2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

IPMES: A Tool for Incremental TTP Detection over the System Audit Event Stream

Hong-Wei Li
*CITI**
Academia Sinica
Taipei, Taiwan
g6_7893000@hotmail.com

Ping-Ting Liu
Department of Computer Science
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
xyfc128@gmail.com

Bo-Wei Lin
Department of Computer Science
National Yang Ming Chiao Tung University
Hsinchu, Taiwan
0800680274united@gmail.com

Yi-Chun Liao
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan
lyck92@gmail.com

Yennun Huang
*CITI**
Academia Sinica
Taipei, Taiwan
yennunhuang@citi.sinica.edu.tw

***Abstract*—Advanced persistent threat (APT) cyberattacks are serious threats to corporations and governments. The prolong dwell time associated with APTs significantly increase the difficulty on detecting them in provenance graphs. To reduce the detection complexity, some works have demonstrated the effectiveness of employing pattern matching on provenance graphs in conjunction with APT lifecycle models to pinpoint short-duration attack steps, also known as ”tactics, techniques, and procedures” (TTPs). However, when dealing with more complex TTPs, particularly those involving graph-based and partial ordering, few tools can incrementally and efficiently handle them. In this paper, we present IPMES[1], a tool that has been publicly released to address this gap. By leveraging specific optimizations, it provides efficient incremental matching for those TTPs, and can handle practical system audit event streams. Experiments conducted on synthetic and real-world data demonstrated the practical feasibility of IPMES in TTP detection.**

***Index Terms*—TTP detection, graph pattern matching, incremental pattern matching, system audit event**

I. Introduction

Advanced persistent threats (APTs) are one of the most serious challenges faced by enterprises and governments. Several APT lifecycle models have been developed to describe these prolonged and sophisticated cyberattacks. Mandiant [1] revealed several steps that are typically followed in these targeted attacks. The MITRE ATT&CK Framework [2], created by MITRE in 2013, further classified attacks based on their intention and technique as “tactics, techniques, and procedures” (TTPs). For example, an attacker exploits common vulnerabilities and exposures (CVEs) on external servers to gain initial access to a victim system through the network. Subsequently, the attacker penetrates the system by installing a backdoor shell, moving laterally through the network, and gaining further access. Eventually, an attacker achieves a goal, such as exfiltrating critical data or destroying a victim system. These models enable researchers to focus on detecting each short-duration attack step rather than viewing the whole APT.

*Research Center for Information Technology Innovation
[1]https://github.com/littleponywork/IPMES

In response to security incidents, analysts typically use system audit events and provenance graphs constructed from these events because of the abundant information contained in them. Each system audit event describes a subject (e.g., process) that performed some operation on an object (e.g., file) at some point in time. These events are usually recorded by the audit system in an operating system (OS) (such as the audit system events in Windows [3] or the audit log in Linux [4]). For advanced analysis, a provenance graph constructed from these events can provide a whole view of the events and the relations between involved subjects and objects. Some works [5]–[7] have demonstrated that using pattern matching on provenance graphs in combination with APT lifecycle models to identify short-duration attack steps (or TTPs) is a effective technique for APT detection.

As far as we know, there are very few tools specifically designed to incrementally detect TTP patterns from system audit event streams. This is because several challenges need to be overcome: (1) Generally, a TTP can be described by a graph-based pattern with partial ordering that captures the relationships among the programs and resources involved, as well as the temporal order in related events. Tools must be able to match such complex patterns. (2) Although incremental matching does not require storing the entire provenance graph, it does require maintaining all intermediate match states. Therefore, it necessitates efficient data structures and effective pruning mechanisms to reduce searching time and unnecessary matching states.

Apache FlinkCEP [8], Siddhi [9], and Esper [10] are existing complex event processing (CEP) tools that can incrementally analyze and correlate multiple streams of event data to detect complex patterns and relationships based on predefined rules. However, in our use experience, CEP tools are not suitable for matching graph-based patterns with partial ordering. For example, a CEP pattern usually represented by a sequence of event patterns (like $A \rightarrow B \rightarrow C$). When we want to describe partial ordering (i.e., $e_a \rightarrow e_b \rightarrow e_c$

and $e_a \rightarrow e_c \rightarrow e_b$ are both acceptable), we need write either extra rules ($A \rightarrow C \rightarrow B$) or a looser rule ($A \rightarrow (B\ or\ C) \rightarrow (B\ or\ C)$) with post filtering for incorrect instances ($e_a \rightarrow e_{b_1} \rightarrow e_{b_2}$). Obviously, both of these methods lack scalability. In addition, those CEP tools are designed for general purposes; they do not have specific optimizations for subgraph matching by exploiting graph-structured features, such as shared entity relation. This limitation becomes more pronounced in audit event stream with more complex patterns.

[Contributions] To address the above challenges, we present IPMES, a specialized tool for TTP detection, catering to the needs of cybersecurity professionals, incident responders, and organizations aiming to enhance their real-time threat detection capabilities over system audit event streams. Its features includes following: (1) Adopting the idea from [11], [12], IPMES first divides a graph-based pattern with partial ordering into multiple sequence patterns with total ordering, and then joins the results, which can effectively increase efficiency. (2) IPMES incorporates specific optimizations such as lazy windowing and arranging join order to significantly enhance efficiency. (3) IPMES can handle events with same timestamps and interval timestamps (i.e., $\langle start, end \rangle$) which are common in practical system audit event streams. IPMES is evaluated on the synthetic and real-world data, and the experiment results show its feasibility in practical TTP detection.

II. Related Work

A. APT detection

Based on the APT lifecycle models, Holmes [5] and RapSheet [13] develop effective misuse detection architectures for APT detection by matching instances with smaller patterns (TTPs or indicator of compromises [IOCs]) on provenance graphs and by correlating the matched instances to construct a complete APT detection. Each of their patterns for TTPs or IOCs constitutes only a single event rather than multiple events. Their matchings are very efficient but could lead to excessive false positives. Holmes leverages a normal model learned from benign data to reduce false positives. RapSheet reduces false positives by checking if an alert event is causally dependent on the others through the construction of provenance subgraphs. In contrast, IPMES uses complex patterns to model attacks more precisely to reduce false positives.

SIGMA [14] is a universal security rule format used to describe security events and threat detection rules. Its aim is to make rules portable, allowing them to be utilized across different existing security information and event management systems such as IBM Security QRadar [15] and Splunk Enterprise Security [16]. However, due to rule simplicity, SIGMA may not intuitively describe complex attack patterns or efficiently search for them. On the other hand, IPMES excels in efficiently matching more intricate and complex attack patterns.

As deep learning has demonstrated remarkable success in various domains, DeepHunter [17] and ProvG-Searcher [18] tackle the challenge of efficient provenance graph search by employing graph-based neural networks to embed both provenance and pattern graphs. Subsequently, match results are determined based on these embeddings. However, due to the inherent nature of neural networks, the matches are inexact (allowing slight discrepancies between searched results and queries) and require recalculating embeddings when the provenance graph changes. In contrast, IPMES provides exact matching and efficiently handles dynamic provenance graphs without the need for repeated searches.

TABLE I: Comparison table for APT detection

Works	Attack Pattern Constitution	Match Class	NTR	Detection Efficiency	ESS
RapSheet [13]	single event or a few events	exact	○	high	×
Holmes [5]	single event or short path	exact	○	high	○
SIGMA [14]	single event or a high events	exact	○	high	○
ProvG-Searcher [18], DeepHunter [17]	graph with time constraints	inexact	×	depend on graph size	×
Atlas [19]	not pattern matching based	×	×	high	×
Anubis [20]	not pattern matching based	×	×	high	○
IPMES	graph with time constraints	exact	○	depend on attack intensity	○

Note: NTR means No training required. ESS means event stream support.

Atlas [19], and Anubis [20] utilize a sequence-based deep learning model to detect APT attacks. Their methods mainly extract small subgraphs containing related nodes from the snapshots of provenance graphs and transform the subgraphs into sequences for training and detection. Although these systems can efficiently detect different APTs, they require huge volumes of labeled data for training and retraining, particularly when identifying new APTs or shifts in normal behaviors. On the other hand, IPMES does not require training data and only updates patterns upon discovering new APTs. Comparisons of the above works are shown in Table I.

B. Graph pattern matching

While subgraph isomorphism, a widely used for graph pattern matching, is known to be NP-complete and overly restrictive for identifying meaningful instances in social network applications, alternative low-complexity graph pattern matching semantics have been proposed such as graph simulation [21], [22], strong simulation [23], and degree-preserving dual simulation [24]. These pattern matching methods do not have strong graph-structured relations between nodes/edges like subgraph isomorphism, and can be done in $O(n^3)$. IPMES is compatible with the pattern matching semantics by modifying the corresponding relation checking. Avoiding using subgraph isomorphism, Song et al. [25] proposed a Node-Neighbor Tree structure to filter out false candidates over graph streams efficiently. Their work focused on answering whether the current graph exists an instance of a graph pattern, while IPMES can answer all instances in the current graph.

The works [11], [12] achieve incremental matching with time ordering by dividing a query graph (i.e., behavioral pattern) into multiple subpatterns and recording all intermediate match states in the proposed data structure (Match-Store tree). Its evaluations use real network traffic data and a synthetic social stream benchmark, while IPMES's evaluations use real and synthetic system audit logs, which have more complexity and higher data rate. In addition, those works assume that timestamps in the data stream are strictly increasing, whereas IPMES can accommodate event streams with non-strictly increasing timestamps and interval timestamps like $\langle from, to\rangle$.

III. IPMES

The core concept of IPMES involves decomposing a target behavioral pattern into multiple subpatterns, matching events against these subpatterns, and then combining the match results of these subpatterns to generate complete instances. Figure 1 illustrates an overview of IPMES. Our implementation adopts a layered structure, tailored for efficient streaming data processing. Each layer processes the output of its preceding layer as input. IPMES consists of three layers: **Matching Layer**, **Composition Layer**, and **Join Layer**. A target behavioral pattern is decomposed in **Prepossessing**. Upon reading an event, **Matching Layer** splits it based on its interval timestamps, and reorders events according to their matched event patterns of the subpatterns. Then, **Composition Layer** uses events to match against subpatterns. Finally, **Join Layer** collects all match results of all subpatterns and merges them into complete instances as outputs.

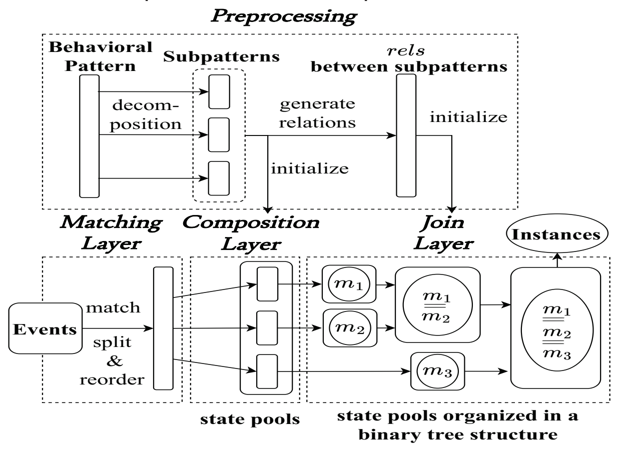

Fig. 1: IPMES flowchart

A. Preliminaries

A system audit event describes a subject performing some operation on an object at some point in time; an event is the minimum unit of record in an audit system. To accommodate various system audit events, each event evt that entails a subject sub and an object obj from timestamp t_s to timestamp t_e, and identified by a unique serial ID $evtid$, is represented as $(t_s, t_e, sig_{(evt,sub,obj)}, evt_{id}, sub_{id}, obj_{id})$. A signature $sig_{(evt,sub,obj)}$ is a string containing those important attributes in evt, sub and obj such as event operation label, subject/object type and name, etc., and is used for pattern matching. The interval timestamp definition allows an audit system to merge the consecutive events with the same operations (no other operations in between) for data reduction [26]. In cases where this aggregation is not applicable, the timestamp t_s and t_e are just set to the same value.

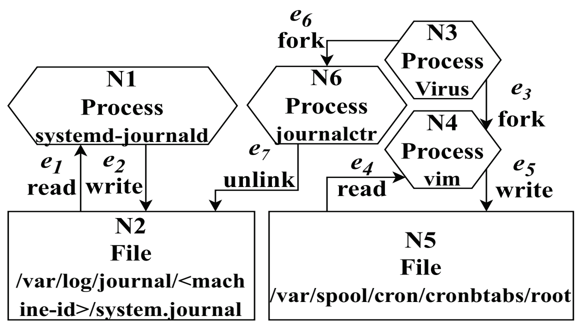

Fig. 2: Provenance graph example

TABLE II: Simplified view of the original events in Figure 2

t_s	t_e	$sig_{(evt,sub,obj)}$	evt_{id}	sub_{id}	obj_{id}
0	10	read# File::path::/var/log/journal/system.journal# Process::name::systemd-journald	e_1	N2	N1
5	15	write# Process::name::systemd-journald# File::path::/var/log/journal/system.journal	e_2	N1	N2
7	7	fork# Process::name::virus# Process::name::vim#	e_3	N3	N4
8	11	read# File::path::/var/spool/cron/crontabs/root# Process::name::vim#	e_4	N5	N4
9	11	write# Process::name::vim# File::path::/var/spool/cron/crontabs/root	e_5	N4	N5
11	12	fork# Process::name::virus# Process::name::journalctr	e_6	N3	N6
11	12	unlink# Process::name::journalctr# File::path::/var/log/journal/system.journal	e_7	N6	N2

An event stream is defined as an arbitrary or countably infinite length sequence of events ordered by their timestamps (with t_s taking precedence, and in the case of equality, t_e follows), i.e., $\langle e_1, e_2, ..., e_n\rangle$ or $\langle e_1, e_2, ...\rangle$ where $e_i.t_s < e_j.t_s$ or $e_i.t_e \leq e_j.t_e$ if $e_i.t_s = e_j.s$ for all $i < j$.

A provenance graph is defined as a directed multigraph, and can be trivially constructed from a set of events (events as edges and involved entities as nodes). It would provide a comprehensive overview of the events, including the relationships between them. Figure 2 presents an example of an provenance graph recording both a normal process, "systemd-journald", and a malicious process, "virus". "Systemd-journald" frequently reads and writes a system journal file to record system information. Meanwhile, "virus" first uses "vim" to modify the system crontab file and schedule a task for itself to run persistently within the system. Subsequently, "virus" deletes the system journal file through "journalctr". Table II presents a simplified view of the original events.

A behavioral pattern (or a subpattern) is defined as a tuple of an event pattern set and a set of relations between them, represented by $(EP, rels)$. Each event pattern in EP is similar to an event, but uses a RegEx instead of a string in $sig_{(evt,sub,obj)}$ field. An event is considered a matched instance of a event pattern, or simply an instance, representing it matches the pattern without violating any specified relations. This concept is similar to an instance of a behavioral pattern but pertains to a set of events. In this study, we utilize the concept of subgraph isomorphism with time constraints to establish relations. The motivation is that any subsequence within an event stream is inherently a subgraph within the associated provenance graph. Moreover, Introducing time constraints explicitly enhances the expressiveness of event ordering within this subgraph. The relations $rels$ include:

R1. Temporal Relation (with a partially ordered set of EP): The timestamps of the corresponding matched events do not violate the ordering.

R2. Shared Entity Relation (with a set of relationships between the entities of EP): Each relationship specifies that the entity's ID of the corresponding matched event of an event pattern is the same as the one of another event pattern.

R3. Event Unique Relation: All matched event IDs are unique.

R4. Entity Unique Relation: All entity IDs in matched events are unique unless they have a relationship in the Shared Entity Relation.

Similar to the construction of a provenance graph from events, EP with relations R2, R3, and R4 can be used to construct a pattern graph that is identical to a provenance graph except that the signature is replaced with a RegEx. Figure 3 illustrates a behavioral pattern example, depicting an arbitrary process characterized by two malicious behaviors. The first is T1070-Indicator Removal [27], involving the deletion of log files generated within systems to eliminate evidence of its presence. The second is T1053-Scheduled Task/Job [28], exploiting task scheduling functionality to facilitate the initial or recurring execution of malicious code. The behavioral pattern contains five event patterns, and their relations R2, R3, and R4 are entailed in the pattern graph. In short, a behavioral pattern describes a bundle of events with patterns for specific attributes in edges/subjects/objects and their occurrence order to express a program behavior.

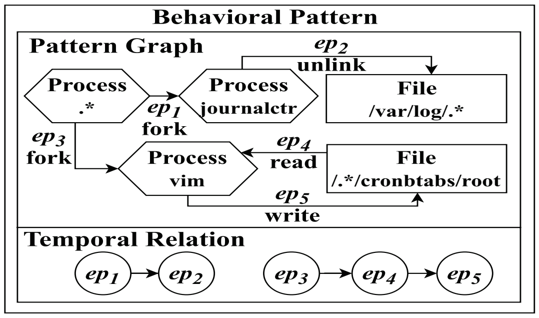

Fig. 3: Behavioral pattern example

[Problem Definition] Given a behavioral pattern P and a event stream ES, the aim is to find all instances of P in the provenance graph G_{ES} from ES. Due to the adoption of the subgraph isomorphism with time constraints to define relations in behavioral patterns, this problem has been proven to be #P-complete [29], [30].

To describe a match state in the following proposed algorithm, an intermediate match state (w.r.t. a behavioral pattern P or a subpattern), or "state" for convenience, is defined as (E, f_P) where E a set of already matched events and f_P is a one-to-one function that maps each event in E to the corresponding event pattern in P. Updating an intermediate match state s with an event e and an event pattern e_P of P is to add e to $s.E$ (i.e., $s.E \cup \{e\}$) and update the mapping (i.e., $s.f_P(e) = e_P$) without violating any relations in P. For convenience, $m1 \| m2$ represents a merged state updated from all events of state $m1$ and $m2$. An intermediate match state is complete iff f_P is a bijective function which means each event pattern matches a unique event. Obviously, a complete intermediate match state of a pattern corresponds to an instance of the pattern.

B. Preprocessing

Given a behavioral pattern $(EP, rels)$, IPMES adopts the decomposition algorithm proposed by Li et al. [11] to decompose the pattern into subpatterns. The algorithm guarantees that all event patterns of each subpatterns follow a total ordering and have shared entity relationships between adjacent event patterns. For those remaining relationships that cross two distinct subpatterns, we record them and will check them in **Join Layer**. Figure 4 presents the subpatterns, sp_1 and sp_2, decomposed from Figure 3. Their total orderings are respectively $ep_1 \rightarrow ep_2$ and $ep_3 \rightarrow ep_4 \rightarrow ep_5$, and the shared entity relationship ser does not belong to any subpatterns. For convenience, we use $e_k \sim ep_j@sp_i$ to represent a event e_k match an event pattern ep_j of a subpattern sp_i.

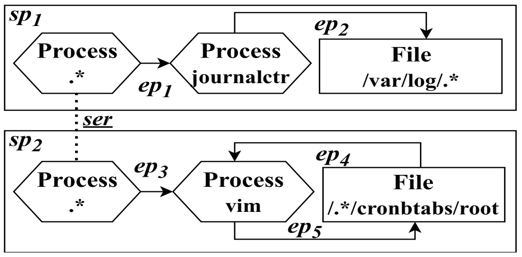

Fig. 4: Subpattern example decomposed from Figure 3, where ser represents a shared entity relationship

C. Matching Layer

Matching Layer handles issues regarding timestamps and reorders events according to their matched event patterns. Recall that an event timestamp is represented as an interval. To match accurately, we must consider all possible temporal relationships between events. For instance, consider e_1 and e_2 in Table II. Both $e_1 \rightarrow e_2$ (for $6 \rightarrow 13$) and $e_2 \rightarrow e_1$ (for $7 \rightarrow 9$) should be taken into consideration in temporal relation checking. To simplify the implementation of timestamp interval comparisons, those events with $t_s \neq t_e$ are split into two distinct events with same field values except for timestamps (e.g., $split(e_1) = \langle e_{1s}, e_{1e} \rangle$ where $e_{1s}.t_s = e_{1s}.t_e = 0$ and $e_{1e}.t_s = e_{1e}.t_e = 10$ in the above case). This technique preserves possible temporal relationships between events.

Now, all events have been split ($e.t_s = e.t_e$ for all events). Depending on the timestamp granularity of input event streams, some events may have identical timestamps (e.g., e_6 and e_7 in Table II). In this case, we must also consider all possible temporal relationships that comply with subpatterns, since no more fine-grained information is logged. **Matching Layer** reorders these events according to their matched event patterns, aligning them with the total orderings of the subpatterns. In Figure 4 and Table II, consider an input event sequence with identical timestamps 11: $\langle e_{7s}, e_{5e}, e_{4e}, e_{6s}\rangle$. The sequence will be matched and rearranged as $\langle e_{6s} \sim ep_1@sp_1, e_{7s} \sim ep_2@sp_1, e_{4e} \sim ep_4@sp_2, e_{5e} \sim ep_5@sp_2, \rangle$.

D. Composition Layer

Composition Layer mainly updates stored intermediate match states of subpatterns, and sends complete states to **Join Layer**. Algorithm 1 illustrates the procedure of **Composition Layer**. To efficiently update states, **Composition Layer** maintains an array of pools to categorize states for each subpattern, Here, $pools[i][j]$ denotes the pool that stores states waiting for an event matched by $ep_j@sp_i$.

Upon receiving a event $e_k \sim ep_j@sp_i$ from **Matching Layer**, **Composition Layer** tries to update s with e_k for all state s in $pools[i][j]$ (at Line 2–9). To eliminate unnecessary states, **Composition Layer** employs a lazy window mechanism for each pools: Instead of continually checking all states for expiration in all pools, the expired states, whose event-timestamp span exceeds the predefined window size, are only removed when being traversed in the pools. If a newly updated intermediate match state is complete (at Line 12–16), it will be sent to **Join Layer**; otherwise, the state will be added to the next pool of the same subpattern. Figure 5 illustrates an example for **Composition Layer** in handling Table II and 4. At $t = 7$, the empty state (i.e., $(\emptyset, \emptyset \rightarrow \emptyset)$) in $pools[2][1]$ is updated with $e_3 \sim ep_3@sp_2$ and subsequently added to $pools[2][2]$. Then, at $t = 8$, similar operations occur, where a new state is added to $pools[2][3]$ by updating a state in $pools[2][2]$ with $e_{4s} \sim ep_4@sp_2$. At $t = 9$, a complete state is formed by updating a state in $pools[2][3]$ with $e_{5s} \sim ep_5@sp_2$, and is then sent to **Join Layer**.

E. Join Layer

Join Layer joins complete states of subpatterns into complete states of the target behavioral pattern (i.e., instances). Algorithm 2 illustrates the procedure of **Join Layer**. Because the event splitting procedure in **Matching Layer** may cause identical instances, **Join Layer** uses a hashset to ensure instance uniqueness. In addition, **Join Layer** maintains a binary-tree structured pool T to store states. All leaf nodes of T (like m_i in Figure 1) stores complete states of subpatterns (sp_i), while all non-leaf nodes (like $m_1 \| m_2$ in Figure 1) stores joined states from its two descendants. It is clear that any states in the root node are complete since each event pattern in the behavioral pattern has a corresponding matched event, and all relation checks are passed.

Whenever a node of T receives a new state, **Join Layer** tries to join the state with all states in the node's sibling (at Line 10–17). Newly joined states are then pushed to the node's

Algorithm 1: Composition Layer

```
Input: an event from Matching Layer
       ($evt \sim ep_j@sp_i$), subpatterns $SP$, window size
       $ws$, Join Layer $JoinLayer$
Output: update states in the pools, and send complete
        states of subpatterns to Join Layer
Initialize: $pools[i][j] \leftarrow \{\}, \forall sp_i \in SP, \forall ep_j \in sp_i$
Initialize: $pools[i][1].add((\emptyset, \emptyset \rightarrow \emptyset)), \forall sp_i \in SP$
1  $newStates \leftarrow \{\}$
2  for state $s \in pools[i][j]$ do
3      /* lazy window mechanism */
4      if $s.isExpired(ws)$ then
5          $pools[i][j].remove(s)$
6      /* check if $s$ can be updated with $evt$ */
7      if $checkRelation(s, evt, SP.rels)$ then
8          /* a new match state is generated */
9          $newStates.add(s \| \{evt \sim ep_j\})$
10 if $j == sp_i.|EP| + 1$ then
11     /* new states are complete for $sp_i$ */
12     for state $s \in newStates$ do
13         $JoinLayer.receive(s)$
14 else
15     /* new states are incomplete and lack an event
          matched by $ep_{j+1}@sp_i$ */
16     $pool[i][j+1].addAll(newStates)$
```

parent (at Line 19–22). **Join Layer** repeats this process in a bottom-up fashion until no new states are joined or the root has been reached. Figure 5 illustrates an example for **Join Layer** in handling Table II and 4. At $t = 11^{(1)}$, the leaf node corresponds to sp_1 receives a state ($N3 \xrightarrow{e_{6s}} N6 \xrightarrow{e_{7s}} N2$) from **Composition Layer**. Since the node's sibling is not empty, **Join Layer** tries to join the state with states in the sibling, which results in pushing a complete state to the root.

Join Layer incorporates the following optimizations to enhance efficiency. (1) **Join Layer** also employs the lazy window mechanism for each node to eliminate unnecessary states. (2) A node corresponding to a subpattern with more event patterns are placed in a deeper leaf nodes of T. This heuristic assumes that larger subpatterns have fewer corresponding complete states. (3) Since the joining order is predetermined, **Join Layer** can check only necessary relations during joining.

IV. Evaluation

Various experiments were performed to evaluate the IPMES to determine the following:

Q1. Efficiency: Whether **Composition Layer** and **Join Layer** design is efficient in TTP detection.
Q2. Feasibility: Whether applying IPMES for TTP detection is feasible.

For generating experiment datasets, we used a simulated scenario and a real-world scenario. In the simulated scenario, we first referenced the APT3 (threat actor) emulation plan from Mitre Att&ck [31] to implement an APT attack which primarily exploits CVE-2021-37678 [32] to implant a malware for malicious activities and entirely covers 19 different TTPs.

Algorithm 2: Join Layer

```
Input: a complete state $m$ of a subpattern $sp_i$, a
       behavioral pattern $P$, window size $ws$
Output: complete states of $P$ $ins$
Initialize: $node.pool \leftarrow \{\}$, $\forall node \in$ the tree $T$
Initialize: $ins \leftarrow$ new HashSet()
1  /* locate the node of $sp_i$ in $T$ */
2  $node \leftarrow T.getNode(sp_i)$
3  $node.pool.add(m)$
4  /* $states$ holds the states newly arrived at $node$ */
5  $states \leftarrow \{m\}$
6  /* do a bottom-up join */
7  while $node \neq T.root$ do
8      $newJoinedStates \leftarrow \{\}$
9      /* traverse pools in $node$ and its sibling and try to
        join states */
10     for $s_1 \in states, s_2 \in node.sibling.pool$ do
11         /* lazy window mechanism */
12         if $s_2.isExpired(ws)$ then
13             $node.sibling.pool.remove(s_2)$
14         else
15             if $checkRelations(s_1, s_2)$ then
16                 // a new state is joined from $s_1$ and $s_2$
17                 $newJoinedStates.add(s_1 \| s_2)$
18     if !$newJoinedStates.isEmpty()$ then
19         /* newly joined state exists; switch to the
            node's parent for further joining */
20         $node.parent.pool.addAll(newJoinedStates)$
21         $states \leftarrow newJoinedStates$
22         $node \leftarrow node.parent$
23     else
24         break
25 if $node = T.root$ then
26     $ins.addAll(states)$
27 return $ins$
```

Then, in our controlled environment, we simulated three benign users concurrently and continually executing normal operations as benign workloads, and one attacker periodically executing the APT attack per hour. System audit events were logged by SPADE [33], [34], and eventually, three synthetic datasets, *Attack* (containing only attacks), *Benign* (containing only benign workloads), and *Mix* (containing both benign workloads and attacks), were generated based on this scenario.

In real-world scenario, we used DARPA Trace [35] as a data source. According to their documents, we extracted the logs for 4 intervals as 4 datasets, one of which (*DD1*) does not contain any attacks, and three of which (*DD2*, *DD3* and *DD4*) cover four APT attacks (Firefox Backdoor, Browser Extension, Phishing Email, and Pine Backdoor). Information on the above datasets is presented in Table III. All system audit events for all datasets were preprocessed using SPADE and Neo4j [36]; the events were then output in our custom JSON format. In preprocessing, only consecutive events with the same operation and the same subject and object are aggregated.

The generation of precise behavioral patterns for attack behaviors without covering normal behaviors is beyond the scope of this paper. For experiments, behavioral patterns were selected by inspecting the provenance graphs constructed from the datasets and referencing related works like [37]. First, some events were manually extracted as event patterns that were considered to belong to a behavioral pattern from the datasets. In the signature of the event patterns, non-general values (such as the object hash, UUID and timestamp) were removed, and some values (such as non-utility program names, IPs and port numbers, and file or directory names) were replaced with wildcard string (i.e., ".*") in RegEx for pattern generality. The temporal relation of each event pattern was defined based on the original event timestamps and human knowledge (this relation may be a partial ordering if some events have the same timestamp). The other relations can be directly retrieved from the provenance graph.

For the DARPA Trace datasets, five behavioral patterns in APT-level (containing multiple TTPs) were extracted and denoted as **DP1**–**DP5**. These patterns correspond to four APT attacks documented in the dataset, each characterized by a duration lasting 92–996 seconds, involving 9–20 entities and 15–30 events. For the synthetic datasets, twelve behavioral patterns in TTP-level were extracted and denoted as **SP1**–

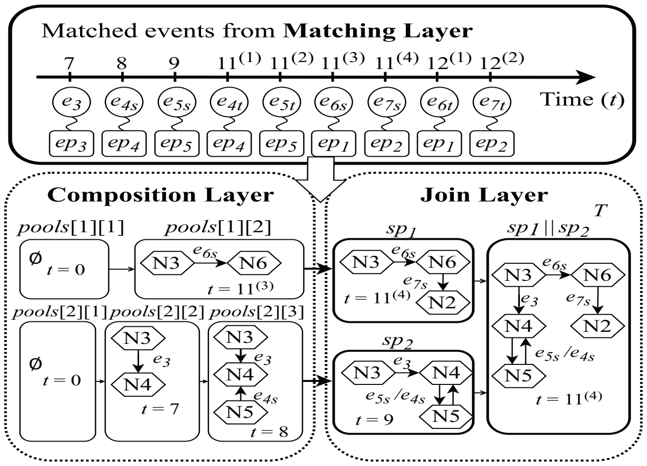

Fig. 5: An example for **Composition Layer** and **Join Layer** in handling Figure 3, Figure 4, and Table II

TABLE III: Dataset characteristics

Scenario	Dataset Name	Duration	# of nodes (entities)	# of edges (events)
Simulated	*Benign*	12 hr.	688,016	771,417
	Attack	12 hr.	1,793,045	2,107,866
	Mix	12 hr.	2,393,438	2,885,867
Real-world (DARPA Trace)	*DD1* 2018/04/08 (19:00 – 23:00)	4 hr.	3,764,417	6,609,005
	DD2 2018/04/10 (09:40 – 16:00)	6.33 hr.	10,118,397	7,062,697
	DD3 2018/04/13 (12:00 – 17:00)	5 hr.	12,463,586	7,692,127
	DD4 2018/04/13 (14:00 – 19:00)	5 hr.	17,618,165	10,851,966

SP12, each characterized by a duration lasting 0.02–35 seconds, involving 5–26 entities and 4–36 events.

All experiments were run on an Ubuntu 18.04 host equipped with four Intel Xeon Platinum 8280 Processors at 2.70 GHz, 1.47 TB RAM, and 21 TB disk storage. CPU time was measured using the “time” utility in Linux, while memory usage data was obtained from the JVM.

A. *Matching efficiency*

For answering the Q1. Efficiency, we used two baseline setting, IPMES with naive join and Siddhi. IPMES with naive join setting is similar to IPMES but replaces the binary tree structure with a large array directly storing all states in **Join Layer**. This implies that the naive join setting does not take into account the join order. In the Siddhi setting, **Composition Layer** is replaced with Siddhi [9] CEP tool for matching subpatterns because Siddhi is proved its practicality with large-scale provenance data in the work [7]. Table IV presents the matching and performance results for the synthetic datasets with IPMES, IPMES with naive join, Siddhi (all window sizes are 1800 seconds). Generally, the CPU time of IPMES is less than $\text{IPMES}_{\text{naive}}$(1.1x–10.1x) and Siddhi(2.1x–4.3x) in average. The peak heap memory of IPMES is also less than $\text{IPMES}_{\text{naive}}$(1.0x–1.2x) and Siddhi(1.6x–1.8x) in average. This indicates that, without the consideration of join order, $\text{IPMES}_{\text{naive}}$ generates more states, resulting in an increased cost in traversing states. On the other hand, Siddhi exhibits lower efficiency in matching subpatterns as it employs an abstract syntax tree for general relation checkings. This approach determines which variables' values to obtain during runtime. In contrast, IPMES achieves relation checkings by explicitly specifying the required variables in the code.

Table V presents the matching and performance results for the DARPA Trace datasets with IPMES, IPMES with naive join, Siddhi (all window sizes are 1000 seconds). In most cases, the CPU time of IPMES is close to $\text{IPMES}_{\text{naive}}$ except for **DP5** in *DD4*. This is because the number of states are a few in the datasets such that the benefits from the join order is not noticeable, while **DP5** has up to 873,785 states at peak and 21,453,120 instances in *DD4*. The CPU time of Siddhi is very larger than IPMES (up to 936x in average). The explanation about IPMES' high efficiency is provided in the previous paragraph, and this issue becomes more pronounced in more complex patterns.

B. *Window size*

To investigate window size impacts on efficiency and sensitivity, we selected **SP7** and the *Mix* dataset for experiments on window size. The reason is that the previous experiments revealed that sufficient instances of the behavioral pattern in the dataset and a larger pool size in executing IPMES. Moreover, despite DARPA datasets containing numerous instances of patterns, all these instances belong to a single attack.

For efficiency, Figure 6 presents the peak pool size and required CPU time for various window size. The CPU time grows slowly under window size 3200 seconds, and then grows fast. This is because the attacks executed once per hour in our simulated scenario. A larger window size exceeding 3600 seconds causes the pool storing states associated with different attack instances, thereby reducing the efficiency in join. Note that the peak pool size of window size 2 second is larger than ones of window size 4, 8 and 20 seconds. This is because of the lazy window mechanism.

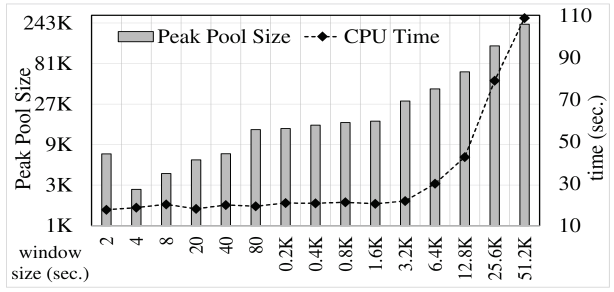

Fig. 6: Peak pool size and required CPU time for various window size with **SP7** on *Mix* dataset

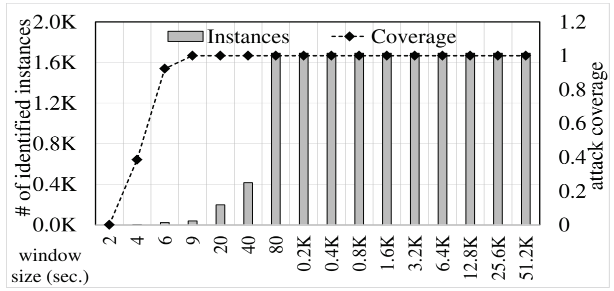

Fig. 7: Number of identified instances and coverage for various window size with **SP7** on *Mix* dataset

For sensibility, Figure 7 presents the number of identified instances and coverage for various window size. The coverage means the ratio of the number of discovering an attack by IPMES (identify at least one instance in one attack) and the number of all attacks (= 13 times in this dataset). A window size exceeding 80 seconds can not find more instances, and a window size under 9 seconds would lose some coverage. This means the **SP7** attack can last up to 80 seconds, and revealing its behavior at a minimum of 9 seconds. Combined the efficiency results in Figure 6, in practice guideline, we suggest the proper window size should be set in accordance with the threat reports or threat models at hand for better detection efficiency and sensibility. Note that window sizes within the intervals of identical attacks are also considered acceptable without significantly compromising efficiency if the attacks are sparse. Intense attacks will be discussed in Sec. V.

About the Q2. Feasibility, In simulated datasets, where the log durations were around 12 hours, the average CPU time per pattern was 12–20 seconds. In the DARPA Trace datasets, where the log durations were around 5 hours, the average CPU time per pattern was 0.5–14 minutes. These results imply that IPMES can feasibly handle around 20–3600 TTP patterns for practical online detection without any advanced

TABLE IV: Matching and performance results for synthetic datasets

Pattern No.	CPU Time (ratio r.s.t to IPMES) IPMES / IPMES$_{\text{naive}}$ / Siddhi			Peak Heap Memory (ratio w.r.t IPMES) IPMES / IPMES$_{\text{naive}}$ / Siddhi		
	Attack	*Mix*	*Benign*	*Attack*	*Mix*	*Benign*
SP1	1.0x / 1.0x / 2.0x	1.0x / 1.1x / 1.7x	1.0x / 1.0x / 2.1x	1.0x / 0.8x / 2.1x	1.0x / 1.0x / 1.6x	1.0x / 1.0x / 1.5x
SP2	1.0x / 1.0x / 1.6x	1.0x / 1.1x / 1.7x	1.0x / 1.0x / 1.5x	1.0x / 1.0x / 1.3x	1.0x / 1.0x / 1.4x	1.0x / 0.8x / 1.2x
SP3	1.0x / 4.5x / 1.9x	1.0x / 3.7x / 2.0x	1.0x / 1.0x / 1.9x	1.0x / 1.1x / 2.3x	1.0x / 1.3x / 1.6x	1.0x / 1.0 / 2.1x
SP4	1.0x / 1.2x / 1.7x	1.0x / 0.9x / 1.6x	1.0x / 1.0x / 1.6x	1.0x / 1.0x / 1.6x	1.0x / 0.8x / 2.1x	1.0x / 1.0x / 1.4x
SP5	1.0x / 0.9x / 2.1x	1.0x / 1.0x / 4.8x	1.0x / 1.1x / 40.0x	1.0x / 1.1x / 1.9x	1.0x / 1.4x / 2.0x	1.0x / 1.0x / 3.5x
SP6	1.0x / 1.2x / 1.6x	1.0x / 1.1x / 1.8x	1.0x / 1.1x / 1.9x	1.0x / 1.7x / 1.8x	1.0x / 0.8x / 1.5x	1.0x / 0.8x / 1.1x
SP7	1.0x / 5.6x / 1.5x	1.0x / 69.0x / 1.7x	1.0x / 0.9x / 1.5x	1.0x / 1.8x / 1.2x	1.0x / 3.7x / 1.6x	1.0x / 1.0x / 1.2x
SP8	1.0x / 1.6x / 1.5x	1.0x / 3.5x / 1.5x	1.0x / 5.6x / 1.7x	1.0x / 1.1x / 2.3x	1.0x / 0.8x / 2.0x	1.0x / 1.0x / 1.2x
SP9	1.0x / 1.1x / 1.6x	1.0x / 1.0x / 3.0x	1.0x / 1.1x / 1.8x	1.0x / 1.0x / 1.5x	1.0x / 0.7x / 1.5x	1.0x / 1.0x / 1.4x
SP10	1.0x / 1.1x / 1.3x	1.0x / 1.0x / 1.6x	1.0x / 1.0x / 1.6x	1.0x / 1.0x / 2.1x	1.0x / 0.8x / 1.5x	1.0x / 1.0x / 1.7x
SP11	1.0x / 1.0x / 1.5x	1.0x / 1.0x / 1.7x	1.0x / 1.0x / 1.6x	1.0x / 1.0x / 1.5x	1.0x / 1.0x / 1.6x	1.0x / 1.2x / 1.5x
SP12	1.0x / 1.0x / 1.4x	1.0x / 1.1x / 2.0x	1.0x / 1.0x / 1.5x	1.0x / 1.0x / 1.7x	1.0x / 1.0x / 1.8x	1.0x / 1.0x / 1.2x
Average	1.0x / 2.4x / 2.1x	1.0x / 10.1x / 2.8x	1.0x / 1.1x / 4.3x	1.0x / 1.2x / 1.8x	1.0x / 1.2x / 1.7x	1.0x / 1.0 / 1.6x

Note: A cell with a filled background indicates that the column dataset contains instances of the row pattern. The average row is calculated using the average CPU time rather than the average of the ratios.

TABLE V: Matching and performance results for DARPA Trace datasets

Pattern No.	CPU Time (ratio w.r.t IPMES) IPMES / IPMES$_{\text{naive}}$ / Siddhi				Peak Heap Memory (ratio w.r.t IPMES) IPMES / IPMES$_{\text{naive}}$ / Siddhi			
	DD2	*DD3*	*DD4*	*DD1*	*DD2*	*DD3*	*DD4*	*DD1*
DP1	1.0x | 1.2x | 1039x	1.0x | 1.1x | 2390x	1.0x | 1.1x | 2696x	1.0x | 1.2x | 1965x	1.0x | 1.7x | 2.6x	1.0x | 1.2x | 2.7x	1.0x | 0.8x | 1.2x	1.0x | 1.0x | 1.5x
DP2	1.0x | 0.9x | 2.0x	1.0x | 1.1x | 2.1x	1.0x | 1.0x | 2.3x	1.0x | 1.0x | 2.0x	1.0x | 0.8x | 1.5x	1.0x | 1.1x | 2.2x	1.0x | 0.8x | 1.9x	1.0x | 1.2x | 2.4x
DP3	1.0x | 0.8x | 3.1x	1.0x | 1.1x | 5.0x	1.0x | 0.9x | 5.2x	1.0x | 1.1x | 4.2x	1.0x | 0.9x | 1.8x	1.0x | 1.2x | 1.6x	1.0x | 1.0x | 1.4x	1.0x | 1.1x | 1.7x
DP4	1.0x | 0.9x | 5.9x	1.0x | 0.9x | 27.1x	1.0x | 1.0x | 49.2x	1.0x | 1.1x | 3.4x	1.0x | 0.8x | 1.6x	1.0x | 0.7x | 1.4x	1.0x | 1.5x | 1.9x	1.0x | 0.8x | 1.2x
DP5	1.0x | 0.9x | 13.3x	1.0x | 1.1x | 15.3x	1.0x | 13.7x | 61.5x	1.0x | 1.1x | 23.7x	1.0x | 1.0x | 1.5x	1.0x | 1.1x | 1.3x	1.0x | 1.5x | 1.1x	1.0x | 1.0x | 1.4x
Average	1.0x | 1.0x | 348x	1.0x | 1.1x | 936x	1.0x | 11.2x | 316x	1.0x | 1.1x | 730x	1.0x | 1.0x | 1.8x	1.0x | 1.0x | 1.8x	1.0x | 1.3x | 1.3x	1.0x | 1.0x | 1.6x

Note: A cell with a filled background indicates that the column dataset contains instances of the row pattern. The average row is calculated using the average CPU time rather than the average of the ratios.

parallel optimizations and optimal settings. Recall that pattern size used in experiments is up to 36 events. we believe this size is sufficient to encompass the majority of TTP patterns. Alternatively, we may consider adopting the decomposition approach mentioned in **preprocessing** on larger patterns to reduce the size of individual patterns. Except for a large number of complex patterns, high-throughput event streams in real-world scenarios may pose challenges in processing time. To overcome this, we may introduce pipeline techniques and independent operations parallelization within each layer to increase the throughput of IPMES.

In summary, these results from both simulated and real-world data show that IPMES is effective and feasible for practical TTP detection. Although improper window size settings may impact detection sensitivity or efficiency, we provide practical guidelines for selecting appropriate window sizes.

V. Limitations and Future Work

Even if the number of generated states is reduced as much as possible, intentional attacks can still cause the pool to overflow in order to avoid detection. However, the event generation rate of the attacks is bound to increase noticeably, and the attacks may be detected by other frequency-based or sequence-based detection methods [38]–[41]. In other words, using IPMES can partially limit the stealthiness of attacks.

The behavioral pattern definitions of this study cannot describe event occurrence times in a range (e.g., describe a process writing the same file 3–5 times) or a reachability relation between the involved system entities (e.g., describe some secret file leaking information to an untrusted socket). Defining multiple behavioral patterns for different event occurrence times or the possible number of internal nodes between starting entities and end entities is not reasonable, because these similar behavioral patterns result in repeated matching for most of the same events. However, these features should be included to extend a behavioral pattern's descriptive power in the future. This also requires careful design because additional descriptive power usually increases detection overhead.

VI. Conclusion

The extended dwell time of APT attacks poses a challenge in detecting them within vast provenance graphs. To address this, we present IPMES, a specialized tool for incremental TTP detection over system audit event streams, aiding in the identification of APT attack steps. IPMES employs a divide-and-conquer approach to match complex graph-based patterns and incorporates specific optimizations to enhance efficiency, a capability not realized by general-purpose CEP tools. Furthermore, IPMES can handle events with identical timestamps and interval timestamps without losing their order relation during matching. Evaluations through experiments on synthetic and real-world data have demonstrated that IPMES is a practical TTP detection tool. IPMES is expected to be provided to Telecom Technology Center, Institute for Information Industry, and Industrial Technology Research Institute organizations in Taiwan to facilitate the detection of APT attacks.

Acknowledgment

The work was supported by National Science and Technology Council of the Republic of China (Grant No: 112-2221-E-001-012 and 112-2634-F-001-001-MBK).

References

[1] "The cyber kill chain," https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html (accessed Nov. 1, 2022).

[2] "Mitre att&ck," https://attack.mitre.org (accessed Nov. 1, 2022).

[3] "Audit system events," https://learn.microsoft.com/en-us/windows/security/threat-protection/auditing/basic-audit-system-events (accessed Nov. 1, 2022).

[4] "Understanding audit log files," https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/security_guide/sec-understanding_audit_log_files (accessed Nov. 1, 2022).

[5] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "Holmes: Real-time apt detection through correlation of suspicious information flows," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1137–1152.

[6] S. M. Milajerdi, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1795–1812.

[7] P. Gao, X. Xiao, D. Li, Z. Li, K. Jee, Z. Wu, C. H. Kim, S. R. Kulkarni, and P. Mittal, "Saql: A stream-based query system for real-time abnormal system behavior detection," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 639–656.

[8] "Flinkcep — complex event processing for flink," https://nightlies.apache.org/flink/flink-docs-master/docs/libs/cep/ (accessed Nov. 1, 2022).

[9] "Siddhi — stream processing and complex event processing engine," https://github.com/siddhi-io/siddhi (accessed Nov. 1, 2022).

[10] "Esper — complex event processing, streaming sql and event series analysis," https://github.com/espertechinc/esper (accessed Nov. 1, 2022).

[11] Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Time constrained continuous subgraph search over streaming graphs," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 2019, pp. 1082–1093.

[12] Y. Li, L. Zou, M. T. Özsu, and D. Zhao, "Space-efficient subgraph search over streaming graph with timing order constraint," *IEEE Transactions on Knowledge and Data Engineering*, vol. 34, no. 9, pp. 4453–4467, 2022.

[13] W. U. Hassan, A. Bates, and D. Marino, "Tactical provenance analysis for endpoint detection and response systems," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1172–1189.

[14] "Sigma - generic signature format for siem systems," https://github.com/SigmaHQ/sigma (accessed Feb. 1, 2024).

[15] "Ibm security qradar," https://www.ibm.com/products/qradar-siem (accessed Feb. 1, 2024).

[16] "Splunk enterprise security," https://www.splunk.com/en_us/products/enterprise-security.html (accessed Feb. 1, 2024).

[17] R. Wei, L. Cai, L. Zhao, A. Yu, and D. Meng, "Deephunter: A graph neural network based approach for robust cyber threat hunting," in *Security and Privacy in Communication Networks: 17th EAI International Conference, SecureComm 2021, Virtual Event, September 6–9, 2021, Proceedings, Part I 17*. Springer, 2021, pp. 3–24.

[18] E. Altinisik, F. Deniz, and H. T. Sencar, "Provg-searcher: A graph representation learning approach for efficient provenance graph search," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2247–2261.

[19] A. Alsaheel, Y. Nan, S. Ma, L. Yu, G. Walkup, Z. B. Celik, X. Zhang, and D. Xu, "Atlas: A sequence-based learning approach for attack investigation," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 3005–3022.

[20] M. M. Anjum, S. Iqbal, and B. Hamelin, "Anubis: a provenance graph-based framework for advanced persistent threat detection," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, 2022, pp. 1684–1693.

[21] W. Fan, J. Li, S. Ma, N. Tang, Y. Wu, and Y. Wu, "Graph pattern matching: From intractable to polynomial time," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 264–275, 2010.

[22] W. Fan, J. Li, S. Ma, N. Tang, and Y. Wu, "Adding regular expressions to graph reachability and pattern queries," in *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011, pp. 39–50.

[23] S. Ma, Y. Cao, W. Fan, J. Huai, and T. Wo, "Capturing topology in graph pattern matching," *Proceedings of the VLDB Endowment*, vol. 5, no. 4, 2011.

[24] L. Chen and C. Wang, "Continuous subgraph pattern search over certain and uncertain graph streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 8, pp. 1093–1109, 2010.

[25] C. Song, T. Ge, C. Chen, and J. Wang, "Event pattern matching over graph streams," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 413–424, 2014.

[26] Z. Xu, Z. Wu, Z. Li, K. Jee, J. Rhee, X. Xiao, F. Xu, H. Wang, and G. Jiang, "High fidelity data reduction for big data security dependency analyses," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 504–516.

[27] "T1070-indicator removal," https://attack.mitre.org/techniques/T1070/ (accessed Feb. 15, 2024).

[28] "T1053-scheduled task/job," https://attack.mitre.org/techniques/T1053/ (accessed Feb. 15, 2024).

[29] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, 1971, pp. 151–158.

[30] N. Livne, "A note on #p-completeness of np-witnessing relations," *Information processing letters*, vol. 109, no. 5, pp. 259–261, 2009.

[31] "Adversary emulation plans," https://attack.mitre.org/resources/adversary-emulation-plans (accessed Nov. 1, 2022).

[32] "Cve-2021-37678," https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-37678 (accessed Nov. 1, 2022).

[33] A. Gehani and D. Tariq, "Spade: Support for provenance auditing in distributed environments," in *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*. Springer, 2012, pp. 101–120.

[34] "Spade: Support for provenance auditing in distributed environments," https://github.com/ashish-gehani/SPADE (accessed Nov. 1, 2022).

[35] "Transparent computing engagement 3," https://github.com/darpa-i2o/Transparent-Computing/blob/master/README-E3.md (accessed Nov. 1, 2022).

[36] "Neo4j graph database," https://neo4j.com (accessed Nov. 1, 2022).

[37] J. Zeng, Z. L. Chua, Y. Chen, K. Ji, Z. Liang, and J. Mao, "Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics." in *NDSS*, 2021.

[38] M. Xie, J. Hu, X. Yu, and E. Chang, "Evaluating host-based anomaly detection systems: Application of the frequency-based algorithms to adfa-ld," in *Network and System Security: 8th International Conference, NSS 2014, Xi'an, China, October 15-17, 2014, Proceedings 8*. Springer, 2014, pp. 542–549.

[39] W. Haider, J. Hu, and M. Xie, "Towards reliable data feature retrieval and decision engine in host-based anomaly detection systems," in *2015 IEEE 10th Conference on Industrial Electronics and Applications (ICIEA)*. IEEE, 2015, pp. 513–517.

[40] T. Rachidi, O. Koucham, and N. Assem, "Combined data and execution flow host intrusion detection using machine learning," in *Intelligent Systems and Applications: Extended and Selected Results from the SAI Intelligent Systems Conference (IntelliSys) 2015*. Springer, 2016, pp. 427–450.

[41] T. Mouttaqi, T. Rachidi, and N. Assem, "Re-evaluation of combined markov-bayes models for host intrusion detection on the adfa dataset," in *2017 Intelligent Systems Conference (IntelliSys)*. IEEE, 2017, pp. 1044–1052.