to cybersecurity. To defend those attacks, one has to be aware of its existence.

Early works [19, 29] on malware detection extracted binary signatures (i.e., a sequence of bits) as features to detect malicious files. However, binary signature suffers poor adaptability [22, 40]. To address this problem, behavior-based approach are proposed.

Both high-level (e.g., application level) and low-level (e.g., binary instruction level) behaviors have been studied. For low-level, Christodorescu et al. identified low-level instruction semantics that commonly appear in malware [8], and Wichern et al. applied data-flow behaviors in programs to detect malware [45]. On the other hand, high-level behaviors also have been considered. Kirda et al. detected spyware by their suspicious behaviors on interacting with web browsers [22]. Moreover, common behaviors in malware were identified and grouped into clusters [4, 5, 30, 36].

In our work, we provide a foundation to support such behavioral queries in a large corpus of system-level logs, which enables retroactive inspection on historical events.

7. CONCLUSION

Security monitoring for computer systems generate huge amounts of log data that record the interaction of system entities (e.g., processes, files, and sockets). While these logs are promising for system administrators to query security-related behaviors, system administrators face a query formulation problem on how to compose queries for accessing data consisting of 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%).
8. REFERENCES

where in the following steps.

ble timestamps on edges. The transformation is conducted

g sequence of consecutive growth that grows

pruning are satisfied and none of patterns in

be the most discriminative, if the conditions in supergraph

score less than \( R_G \leq k \), which contradicts with the condition that none of patterns

Assume there exists a pattern

which means

is also a valid sequence of consecutive growth (with some

First, given the conditions

and

2.

1.

First, we prove the NP-hardness by reducing clique prob-

Proof.

Let \( g_1 \) and \( g_2 \) be temporal graph patterns where

is discovered before \( g_2 \) and they satisfy the conditions in

supergraph pruning.

First, given the conditions \( g_2 \geq g_1 \), and \( R(G,p,g_2) = R(G,p,g_1) \), and \( R(G_n,g_2) = R(G_n,g_1) \), we can derive the following properties,

1. \( \text{freq}(G_p,g_1) = \text{freq}(G_p,g_2) \);

2. \( \text{freq}(G_n,g_1) = \text{freq}(G_n,g_2) \).

Second, we prove none of patterns in \( g_2 \)'s branch will be the most discriminative, if the conditions in supergraph pruning are satisfied and none of patterns in \( g_1 \)'s branch is the most discriminative. Assume there exists a pattern \( g_2 \) whose discriminative score is no less than \( F^* \) and \( s \) is the sequence of consecutive growth that grows \( g_2 \) into \( g_2 \). Since \( g_1 \) and \( g_2 \) have the same number of nodes, we can safely claim \( s \) is also a valid sequence of consecutive growth (with some timestamp shift) that grows \( g_1 \) into \( g_1 \) by Proposition 1. By

\[
\text{freq}(G_p,g_1) = \text{freq}(G_p,g_1) \quad \text{and} \quad \text{freq}(G_p,g_2) = \text{freq}(G_p,g_1),
\]

we can infer \( \text{freq}(G_n,g_2) = \text{freq}(G_n,g_1) \). Similarly, we can infer \( \text{freq}(G_n,g_2) = \text{freq}(G_n,g_1) \). Thus, we have \( F(g_2') = F(g_1) \), which means \( g_1 \) is one of the most discriminative patterns which contradicts with the condition that none of patterns in \( g_1 \)'s branch is the most discriminative. Therefore, we can claim any patterns in \( g_2 \)'s branch will have discriminative score less than \( F^* \), and the branch can be safely pruned.

In sum, we have proved the correctness of this Proposition.

C. PROOF OF PROPOSITION 3

Proof.

We prove temporal subgraph test problem is NP-
hard and NP.

First, we prove the NP-hardness by reducing clique problem

[26] to temporal subgraph test problem. Given a static

graph \( G = (V,E) \) where \( V \) and \( E \) are node and edge sets,

the decision problem of clique problem is to decide whether

there exists a graph \( G' = (V',E') \) such that \( G' \) is a subgraph

of \( G \), \( G' \) is a clique, and \( |V'| \geq k \). We transform a static

graph \( G = (V,E) \) into a temporal graph \( G_t = (V_t,E_t,T) \),

where \( V_t \) is a node set, \( E \subseteq V_t \times V_t \times T \) is a temporal edge set,

and \( T \subseteq \mathbb{N} \) is a set of non-negative integers indicating possible

timestamps on edges. The transformation is conducted in the following steps.

1. We set \( V_t = V \), \( \forall u \in V_t \), a unique node ID is assigned,

denoted as \( id(v) \), such that \( \forall u, v \in V_t \), either \( id(u) > \)

\( id(v) \) or \( id(u) < id(v) \).

2. We transform the undirected edge set \( E \) into a directed

edge set \( E_d \) such that \( \forall (u,v) \in E \), \( (u,v) \in E_d \) and

(\( v,u \) \) \( \in E_d \).

3. We assign timestamps to edges in \( E_d \) and form a temporal

edge set \( E_t \) as follows:

\( \forall (u,v,t) \in E_t \), where the timestamp \( t = |V| \).

We assign timestamps to edges has two implications:

(1) \( \forall (u_1,v_1,t_1),(u_2,v_2,t_2) \in E_t \), if \( id(u_1) < id(u_2) \),

then \( t_1 < t_2 \); and (2) \( \forall (u,v,u_1,t_1),(u,v_2,t_2) \in E_t \), if

\( id(v_1) < id(v_2) \), then \( t_1 < t_2 \).

Following the same steps, a clique graph \( G_t' = (V_t',E_t') \)

with \( |V'| \geq k \) is transformed into a temporal graph \( G_t' = (V_t',E_t',T) \). It is not hard to see \( G \) contains a clique of size

no less than \( k \) if and only if \( G_t' \subseteq G_t \). Since the reduction

is performed in polynomial time, temporal subgraph test

problem is at least as hard as clique problem. Therefore, we

have proved it is NP-hard.

Second, temporal subgraph test problem is NP. Let \( G_1 \)

and \( G_2 \) be two temporal graphs. Given the two injective

functions \( f \) and \( \tau \) between \( G_1 \) and \( G_2 \), we can verify whether

\( G_1 \subseteq G_2 \) in polynomial time. Therefore, we have proved it is NP.

In sum, we have proved temporal subgraph test problem is NP-complete.

D. PRUNING FOR TEMPORAL SUB-

GRAPH TESTS

In order to further improve the speed, we adapt existing

pruning techniques for subsequence matching to temporal

subgraph tests.

Injective mapping enumeration is the potential bottleneck of this algorithm. As injective mapping enumeration is performed between node and enhanced node sequences without considering edge information, false mappings could be enumerated, and the number of false mappings directly affects the speed of temporal subgraph tests. Therefore, if false mappings can be detected earlier, we can prune the search on those false mappings, and improve the speed on node mapping enumeration.

To reduce the number of false node mappings, we consider the following techniques to detect false node mappings, and prune unpromising enumeration branches.

- **Label sequence test.** Label sequence test detects false node mappings without considering node IDs. After we replace the nodes in node, edge, and enhanced node sequences with node labels, the subsequence relation between node and enhanced node sequences and the subsequence relation between edge sequences should hold, if any temporal subgraph relation exists. Otherwise, we can safely claim no temporal subgraph relations exist.

- **Local information match.** Local information match considers local structure information to decide node mapping instead of node label only. The considered local structure information includes in/out node degree and in/out edge sequence.

- **Prefix pruning.** Any partial node mapping forms a prefix. If a prefix has been searched before, we can safely prune the search if we meet the prefix again.

E. DATASET DESCRIPTION
Training data. Training data is a set of temporal graphs representing a set of syscall logs generated from a closed environment (i.e., a server used to collect syscall logs with as little noise as possible), where security-related behaviors are performed.

First, we target at 12 behaviors as representatives for the basic security-related behaviors that have drawn attention in cybersecurity study [3, 5, 14, 49]. We execute the popular applications that perform these behaviors and collect the syscall logs as training data. In general, we consider security-related behaviors from five categories.

1. File compression/decompression includes behaviors that compressed/decompress files using build-in compression software. These behaviors usually happen when malware compresses private information files for stealing information, or decompresses source code files to build any malicious binaries. Example 1 shows one example scenario where the graph query includes the behavior of file compression. In this example, gzip2-based and gzip-based decompression behaviors are included.

2. Source code compilation includes behaviors where source code files are compiled into excitabile binaries. These behaviors are usually used by malware to build malicious binaries. In this study, we focus on the behaviors of gcc-based and g++-based compilations.

3. File download/upload are behaviors where system build-in software downloads files from remote servers, or uploads local files to remote servers. When malware wants to start new types of attacks, it first needs to conduct file download to obtain new source code files for building new malicious binaries. Meanwhile, malware usually steals private information by uploading private files to unknown remote servers. The behavior “(send-to-remote-server)” in Example 1 belongs to such category. In this study, the behaviors of ftp-based, scp-based, and wget-based download are considered.

4. Remote login involves behaviors where a user logs into a remote server. For attackers, these behaviors are usually the first step for breaking into a computer system. The behavior “sshd-login” in Example 1 falls into this category. We include the behaviors of ssh-based login (client side), sshd-based login (server side), and ftpd-based login (server side) in this study.

5. System software management includes behaviors where software applications are installed or updated by system build-in software. These behaviors are conducted by attackers when they want to install malware into a computer system. We investigate the behaviors of apt-get based update and apt-get based installation in this study.

For each target behavior, it is independently performed 100 times in a closed environment with the guarantee that no other target behaviors are performed, and accordingly, 100 temporal graphs (100 syscall logs) are collected. More details about the syscall logs for each behavior are shown in Table 1, where the average number of nodes/edges in each syscall log is reported.

Second, we collected background syscall logs from the same closed environment. In this process, the closed environment runs only the default applications, and none of the target behaviors are performed. First, we kept the closed environment running for one week and collected the syscall logs. Second, we randomly sampled 10,000 temporal graphs from the syscall logs of the closed environment. Each sampled temporal graph is based on a fixed time interval, which is the longest time duration observed among the 12 target behaviors. These 10,000 temporal graphs are used as the background data. The details of the background graph data are shown in Table 1, where the averages of nodes/edges in the syscall logs are reported.

Test data. Test data includes a large temporal graph representing a syscall log collected from another independent process. In an ordinary desktop, in addition to the default workload the desktop hosts, the following procedure is executed every one minute: (1) one of the target behaviors is randomly selected and performed; and (2) we record the time interval during which the selected behavior is performed as the ground truth. In this way, we collected the syscall logs of 7 days, which is a large temporal graph with 546,882 nodes, 6,571,180 edges. This temporal graph contains 10,000 behavior instances for the 12 target behaviors. Based on the patterns discovered by our algorithms, we formulate graph pattern queries to search those behavior instances from the test data.

Note that the closed environments for training data collection and test data collection are implemented in servers installed with Ubuntu 14.04. But the techniques discussed in this paper are also applicable to other operating systems.

F. DOMAIN KNOWLEDGE BASED RANKING FUNCTION

When TGMiner returns multiple discriminative temporal graph patterns that have the same highest discriminative score, the returned patterns are further ranked using the interest score of the nodes in the patterns. The interest score of a node is defined using domain knowledge, and the interest score for a pattern is the sum of the interest scores for all the nodes in the pattern. From all the returned temporal graph patterns, the top-5 temporal graph patterns are used to formulate graph pattern queries for the target behavior.

In this work, we define the interest score as follows. Given a node label \( l \),

\[
\text{interest}(l) = \frac{1}{\text{freq}(l)},
\]

where \( \text{freq}(l) \) is the number of temporal graphs in training data that contain this node label \( l \). Moreover, we keep a blacklist. For the node labels in the blacklist, such as “TmpFile”, “CacheFile”, “/proc/stat/*”, and many other node labels that provide little security-related information, we manually set their interest score as 0.

G. PROOF FOR LEMMA 5

Proof. We prove the sufficiency and necessity as follows.

First, we prove if \( g_1 \subseteq g_2 \), there exists nodeseq\((g_1) \subseteq \text{enhseq}(g_2) \) with an injective node mapping \( f' \) such that \( f'(\text{edgeseq}(g_1)) \subseteq \text{edgeseq}(g_2) \). Let \( m \) be the number of edges in \( g_1 \). We establish the proof by mathematical induction.

- Basis: we prove the statement holds for \( m = 1 \). If \( g_1 \subseteq g_2 \), we can find nodeseq\((g_1) \subseteq \text{enhseq}(g_2) \), since a source node always appears earlier than a destination node in an enhanced node sequence, and the resulting node mapping \( f' \) satisfies \( f'(\text{edgeseq}(g_1)) \subseteq \text{edgeseq}(g_2) \).
• Inductive step: we prove if the statement holds for 
\(m = k\), it holds for \(m = k + 1\). Assume the state-
ment holds for \(m = k\). There are three growth op-
tions to grow a \(k\)-edge temporal graph pattern to a
\((k + 1)\)-edge temporal graph pattern. Consider \(g_1\)
is grown from a \(k\)-edge pattern \(g'_1\). (1) For forward
growth, \(\text{nodeseq}(g_1)\) has one more node (a destina-
tion node) compared with \(\text{nodeseq}(g'_1)\). We can find
a corresponding node in \(\text{enhseq}(g_2)\) that maps the ad-
ditional node in \(\text{nodeseq}(g_1)\), since enhanced node se-
quencies always record destination nodes. Therefore,
an injective node mapping is formed. (2) For backward
growth, \(\text{nodeseq}(g_1)\) has one more node \(u_1\) (a source
node) compared with \(\text{nodeseq}(g'_1)\). Let \((v, w, t)\) be the
edge in \(g_2\) that matches the last edge in \(g'_1\). The node
\(u_2\) in \(g_2\) that maps \(u_1\) cannot be \(v\) or \(w\); therefore,
\(u_2\) will be recorded after \(w\) in \(\text{enhseq}(g_2)\) and match
\(u_1\). Therefore, an injective node mapping is formed.
(3) For inward growth, no more node will be added
 into \(\text{nodeseq}(g_1)\) compared with \(\text{nodeseq}(g'_1)\). There-
fore, an injective mapping is formed. For all the cases,
as long as an injective mapping is formed, we can find
\(f'(\text{edgeseq}(g_1)) \sqsubseteq \text{edgeseq}(g_2)\).

Therefore, the statement holds for \(m = k + 1\).

Since both the basis and the inductive step have been per-
formed, by mathematical induction, the sufficiency holds for
all \(m > 0\).

Second, we prove if \(\text{nodeseq}(g_1) \sqsubseteq \text{enhseq}(g_2)\) with an
injective node mapping \(f'\) such that \(f'(\text{edgeseq}(g_1)) \sqsubseteq
\text{edgeseq}(g_2)\), then \(g_1 \sqsubseteq_t g_2\). Through the node mapping
\(f'\), we can construct a temporal graph \(g'_2\) in \(g_2\) such that
\(g_1 \sqsubseteq_t g'_2\). Since \(g'_2 \sqsubseteq_t g_2\), we can conclude \(g_1 \sqsubseteq_t g_2\).

In sum, Lemma 5 is proved. □