Progressive Processing of System-Behavioral Query

Jiaping Gui
NEC Laboratories America, Inc.
jgui@nec-labs.com

Xusheng Xiao
Case Western Reserve University
xusheng.xiao@case.edu

Ding Li
NEC Laboratories America, Inc.
dingli@nec-labs.com

Chung Hwan Kim
NEC Laboratories America, Inc.
chungkim@nec-labs.com

Haifeng Chen
NEC Laboratories America, Inc.
haifeng@nec-labs.com

ABSTRACT

System monitoring has recently emerged as an effective way to analyze and counter advanced cyber attacks. The monitoring data records a series of system events and provides a global view of system behaviors in an organization. Querying such data to identify potential system risks and malicious behaviors helps security analysts detect and analyze abnormal system behaviors caused by attacks. However, since the data volume is huge, queries could easily run for a long time, making it difficult for system experts to obtain prompt and continuous feedback. To support interactive querying over system monitoring data, we propose *ProbeQ*, a system that progressively processes system-behavioral queries. It allows users to concisely compose queries that describe system behaviors and specify an update frequency to obtain partial results progressively. The query engine of *ProbeQ* is built based on a framework that partitions *ProbeQ* queries into sub-queries for parallel execution and retrieves partial results periodically based on the specified update frequency. We concretize the framework with three partition strategies that predict the workloads for sub-queries, where the adaptive workload partition strategy (AdWd) dynamically adjusts the predicted workloads for subsequent sub-queries based on the latest execution information. We evaluate the prototype system of *ProbeQ* on commonly used queries for suspicious behaviors over real-world system monitoring data, and the results show that the *ProbeQ* system can provide partial updates progressively (on average 9.1% deviation from the update frequencies) with only 1.2% execution overhead compared to the execution without progressive processing.

CCS CONCEPTS

• **Security and privacy → Usability in security and privacy**; *Malware and its mitigation*; *Operating systems security*; *Software security engineering*.

KEYWORDS

progressive processing, system monitoring, behavioral query, partition, responsiveness

ACM Reference Format:
Jiaping Gui, Xusheng Xiao, Ding Li, Chung Hwan Kim, and Haifeng Chen. 2019. Progressive Processing of System-Behavioral Query. In *2019 Annual Computer Security Applications Conference (ACSAC '19), December 9–13, 2019, San Juan, PR, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3359789.3359818

1 INTRODUCTION

Modern computer systems are fairly complex due to the numerous installed software programs. Existing application logs (e.g., web logs and firewall logs) provide only a partial view of certain system behaviors [36, 54], and thus are insufficient for analyzing advanced cyber attacks, such as advanced persistent threats (APT) [18]. This motivates the recent trend of leveraging system monitoring logs for security analytics to find possible anomalies. Such monitoring data records system-level interactions among software programs and system resources (e.g., processes, files, network sockets) as a sequence of events over time [5, 7, 12]. A typical event may indicate that at timestamp $t1$, a process originated from a system/software program listens to a specific port or another process reads the data from a specific file.

As system monitoring data provides a global view of system behaviors over time, querying such data for suspicious system behaviors provides valuable information for the detection of system anomalies. For example, system experts may want to know whether certain risky system behaviors, such as malware infection and program crashes, occur at what time range. The knowledge about these risky system behaviors can be either from publicly available knowledge bases [4, 8, 11, 13, 17], or from specific definitions of suspicious behaviors in an organization.

To investigate these interesting system behaviors, system experts often perform *interactive querying*, where they iteratively write queries, acquire quick feedback (usually within a minute) and refine the queries, etc. However, system monitoring produces a huge amount of daily data [33] (over 50GB for an organization consisting of 100 computers), and various analyses require queries over data collected for a long period of time. For example, malicious behaviors involved in APT attacks may require several months to one year worth of data for investigation. Due to the large amount of data, the execution of a query over such data could easily last for more than 10 minutes, making interactive querying difficult to realize.

ACSAC '19, December 9–13, 2019, San Juan, PR, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7628-0/19/12…$15.00
https://doi.org/10.1145/3359789.3359818

To support effective and efficient interactive querying of system monitoring data, we propose *ProbeQ*, a domain-specific query system that progressively processes system-behavioral queries. It allows users to specify queries of system behaviors over system monitoring data. *ProbeQ* is designed with three goals: (1) query syntax is designed to *intuitively* and *concisely* specify interesting system behaviors; (2) query execution is *optimized* for searching system monitoring data; (3) *progressive processing* is employed to report partial results progressively rather than at the end of the whole search. Query 1 is an example query for finding whether any program writes to system log files, which may indicate an attacker clears her traces after the attack:

```
host = 1 // host id
(from "02/01/2019" to "02/07/2019") // time window
// event pattern
proc p1 write file f1['/var/log/wtmp'||'/var/log/lastlog']
return distinct p1, f1 // result projection
update 5s // progressive update frequency
```

Query 1: *ProbeQ* query for a suspicious behavior

Query Language. *ProbeQ* adopts the syntax format *{subject-operation-object}* to specify event patterns for system behaviors, where system programs are represented as *subjects*, system resources are represented as *objects*, and interactions are represented as *operations* initiated by subjects and targeting on objects. As shown in Query 1, *ProbeQ*'s syntax models system behaviors directly and is user-friendly. Moreover, the clause `update 5s` indicates that the query employs progressive processing and aims to return the results every five seconds. Such features are helpful in supporting interactive querying, since system experts could receive quicker feedback from the results and may stop the search if they have found the target behaviors or they want to refine the query.

Parallel Query Execution. The query engine of *ProbeQ* optimizes the search based on the domain-specific characteristics of system monitoring data. In system monitoring data, each event is generated with a timestamp at a specific host in the organization. Thus, these events exhibit strong *spatial* and *temporal* properties: events in different hosts are independent and events in the same host are independent along the time. Based on such *spatial* and *temporal* properties of the data, we propose two types of parallel strategies that partition a query into sub-queries by uniformly splitting the time window or the total workload. The query engine then executes these sub-queries in parallel.

Progressive Processing. Even with parallel executions, the performance speed-up (> 50%) is bounded by the hardware limitations (e.g., CPU and disk I/O), and executing a query that searches a week's data (about 250GB) may still take more than 10 minutes in a server. To enable quicker and continuous feedback, *ProbeQ* employs the technique of progressive processing to report partial query results based on a specified update frequency. System experts typically have different expectations of update frequencies for different queries. For example, some queries that look for the existence of a behavior may require a shorter update frequency (e.g., once every two seconds); other queries that look for what exact files a specific program writes may require a longer update frequency to collect more results for analysis.

The key quality of our progressive query processing is measured by two metrics: (1) *Q.1 Responsiveness*: it should report new results in every update cycle and (2) *Q.2 Overhead*: its overhead should be as low as possible. Clearly, if we do not conduct any query partitioning, we cannot receive any update until the end, failing *Q.1*. At another extreme, if we obtain very frequent updates, the overhead becomes unacceptably high, failing *Q.2*. For example, given a query with a one-day time window, if we spawn a sub-query for each second of the time window, i.e., 3600*24 sub-queries in total, we can easily meet *Q.1*. However, such extreme strategy brings a significant overhead due to various costs such as establishing connections to databases and parsing the sub-queries, causing a query to take more than 28 hours to finish (originally 270 seconds, about 350 times slower). Given an update frequency U_f (e.g., 10 seconds), an ideal partition is to assign a certain amount of workloads to the sub-queries such that *each sub-query takes exact* U_f *time to finish*, incurring the lowest overhead possible while satisfying *Q.1*. In practice, it is almost impossible to achieve such ideal partitioning due to the complex nature of computer systems, but a suboptimal solution is still possible: making the average execution time of the sub-queries *close to* U_f.

Based on these insights, we first propose two partition strategies (*Fixed Time Window (FixTW)* and *Fixed Workload (FixWd)*) that predict the workload for sub-queries based on the measured event processing rate (i.e., # events processed per second) and the given update frequency. As suggested by their names, FixTW computes a time window based on the predicted workload and uses the time window to partition the query, while FixWd ensures that each sub-query has the same workload as the predicted workload. However, we observed that databases typically employ cache mechanism to load a chunk of data from disk, and the data stored for a host for a certain period of time is often loaded together. With the loaded data in the cache, some sub-queries partitioned by these two strategies finish execution much faster. Thus, the inaccurate prediction of workload causes the average execution time of sub-queries to be significantly larger than the update frequency, failing *Q.1*.

Based on these observations, we propose an *Adaptive Workload Partition Strategy (AdWd)*, which dynamically predicts the workloads for subsequent sub-queries based on the latest execution information of already-finished sub-queries. The key insight of AdWd is to leverage online learning technique, such as gradient descent [42, 43, 45], for adjusting the event processing rates based on dynamic execution information.

Evaluations. We have built a prototype system of *ProbeQ* based upon auditd [12] and ETW [7], and deployed it in an anonymous enterprise comprising around 100 hosts. To evaluate the effectiveness of the *ProbeQ* system, we conduct comprehensive evaluations on real-world suspicious behaviors over five-day data (about 230GB stored in PostgreSQL databases [14]). The evaluations are conducted on a server with an Intel(R) Xeon(R) CPU E5-2660 (2.20GHz), 64GB of RAM, and a RAID that supports four concurrent reads/writes. The results show that AdWd is able to progressively provide the partial results (on average 9.1% deviation from the update frequencies) and is 32.8% faster than the corresponding SQL queries. The overhead of progresseive processing is only 1.2% compared to no progressive processing for the execution of same queries.

We also compare AdWd with FixWd and FixTW, and the results show that AdWd outperforms FixWd and FixTW on the deviations of sub-query execution time from the update frequencies (6.3% vs. 119% and 148%) and the percentage in getting new results at every update cycle i.e., *Q.1* responsiveness (84.5% vs. 71.5% and 62.7%). In terms of *Q.2* overheads on the total execution time without progressive processing, AdWd (9.5%) has a comparable performance to FixWd (6.4%) and FixTW (5.9%) when the update frequency is greater than 2s.

We summarize our major contributions as follows:

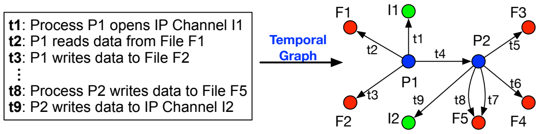
- A domain specific language in *ProbeQ* that allows system experts to concisely specify interesting system behaviors and update frequencies to obtain partial results progressively.
- A study on different parallel strategies for partitioning a *ProbeQ* query into sub-queries for parallel execution.
- A framework of progressive processing that partitions a *ProbeQ* query into sub-queries and reports partial results based on a specified update frequency. We propose a measurement process to model event processing rates for the query and propose three partitioning strategies (FixTW, FixWd, and AdWd) that predict workloads for sub-queries based on the event processing rate.
- Comprehensive evaluations on the *ProbeQ* queries for real-world suspicious behaviors and comparisons among FixTW, FixWd, and AdWd.

2 SYSTEM MONITORING DATA

System monitoring data records the interactions among system programs and system resources as events. Each of the recorded interactions represents a system event that occurs at a particular host. It consists of the initiator, the type, and the target of the interaction. Initiators are processes originated from software programs such as firefox, while targets are system resources, such as files, processes and network connections on Windows, Linux/Unix or Mac OS. Besides, each interaction is associated with a timestamp, indicating when the interaction occurs.

Based on above data characteristics, system monitoring data can be represented as a temporal graph, with system entities as heterogeneous nodes and system events as edges with timestamps (i.e., pointing from the initiator node to the target node). Figure 1 gives an example of modeling system monitoring data as a temporal graph. In Figure 1, we have nine events annotated with the timestamps $t1$–$t9$. We have shown three major types of system entities: files ($F1$–$F5$), processes ($P1$–$P2$), and network connections ($I1$–$I2$). Each edge points from the initiator (subject) to the target (object). This indicates that the initiator performs a certain operation on the target. For example, at timestamp $t1$, the initiator process $P1$ opens the target network connection $I1$ to send or receive some data, as indicated by the type of the interaction "open". By modeling system monitoring data as temporal graphs, we can easily understand the temporal event nature of the data and observe the system behaviors in terms of their interactions with system resources.

Data Collection and Storage. To collect such data, we implement data collection agents for Windows and Linux based on ETW event tracing [7] and Linux Audit Framework [12]. Our current agents monitor system audit events about the system calls that are crucial in security analysis. The monitored system calls are mapped to three major types of events: (1) process creation and termination, (2) file accesses, and (3) network accesses. The collected data is then sent to a central server, where the data will be modeled and stored into databases. At this server side, a global clock is used to correct the time drifting of the events collected at different agents.



t1: Process P1 opens IP Channel I1
t2: P1 reads data from File F1
t3: P1 writes data to File F2
⋮
t8: Process P2 writes data to File F5
t9: P2 writes data to IP Channel I2

Figure 1: System monitoring data as a temporal graph

Table 1: Representative grammar of *ProbeQ*

<table>
<tr><td><probeq></td><td>::=</td><td>(<evt_cstr>)* <query></td></tr>
<tr><td><evt_cstr></td><td>::=</td><td><cstr>| ‘(’ <time_window>‘)’</td></tr>
<tr><td><query></td><td>::=</td><td><evt_patt>+ <return><progress>? <t_num>?</td></tr>
<tr><td><evt_patt></td><td>::=</td><td><entity><op_exp><entity><evt>?</td></tr>
<tr><td><entity></td><td>::=</td><td><type><entity_id>? (‘[’ <attr_cstr>‘]’)?</td></tr>
<tr><td><attr_cstr></td><td>::=</td><td><cstr></td></tr>
<tr><td></td><td>|</td><td><attr_cstr>(‘,’ | ‘||’) <attr_cstr></td></tr>
<tr><td></td><td>|</td><td>‘(’ <attr_cstr>‘)’</td></tr>
<tr><td><cstr></td><td>::=</td><td><attr_name><bop><val></td></tr>
<tr><td></td><td>|</td><td>‘!’? <val></td></tr>
<tr><td></td><td>|</td><td><attr_name>‘not’? ‘in’ <val_set></td></tr>
<tr><td></td><td>|</td><td><attr_name>‘not’? ‘in’ <query_id></td></tr>
<tr><td><val_set></td><td>::=</td><td>‘(’ <val>(‘,’ <val>)* ‘)’</td></tr>
<tr><td><op_exp></td><td>::=</td><td>‘!’? <op></td></tr>
<tr><td></td><td>|</td><td><op_exp>(‘,’ | ‘||’) <op_exp></td></tr>
<tr><td></td><td>|</td><td>‘(’ <op_exp>‘)’</td></tr>
<tr><td><evt></td><td>::=</td><td>‘as’ <evt_id>(‘[’ <attr_cstr>‘]’)?</td></tr>
<tr><td><return></td><td>::=</td><td>‘return’ ‘distinct’? <res>(‘, ’ <res>)*</td></tr>
<tr><td><res></td><td>::=</td><td><entity_id>(‘.’<attr_name>)?</td></tr>
<tr><td><progress></td><td>::=</td><td>‘update’ <val><timeunit></td></tr>
<tr><td><t_num></td><td>::=</td><td>‘using’ <val>‘worker’</td></tr>
</table>

We store the collected data in relational databases powered by PostgreSQL [14]. Relational databases come with mature indexing mechanisms and are scalable to the massive system monitoring data. When storing data, we partition the data based on its *temporal and spatial properties*: separating groups of hosts into table partitions and dumping one database per day for the data collected on that day. Besides, we apply *data deduplication* by storing subjects and objects in entity tables and events in relationship tables. To query an event, users need to join three tables: subject table, object table and event table. We build various types of indices on the attributes that will be queried frequently, such as the executable name of process, the file name, and the source/destination address of network connection.

3 *PROBEQ* DESIGN

Existing popular query languages (e.g., SQL) fail to *intuitively* and *concisely* specify interesting system behaviors on one or more hosts within a certain time window. To address this challenge, we design a new grammar in *ProbeQ*. Table 1 shows the representative set of the grammar rules of *ProbeQ*.

3.1 Data Schema Definition

ProbeQ works for domains where the data can be modeled as entities and events with temporal information, such as enterprise-wide system monitoring data, sensor-based Internet-of-Thing (IoT) data, and performance monitoring information of data centers. Similar to the Data Definition Language in SQL [16], given a type of domain data, *ProbeQ* allows user-defined schema for specifying the entities, events, and their attributes for the domain data. In this work, we mainly focus on querying over system monitoring data.

3.2 *ProbeQ* Query

ProbeQ enables users to intuitively specify event patterns for describing interested system behaviors w.r.t. the interactions with system resources. A *ProbeQ* query consists of two major parts: event constraints (< *evt_cstr* >) that specifies the constraints for hosts and time window, and an event pattern (< *evt_patt* >) that specifies subject/object, event operation, and an optional event ID. Both subjects and objects can have optional attributes (< *attr_cstr* >).

In a *ProbeQ* query, the subject and object are specified as entities (< *entity* >), which consist of entity type (file, process, network connection), optional entity ID, and optional attributes. In Query 1, `proc p` specifies a process entity p, and `file f1['/var/log/wtmp' || '/var/log/lastlog']` specifies a file entity $f1$. The operation (< *op_exp* >) specifies the operation initiated by the subject and targeting at the object, such as *reading* a file. Logical operators (“,” for AND, “||” for OR, “!” for NOT) can be used to combine multiple operations. The event return (< *return* >) specifies which attributes of the found events to return. Based on the events' attributes specified in the event return rule, we output the result tuples.

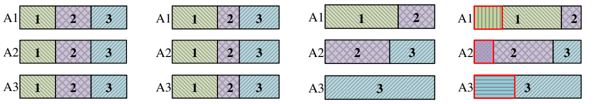
Progressive Processing. The progressive processing clause (<*progress*>) specifies the update frequency for the progressive processing, and the thread number (< *t_num* >) specifies the number of worker threads used to execute the subqueries. If the update frequency is not specified, the query execution will not report the results until the end of the search, and fully leverage the spatial and temporal properties of the system monitoring data to optimize the overall execution time. If the thread number is not specified, the optimal number is inferred based on the number of events to be search over. The details of how the optimal number of threads is inferred is described in Section 4.

3.3 Example

Query 1 and Query 2 give two suspicious behaviors expressed in *ProbeQ*. Query 1 aims to find whether any software program modifies system logs at the host with `host = 1` from `02/01/2019` to `02/07/2019`. Query 2 aims to find whether any software program reads a set of history files (e.g., `.viminfo` and `.bash_history`) concatenated by the `OR` operator (`||`), indicating a risky behavior that probes the user history.

```
proc p read file f['.lesshst' || '.viminfo' || '.bash_history' ||
     '.pgadmin_histoqueries'] as evt
return distinct p, f, evt.starttime
```

Query 2: Command history probing



(a) UniTW (b) ParaWd (c) SeqWd (d) SeqWdInit

Figure 2: Four different types of parallel strategies

4 PARALLEL QUERY EXECUTION

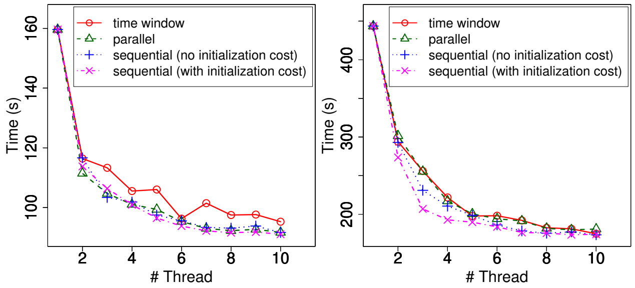
By leveraging the spatial and temporal properties of system monitoring data (independent in both dimensions), we propose four parallel strategies that partition a query into sub-queries with smaller time windows and fewer involved hosts, and execute the sub-queries in parallel to optimize the query execution. We next present the parallel strategies in detail.

Uniform Time Window vs. Uniform Workload: As events in system monitoring data are independent over time, a straightforward partition is to uniformly split the time window for all sub-queries (referred to as *uniform time-window* partition (UniTW)). Figure 2a shows an example of UniTW, which partitions a query searching one-day data of three hosts into three sub-queries (i.e., each sub-query has a time window of eight hours). However, UniTW usually does not split the workload fairly for each sub-query. In practice, a host usually produces events at a different rate during different times of a day. For example, a developer machine produces much more monitoring events during the day than the night.

Based on this observation, we propose another type of parallel strategy, *uniform workload* partition, which balances the workload (i.e., # events to process) for sub-queries. To uniformly partition the workload of a query, we first need to know the total workload the query will process. We collect the statistics of the number of events received every minute for each host, and then predict the workload based on the query's time window and involved hosts. With the predicted workload of the query, we next describe two different strategies that achieve uniform workload partition.

Parallel (ParaWd) vs. Sequential (SeqWd): The first uniform workload strategy, called *parallel workload* partition (ParaWd), is to partition the time window into smaller time windows, so that in each time window, the number of events for all the involved hosts are the same (i.e., the workloads are equal for each sub-query), as shown in Figure 2b. The second uniform workload strategy, called *sequential workload* partition (SeqWd), is to first sequentially concatenate the events of all the hosts as an array of events, and then divide the array uniformly for each sub-query, as shown in Figure 2c.

Initialization Cost: In theory, both of these strategies uniformly partition the workload of a query. However, when we applied the strategies, we found that neither of these strategies lead to the same execution time among all sub-queries. The reason is that there is an initialization cost (e.g., loading indexes) when the query accesses the data of a host for the first time. The subsequent accesses benefit from the cache mechanism and thus have no such cost (or at least the cost is negligible). Hence, we propose the third uniform workload strategy, called *sequential workload with initialization cost* partition (SeqWdInit). As shown in Figure 2d, this strategy takes into account the initialization cost for the first access to each



(a) Events of One Host **(b) Events of Three Hosts**

Figure 3: Comparison of different parallel strategies

host's data. It converts the initialization time on each host to a virtual workload (i.e., the initialization time multiplied by the event processing rate), inserts the virtual workloads (shown in red border) to the workloads of each host, and then applies SeqWd to partition the combined workload.

Comparison of Parallel Strategies: We conducted a measurement study of these parallel strategies on two sets of typical queries that span one day and multiple days. The first set of queries search events from a single host, and the second set of queries search events from multiple hosts, each of which belongs to a different partition table (i.e., different groups). Each query was executed for three times and the average execution time was computed as the final result. To execute the sub-queries in parallel, we used a worker thread to execute each sub-query.

Figure 3a shows the results of one query (all others have similar results). Among all parallel strategies, SeqWdInit achieves the best performance, while UniTW performs the worst. The reason is that the events generated on the host shown in Figure 3 are non-uniformly distributed (e.g., higher throughput in the daytime than at night). Thus, the partitioned sub-queries have different workloads, causing some sub-queries to take much longer execution time than the others. But UniTW has a comparable performance to that of ParaWd for the query that searches the events from three hosts (shown in Figure 3b), since these hosts have a constant event generation rate for the whole day and sub-queries partitioned by UniTW have similar workloads.

Among four strategies, SeqWd and SeqWdInit perform better than ParaWd. The reason is that each sub-query partitioned by SeqWd and SeqWdInit usually involve events from one or two hosts, while the sub-queries partitioned by ParaWd involves all hosts. As events belong to different hosts are stored in different partition tables, accessing events from multiple partition tables is slower than accessing the same amount of events from one partition table. Moreover, since the initialization cost is not negligible (as shown in Figure 4c), the consideration of this cost further improves the performance as indicated by the performance of SeqWdInit.

Performance Improvement w.r.t. Degree of Parallelism: In general, parallelism improves the query execution time for all parallel strategies. In Figure 3, when the number of worker threads is less than five, the performance improvement by increasing the number of worker threads is more significant (at least 10% by assigning one more worker thread). However, running with over four threads does not contribute to notable improvements (i.e., flat tail lines). The reason is that our server has a RAID that supports four concurrent read/write operations, and four worker threads in theory can already make full use of system resources. When there are more than four worker threads, some threads are accessing some resources while the others are waiting for these resources to be released. In this case, increasing the number of worker threads does not introduce significant improvements.

These results also show that even though the performance improvements brought by parallelism is about 50%, queries could still run for over 200 seconds. This further motivates us to employ progressive processing to assist interactive querying.

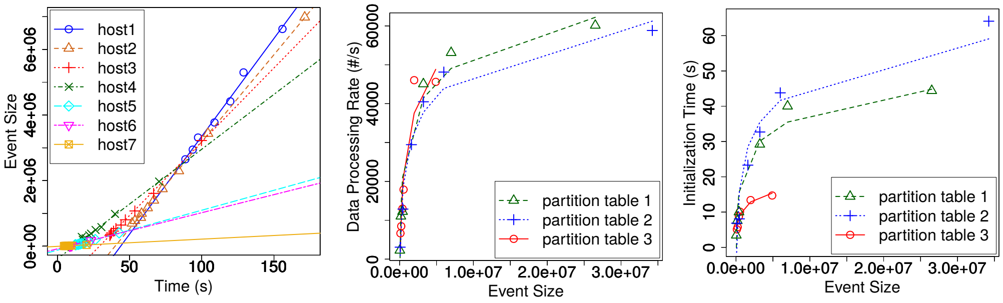
5 MODELING EVENT PROCESSING RATES AND INITIALIZATION TIME

To devise strategies for progressive processing, we first conduct a measurement study of event processing rates and initialization time for the queries that are commonly used to search risky system behaviors, such as Query 1. Unlike general database queries (e.g., SQL queries) that may query different tables for different purposes, in the domain of system monitoring data, behaviorial queries typically search system behaviors w.r.t. system resources like files (described in Section 2), and thus we mainly measure the queries on searching the events rather than other queries that compute statistics or relationships. The study is conducted on the real-world data stored in PostgreSQL databases. We vary the workloads to uncover the relationship between the query execution time and the workload (i.e., event processing rates).

Result Analysis: Figure 4a shows the relationship between the query execution time and the workload for queries that search behaviors at seven different hosts. As we can see, as the workload increases, the execution time increases linearly. We apply linear regression (commonly used for modeling and predicting system behaviors [27, 34, 49, 52]) on the execution results, and the R^2 and p-values of fitted lines range from 0.977 to 0.999, and 2.22E-014 to 1.73E-006, respectively, indicating a strong linear relationship. Such measurement results are as expected: with the indexes built for the host and timestamp columns of event tables, searching the desired events is to scan the index entries and fetch the matched events, and thus the query execution time is linearly correlated with the number of scanned events.

Impact of Host Event Sizes and Partition Table Sizes: We notice that the event processing rates vary for the events at different hosts. In Figure 4a, the event size for a host decreases from host 1 to host 7. As the event size increases, both the event processing rate (i.e., line slope) and the initialization time (i.e., line intercept) also increases. To understand how host event sizes impact event processing rates and initialization time, we further conduct experiments on the queries searching events at different hosts.

Instead of the linear relationship, both the event processing rate (shown in Figure 4b) and the initialization time (shown in Figure 4c) have strong logarithmic relationships [24, 28, 41, 50, 53] with the total event sizes of all hosts stored in the same partition table. We further measure the event processing rates on the hosts in three partition tables (shown in Table 2), and obtain the same conclusion. The potential reason is that the temporal information of events is



(a) Execution time and workload **(b) ep at different hosts** **(c) it at different hosts**

Figure 4: Investigation on relationships about event processing rate ip, initialization time it, and event size

indexed using a tree index, and the complexity of traversing the tree index to fetch events is $log(\#event)$. We also observe that for the hosts with similar event sizes but in different table partitions, their event processing rates and initialization time are different (not shown here due to space limits).

Model Measurement and Prediction: Based on the measurement study, we model event processing rates and initialization time as linear functions for each host, and use these models to predict the workload based on a given update frequency. To derive the linear function for a host h, we partition a query that searches one-day events on h into ten sub-queries. These sub-queries form an arithmetic progression with common difference of $\frac{1}{10}$ of the total workload. We then run sub-queries in sequence, and apply the linear regression on the execution time and workloads to obtain the linear function. We clear caches before each query execution to minimize the background noise.

However, such measurement process is relatively expensive, especially when the number of hosts is large. To address this problem, we propose a prediction technique based on the observed relationships from the measurement results. We choose the host with the largest event size out of all table partitions as the base host, and use its measured model to predict the models of others. We choose such host since the events belonging to the hosts with a small number of events are typically not stored sequentially and their event processing rates are not representative in reflecting the performance of most queries. Formally, given a host h_m that has the largest event size, it has the event processing rate ep_{max} and belongs to the table partition p_m that has in total n hosts. Then for a host h_i that belongs to a partition table p_k with t hosts, h_i's event processing rate ep is computed as follows:

$$ep = ep_{max} * \frac{\ln(e_{i,k}) * \ln(\sum_{j=1}^{t} e_{j,k})}{\ln(e_{m,m}) * \ln(\sum_{r=1}^{n} e_{r,m})}$$

, where $e_{u,v}$ represents the number of events on the host h_u at the table partition p_v. The initialization time it is computed using the same formula. With the prediction, we only need to run queries for measuring one host, which makes our techniques shown in later sections much more scalable and practical.

Table 2: Statistics of Logarithmic Regression

Table ID	Event processing rate		Initialization time	
	R-squared	p-value	R-squared	p-value
1	0.956	7.37E-4	0.95	9.39E-4
2	0.97	3.37E-4	0.918	2.61E-3
3	0.916	2.73E-3	0.963	5.08E-4

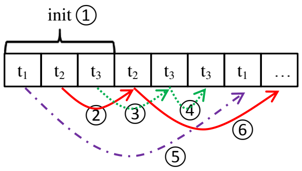


Figure 5: Framework of progressive processing

6 PROGRESSIVE PROCESSING OF *PROBEQ* QUERY

In this section, we first present the framework of progressive processing, explaining how a *ProbeQ* query is partitioned into sub-queries and how to schedule these sub-queries to achieve progressive processing. We then present three different partition strategies that can be plugged into the framework for spawning sub-queries.

6.1 Framework of Progressive Processing

Figure 5 shows *ProbeQ*'s framework of progressive processing. The framework maintains a result queue for each *ProbeQ* query, and probes the queue iteratively based on the specified update frequency to retrieve the latest search results. In order to obtain partial search results for a *ProbeQ* query, the query is partitioned into a set of sub-queries with smaller time window and/or fewer involved hosts, and these sub-queries are executed in parallel (e.g., via worker threads). Depending on the hardware configurations, different numbers of worker threads can be employed to execute the sub-queries. Once a sub-query finishes the execution, its search results are inserted into the query's result queue, and the corresponding worker thread is assigned a new sub-query if any. The newly inserted results will be retrieved in the next probing cycle as latest results.

This framework can be concretized via plugging in a partition strategy that partitions a query into a set of sub-queries. The key to ensure the quality of progressive processing is to devise a strategy that ensures the average execution time of sub-queries is *close to* the update frequency, which can then satisfy both the quality metrics *Q.1* and *Q.2*. We next present three different partition strategies. As SeqWdInit has the best performance, all these strategies follow the same scheme of SeqWdInit.

6.2 Fixed Time Window Partition (FixTW)

For a query searching for events on n hosts $(h_1, \dots, h_n)$ in the time window T, FixTW partitions the query based on a fixed time window (T_i) for events on the host h_i. T_i is computed as $U_f * ep_i / g_i$, where U_f is the update frequency interval, ep_i is the event processing rate of h_i, and g_i is the average data generating rate computed using the total events of h_i divided by the duration of T.

6.3 Fixed Workload Partition (FixWd)

For a query searching for events on n hosts $(h_1, \dots, h_n)$ in the time window T, FixWd partitions the query based on a fixed workload (Wd_i) for events on the host h_i. Wd_i is computed as $U_f * ep_i$, where U_f is the update frequency interval and ep_i is the event processing rate of h_i. Note that the initialization costs are considered as virtual workloads in each host.

6.4 Adaptive Workload Partition (AdWd)

Both FixTW and FixWd assume that the event processing rates are constant. However, as database systems usually employ caches and the event processing rates are much higher when the events are in the caches, the event processing rates fluctuate wildly during runtime. Thus, the execution time of the sub-queries partitioned by FixTW and FixWd could be far from the update frequency. To address this challenge, we propose an adaptive partition strategy, AdWd, which dynamically adjusts event processing rates during runtime. In particular, we propose a technique called *online adaptive workload prediction*, which leverages a set of latest $\langle execution\_time, workload \rangle$ pairs as feedback to learn new event processing rates and predict workloads for subsequent sub-queries. To guide the adaptive learning, we adopt the optimization algorithm *gradient descent* that is commonly used in online learning, and is also utilized in the areas of program analysis and repair [42, 43, 45].

Learning with gradient descent: The goal of learning is to adjust the event processing rate obtained in the non-cache environment to fit the latest execution results. It computes a new data processing rate (ep) and initialization time (it) that approximate the actual event processing rate in the current running environment. In other words, it finds the local minimum ep', it' such that for each new execution result $\langle x, y \rangle$, where x represents the execution time and y represents the workload, the execution time x' of the next estimated workload y' is closest to the update frequency U_f. To compute the gradient $\vec{g}$ w.r.t. the new training data set $\vec{S}$, we utilize the following loss function $loss(\vec{x}, \vec{y}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - (ep x_i + it))^2$, where $\langle \vec{x}, \vec{y} \rangle$ are the execution results in $\vec{S}$ whose size is N.

By taking the derivatives of loss function equation, we obtain the gradient $\vec{g}$:

$$g_a = \frac{\partial loss(\vec{x}, \vec{y})}{\partial ep} = \frac{2}{N} \sum_{i=1}^{N} -x_i * (y_i - (ep * x_i + it)) \quad (1)$$

$$g_b = \frac{\partial loss(\vec{x}, \vec{y})}{\partial it} = \frac{2}{N} \sum_{i=1}^{N} -(y_i - (ep * x_i + it)) \quad (2)$$

After computing the gradient, we update the event processing rate and initialization time respectively as follows: $ep' = ep - \gamma g_a$, $it' = it - \gamma g_b$. The values ep' and it' combine both latest and historical execution results that outline a comprehensive picture of the system runtime environment, while the learning rate γ controls the weight of the latest execution results over the historical results.

Regularization: To avoid over-fitting, we place restrictions on the bounds on the newly learned event processing rate ep'. Over-fitting causes the prediction of a large workload for which the sub-query has no way to return results within the update frequency. Our regularization uses the offline measured/predicted event processing rates as the lower bound. For the upper bound, we compute the instant event processing rate $\frac{y}{x}$ for each latest execution information $\langle x, y \rangle$, and use the largest observed instant event processing rate as the upper bound, which typically indicates that all the workloads assigned to the sub-queries are pre-loaded in the cache.

Algorithm 1: Online Adaptive Workload Prediction

```
Input: $U_f$: update frequency from user input
       $k$: current host
       $S_i$: index of first unprocessed event
       $M[1, \dots, n]$: host measurement/prediction
       $\Delta E$: latest execution information $\langle x, y \rangle$
       $ep_{max}$: observed maximum event processing rate
       $\gamma$: online learning rate
Output: $D$: workload for the subsequent sub-query
Initialize $D \leftarrow 0$ and $U'_f \leftarrow U_f$;
while more workload needed within $U'_f$ do
    $D' \leftarrow 0$;
    $isNew \leftarrow$ false;
    if $Evt[S_i]$ is in a new host $h_k$ then
        $ep, it \leftarrow getEPandIT(M, k)$;
        $ep_{max} \leftarrow ep$;
        $isNew \leftarrow$ true;
        if $it \geq U'_f$ then
            $D \leftarrow D + 1 * ep$;                  // 1s workload
            $S_i \leftarrow S_i + D$;
            return $D$;
    else
        $ep \leftarrow adaptiveLearning(ep, \Delta E, \gamma)$;
        if $ep_{max} \leq \frac{y}{x}$ then
            $ep_{max} \leftarrow \frac{y}{x}$;
    if $isNew$ == true then
        $D' \leftarrow (U'_f - it) * ep$;
    else
        $D' \leftarrow U'_f * ep$;
    // regularization
    $maxD \leftarrow U'_f * ep_{max}$;
    if $D' > maxD$ then
        $D' \leftarrow maxD$;
    $D_r \leftarrow remainingEvent(M, k)$;
    if $D' > D_r$ then
        $U'_f \leftarrow U'_f - \frac{D_r}{ep}$ ;
        $D \leftarrow D + D_r$;
        $k \leftarrow k + 1$
    else
        $D \leftarrow D + D'$;
        break;
$S_i \leftarrow S_i + D$;
return $D$;
```

Algorithm: Algorithm 1 shows how to predict each online adaptive workload. AdWd follows the scheme of SeqWdInit by concatenating the events of all involved hosts as an array *Evt*, converting initialization costs as virtual events, and processing the events in the array sequentially. The inputs to the algorithm are the update frequency (U_f), the current host index (k), the index of the first unprocessed event in *Evt* (S_i), the host measurement/prediction (M), the latest execution information (ΔE), the observed maximum instant event processing rate (ep_{max}), and the learning rate (γ). M stores the information of the involved hosts' offline measured (or predicted) event processing rates and initialization time.

The algorithm starts by initializing the predicted workload (D) to 0 and the remaining time within the update frequency U'_f to U_f (Line 1). If S_i indicates that it is the first time to access the host h_k's data, the event processing rate *ep* and initialization time *it* are retrieved from $M[k]$ accordingly. In this case, if *it* is larger than U'_f (i.e., even without any workload, the execution of this sub-query will not finish within U'_f), we assign the default event size (i.e., events processed within 1s) as the predicted workload D and return D (Lines 9–12). On the other hand, if it is not the first access of the host's data, we set the initialization time as zero and use the adaptive learning to learn a new event processing rate (Line 14). ep_{max} is then updated with the instant event processing rate if necessary (Lines 15–16). Based on the *ep*, the algorithm predicts the workload D' (Lines 17–20). To avoid over-fitting, the algorithm applies the regularization accordingly to smooth D' if necessary (Lines 21–23).

After regularization, we check whether D' exceed the remaining size of the current host h_k (Lines 24–25). If so, we compute the execution time of the remaining workload on the host, deduct it from U'_f, and start a new iteration to predict the workload for a new host h_{k+1} (Lines 26–28). Otherwise, we update the predicted workload D with D' and return D. Finally, the algorithm updates S_i based on D (Line 32).

7 IMPLEMENTATION

Based on the *ProbeQ* grammar (Table 1), we leverage ANTLR 4 [1] to build the lexer and the parser, which can automatically build syntactic parse trees and provide various types of tree walkers. We then build an interpreter on top of the tree walkers to perform semantic analysis of *ProbeQ* queries and execute the queries. After the semantic analysis, our interpreter produces a query context for each *ProbeQ* query and adopts *template-based* [44] synthesis to synthesize queries for retrieving data. As we store data in relational databases, our system synthesizes a SQL query for each sub-query based on the following query template.

```
SELECT |return|
FROM |subject_type| |subject_id|,
     |object_type| |object_id|, |event_type| |event_id|
WHERE |entity_join|
 AND |event_op|
 AND |event_attributes|
```

Query 3: SQL query synthesis template

Table 3: Results of Real-World Case Study

	Update frequency (s)	# sub-queries	Average sub-query execution time (s)	Total time (s)
s1	5	171	5.19	223.4
	10	81	10.55	217.08
	15	67	12.54	212.16
s2	5	142	5.77	205.71
	10	78	10.05	198.36
	15	59	13.08	197.36

8 EVALUATION

We have built a prototype system of *ProbeQ* based on AdWd and deployed it at NEC Labs America comprising around 100 hosts. To evaluate the effectiveness of AdWd, we conduct comprehensive evaluations on the commonly used queries for suspicious behaviors. In our evaluations, we seek to answer the following research questions:

- RQ 1: How effective is AdWd in satisfying two quality metrics of progressive processing?
- RQ 2: How well does AdWd perform in comparison with FixWd and FixTW?
- RQ 3: How much do learning rates impact the performance of AdWd?
- RQ 4: How much do the predicted models in Section 5 impact the performance of AdWd?

8.1 Evaluation Setup

The evaluations are conducted on a server with an Intel(R) Xeon(R) CPU E5-2660 (2.20GHz), 64GB of RAM, and a RAID that supports four concurrent reads/writes. The system monitoring data is stored in a set of PostgreSQL databases, with one database corresponding to one day of data. We use five-day data in April that has a total size of about 230GB for our evaluations, and use 5.0E-4 as the learning rate of the AdWd strategy[1]. Before each query execution, we cleared up the cache as a matter of fact that no data is cached for the first time access. Note that in RQ1 and RQ2 we use the predicted models to obtain all hosts' event processing rates and initialization time (Section 5). While in RQ3, we use the measured models to minimize the background noise when measuring the impacts of learning rates. RQ4 shows the performance comparison between the measured models and the predicted models.

8.2 RQ 1: Effectiveness of AdWd

We conduct evaluations on two suspicious behaviors in the real world: (1) *s1*: command history probing; (2) *s2*: processes erasing traces from system files. The *ProbeQ* queries for *s1*–*s2* are Query 2 and Query 1. These suspicious behaviors are observed rarely in daily usage and thus are common to be searched with a large time window. In our evaluations, we set the time window for *s1*–*s2* to be five days in April on a host. The SQL queries for *s1*–*s2* run for 364.8s and 268.0s respectively. The number of events to be processed is over 44 million.

Table 3 shows the results when using progressive processing with three different update frequencies (5s, 10s, and 15s). These update frequencies are the most representative frequencies specified by system experts. Based on the study results at Section 4, we choose

[1] Learning rates ranging from 1.0E-4 to 1.0E-3 achieve best performance.

Table 4: Comparison among FixTW, FixWd, and AdWd

Strategy	Average sub-query execution time (s)				
	2s	5s	10s	15s	20s
AdWd (5.0E-4)	2.14	5.29	10.71	14.5	18.34
FixWd	5.4	12.1	21.5	28.9	34.79
FixTW	5.91	13.37	24.46	33.5	41.89

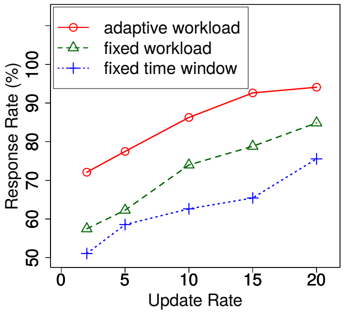


Figure 6: *Q.1 Responsiveness* Comparison among FixTW, FixWd, and AdWd

four worker threads to run the sub-queries. The results show that the average sub-query execution time is quite *close to* the update frequencies (only a 9.1% deviation from the update frequencies), and on average the total execution time of the queries is 32.8% faster than the corresponding SQL queries without any optimization. To evaluate the overhead of AdWd on the total execution time, we run SeqWdInit (i.e., no progressive processing) directly on *s1*–*s2* and the execution time is 220.69s and 197.14s, respectively. Such results show that for these two queries, AdWd has a negligible overhead (max overhead 4.3%) on the total execution time.

8.3 RQ 2: Comparison with FixWd and FixTW

In RQ2, we compare the performances of AdWd, FixWd, and FixTW in terms of average sub-query execution time (Table 4), *Q.1 Responsiveness* (Figure 6), and *Q.2 Overhead* (Table 5). We choose the update frequencies from 2s to 20s, and use eight *ProbeQ* queries that cover five hosts from different partition tables, which cover more scenarios in querying the behaviors. The size of events processed by the queries ranges from 3.5 to 20 million. We execute each query using the worker thread from one to five and obtain their average results. Note that RQ2-RQ4 use the same query set.

Table 4 shows that AdWd has the average sub-query execution time closest to the update frequencies (6.3% deviation), while both FixTW and FixWd have much larger average sub-query execution time (119% and 148% deviations, respectively). This is in line with our expectations, as FixWd and FixTW partition the query according to a fixed time window or workload, which cannot reflect the cache mechanism whose event processing rate is much higher than the non-cache counterpart. In Figure 6, we can see that the average sub-query execution time directly affects the quality of responsiveness, where AdWd achieves a highest response rate (on average 84.5%). The reason is that if the average sub-query execution time deviates significantly from the update frequency, the execution of a sub-query typically spans several update cycles and causes the response rate to drop. Table 5 shows the overhead of each partition strategy on the total execution time. When the update frequency is too small (e.g., 2s), AdWd generates much more sub-queries than the other two strategies and thus has a higher overhead. Nonetheless, if the update frequency is greater than 5s, AdWd's performance is comparable to FixWd and FixTW. Overall, AdWd can achieve the best user experiences in terms of progressively showing the partial results and introducing low overhead.

Table 5: *Q.2 Overhead* Comparison among FixTW, FixWd, and AdWd

Strategy	Overhead (%)				
	2s	5s	10s	15s	20s
AdWd (5.0E-4)	53.82	21.99	7.96	4.37	3.79
FixWd	19.23	10.19	7.15	4.13	4.16
FixTW	22.99	9.46	5.29	5.48	3.35

8.4 RQ 3: Impact of Learning Rates

In AdWd, the learning rate controls the fitting degree of the gradient descent algorithm. A learning rate that is too high or too low causes over-fitting or under-fitting that inaccurately partitions the workload and hence makes the average sub-query execution time deviate from the update frequency. We evaluate five different learning rates in the range from 1.0E-5 to 1.0E-3 as shown in Figure 7. The results show that the learning rates around 5.0E-4 achieve the best performance. Note that in this evaluation only queries with more than 20 sub-query executions are used. In this way, we ensure that there are enough data points for us to measure the impact of learning rates more accurately.

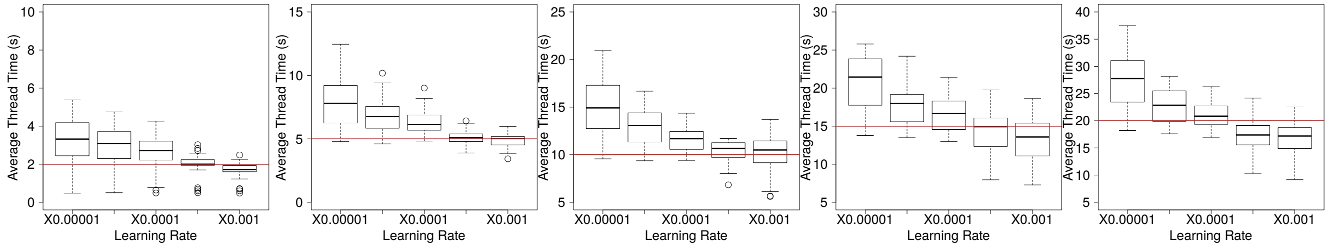
8.5 RQ 4: Impact of Predicted Models

Figure 8 shows the results based on the measured models and the predicted models. Overall, the average sub-query execution time of measured models is slightly closer to the update frequency (variation is also slightly smaller) than the predicted models. However, RQ1 and RQ2 show that such slight imprecision brought by the predicted models do not affect the performance of AdWd significantly and AdWd can still achieve high quality of progressive processing. Moreover, using the predicted models eliminates the need to measure the models for each host, which is not scalable in a large organization, AdWd with the predicted models is a much more practical solution.

9 THREATS TO VALIDITY

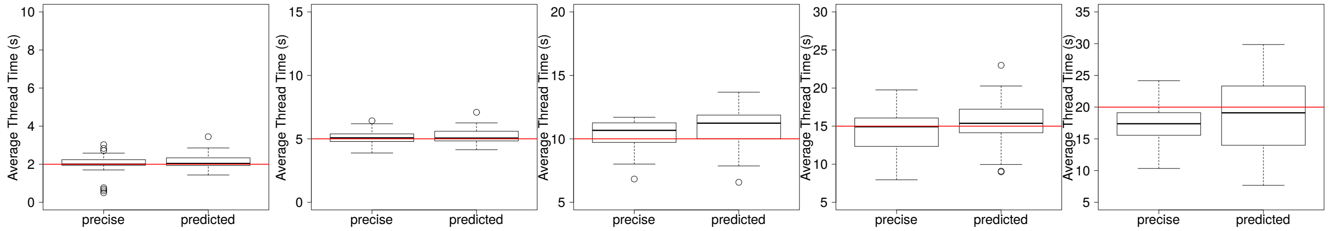
External Validity: Our evaluations are conducted on a limited number of queries, which may introduce threats to the external validity of our results. To mitigate these threats, we choose representative queries that search for suspicious behaviors (shown in this paper), rather than the queries focusing on statistics-related behaviors (e.g., `count(*)`). Also, to understand the impacts of result sizes controlled by various filtering conditions, we further evaluate queries (shown in Figure 9) with different types of filtering conditions: no-filtering, partial-filtering, and complete-filtering, and AdWd achieves a promising performance, similar to the results in Section 8.

Internal Validity: The measurement of a host for the prediction of other hosts may be affected by the background noise and lead to the inaccurate prediction. We mitigate this threat by taking each type of experiments for three times and use their average value for the



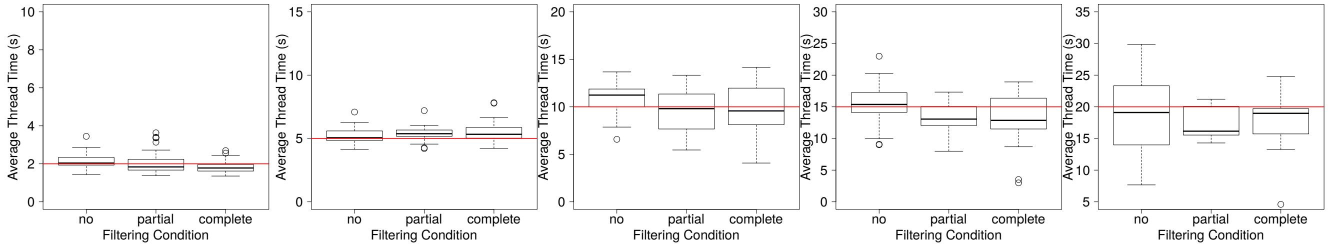
(a) Update frequency: 2s **(b) Update frequency: 5s** **(c) Update frequency: 10s** **(d) Update frequency: 15s** **(e) Update frequency: 20s**

Figure 7: Impacts of different learning rates (1.0E-5, 5.0E-5, 1.0E-4, 5.0E-4, 1.0E-3).



(a) Update frequency: 2s **(b) Update frequency: 5s** **(c) Update frequency: 10s** **(d) Update frequency: 15s** **(e) Update frequency: 20s**

Figure 8: Comparison between precise measurement and prediction when learning rate is 5.0E-4.



(a) Update frequency: 2s **(b) Update frequency: 5s** **(c) Update frequency: 10s** **(d) Update frequency: 15s** **(e) Update frequency: 20s**

Figure 9: Impacts of different filtering conditions when learning rate is 5.0E-4.

prediction. In addition, AdWd is able to adjust the event processing rate dynamically based on the latest execution information, which minimizes the impacts of the prediction imprecision on the overall performance.

10 RELATED WORK

Querying software engineering data: Effective and expressive query languages have been well studied in software repository analysis. Bugzilla [3] enables a user to search bug reports via specifying predicates. JIRA [9] provides an expressive query language JQL [10] to retrieve such reports from revision histories. Kenyon [21] extracts source code change histories from SCM systems to assist software evolution research. TA-RE [30] is an exchange language for mining software changes. SCQL [29] presents a first-order temporal-logic based query language for source control repositories. DebugAdvisor [20] allows a fat query containing natural language description to achieve fast diagnosis. BOA [23] provides a language-based infrastructure for analyzing software repositories. Sun et al. [46] designed a sequential pattern query language to analyze sequential software engineering data. In contrast, *ProbeQ* aims to address efficient system-behavioral queries for OS-level system monitoring. It not only leverages the data characteristics of system events to execute queries in parallel, but also supports progressive-result processing, which is the core challenge for system monitoring.

Program behaviorial query: Research efforts on finding specified program behaviors via query language are remotely related. Brunel et al. [22] proposed an extension to computation tree logic to perform collateral code evolution. Koskinen [31] defined the behavioral profile language by using UML meta-model extensions. Martin et al. [37] presented Program Query Language to verify runtime program behavior. Meredith et al. [39] utilized parametric context free grammars to examine runtime property conformance. Compared to the prior work, which attempted to discover certain program logic, *ProbeQ* targets at a very different problem, which is to efficiently and progressively search system monitoring data in enterprise.

System anomaly detection: Different techniques have been proposed to detect system anomalies in the literature. They can be used in detecting malware [32], internal threat [47], and attack prediction [48]. However, they do not focus on efficiently querying system behaviors or progressively providing execution results. Some related work, such as AIQL [26] and PrioTracker [35] supports timely attack investigation, but can still be stuck when a query takes lots of time to return. Other related work, such as SAQL [25], enables timely anomaly detection directly over the data stream of system monitoring data. Unlike these techniques, *ProbeQ* provides a unified interface to progressively process system-behavioral queries.

Performance optimization for digital forensics queries: Prior efforts have also been made to optimize the query efficiency for security logs. Ning and Xu [40] adopted main memory index structures and query optimization to correlate security alerts. Alink et al. [19] proposed an XML-based approach to manage and search digital forensics traces. Marziale et al. [38] studied the effectiveness of offloading digital forensics tools to a GPU for parallel computation. Winter et al. [51] presented two indexing strategies for robust image hashes. While previous work addressed the query performance in a generic manner, *ProbeQ* designed a domain-specific query language and specific optimization to achieve efficient exploration for fine-grained system monitoring data.

Log management tools: Splunk [15] is a platform that automatically parses application and system logs. It provides a Unix shell-like Search Processing Language to filter log entries based on keywords. Elasticsearch [6] is a distributed search and analytics engine, which is based on Lucene [2] and can be used to search documents. As these tools are not optimized for system monitoring data, they cannot partition queries based on the spatial and temporal properties of data to speed up query execution. Besides, neither supports progressive processing based on adaptive prediction as *ProbeQ* does. Moreover, the shell-like language of Splunk and the search language of Elasticsearch are not intuitive for expressing system events, which are interactions among system resources.

11 CONCLUSION

In this work, we proposed *ProbeQ* to progressively process system-behavioral queries. It has a domain-specific query language that allows system experts to interactively investigate risky system behaviors in an organization by writing concise *ProbeQ* queries and getting partial results progressively over system monitoring data. The query engine of *ProbeQ* partitions a query into sub-queries for parallel execution and reports the partial results periodically based on the specified update frequency. We propose three partition strategies and compare them against two metrics, responsiveness and overhead, on real-world large datasets. Results indicate AdWd outperforms the other two alternatives.

REFERENCES

[1] [n.d.]. ANTLR. http://www.antlr.org/.

[2] [n.d.]. Apache Lucene. https://lucene.apache.org/.

[3] [n.d.]. Bugzilla. http://www.bugzilla.org/.

[4] [n.d.]. Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/.

[5] [n.d.]. DTrace. http://dtrace.org/.

[6] [n.d.]. Elasticsearch. https://www.elastic.co/.

[7] [n.d.]. ETW events in the common language runtime. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx.

[8] [n.d.]. Exploit Database. https://www.exploit-db.com/.

[9] [n.d.]. JIRA. https://www.atlassian.com/software/jira.

[10] [n.d.]. JIRA Query Language. https://confluence.atlassian.com/display/JIRA/Advanced+Searching.

[11] [n.d.]. Kaspersky. www.kaspersky.com/.

[12] [n.d.]. The Linux audit framework. https://www.suse.com/documentation/sles11/book_security/data/part_audit.html.

[13] [n.d.]. McAfee. http://www.mcafee.com/.

[14] [n.d.]. PostgreSQL . http://www.postgresql.org/.

[15] [n.d.]. Splunk. http://www.splunk.com/.

[16] [n.d.]. SQL. http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498.

[17] [n.d.]. Symantec. https://www.symantec.com/.

[18] [n.d.]. Trustwave Global Security Report 2015. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf.

[19] W. Alink, R. A. F. Bhoedjang, P. A. Boncz, and A. P. De Vries. 2006. XIRAF - XML-based Indexing and Querying for Digital Forensics. *Digit. Investig.* 3 (Sept. 2006), 50–58. https://doi.org/10.1016/j.diin.2006.06.016

[20] B. Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. 2009. DebugAdvisor: A Recommender System for Debugging. In *ESEC/FSE '09*. ACM, New York, NY, USA, 373–382. https://doi.org/10.1145/1595696.1595766

[21] Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. 2005. Facilitating Software Evolution Research with Kenyon. In *ESEC/FSE-13*. ACM, New York, NY, USA, 177–186. https://doi.org/10.1145/1081706.1081736

[22] Julien Brunel, Damien Doligez, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 2009. A Foundation for Flow-based Program Matching: Using Temporal Logic and Model Checking. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, NY, USA, 114–126. https://doi.org/10.1145/1480881.1480897

[23] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *ICSE '13*. IEEE Press, Piscataway, NJ, USA, 422–431. http://dl.acm.org/citation.cfm?id=2486788.2486844

[24] Michael Felderer, Emir Tanriverdi, Sarah Löw, and Ruth Breu. 2013. A quality analysis procedure for request data of ERP systems. In *Innovation and Future of Enterprise Information Systems*. Springer, 235–249.

[25] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R. Kulkarni, and Prateek Mittal. 2018. SAQL: A Stream-based Query System for Real-Time Abnormal System Behavior Detection. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*. 639–656.

[26] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. 2018. {AIQL}: Enabling Efficient Attack Investigation from System Monitoring Data. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*. 113–126.

[27] Jiaping Gui, Ding Li, Mian Wan, and William GJ Halfond. 2016. Lightweight measurement and estimation of mobile ad energy consumption. In *Proceedings of the 5th International Workshop on Green and Sustainable Software (GREENS'16)*. ACM, 1–7.

[28] Jiaping Gui, Yue Wu, Chenji Pan, Futai Zou, and Yifei Xie. 2012. Cost based routing in delay tolerant networks. In *2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications-(PIMRC)*. IEEE, 1084–1089.

[29] Abram Hindle and Daniel M. German. 2005. SCQL: A Formal Model and a Query Language for Source Control Repositories. *SIGSOFT Softw. Eng. Notes* 30, 4 (May 2005), 1–5. https://doi.org/10.1145/1082983.1083161

[30] Sunghun Kim, Thomas Zimmermann, Miryung Kim, Ahmed Hassan, Audris Mockus, Tudor Girba, Martin Pinzger, E. James Whitehead, Jr., and Andreas Zeller. 2006. TA-RE: An Exchange Language for Mining Software Repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 22–25. https://doi.org/10.1145/1137983.1137990

[31] Johannes Koskinen. 2008. Behavioral Profiles in Software Engineering: Motivations, Definitions, and Applications. Ph.D. Thesis, Publication 757.

[32] Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. 2010. Accessminer: using system-centric models for malware protection. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 399–412.

[33] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. 2013. LogGC: garbage collecting audit log. In *CCS*.

[34] Ding Li, Shuai Hao, Jiaping Gui, and William GJ Halfond. 2014. An empirical study of the energy consumption of android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 121–130.

[35] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. 2018. Towards a timely causality analysis for enterprise security. In *Proceedings of the 25th Network and Distributed System*

Security Symposium (NDSS). The Internet Society, San Diego, California, USA.

[36] Jian-Guang Lou, Qiang Fu, Shengqi Yang, Ye Xu, and Jiang Li. 2010. Mining Invariants from Console Logs for System Problem Detection. In *USENIX ATC*.

[37] Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding Application Errors and Security Flaws Using PQL: A Program Query Language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 365–383. https://doi.org/10.1145/1094811.1094840

[38] Lodovico Marziale, Golden G. Richard, III, and Vassil Roussev. 2007. Massive Threading: Using GPUs to Increase the Performance of Digital Forensics Tools. *Digit. Investig.* 4 (Sept. 2007), 73–81. https://doi.org/10.1016/j.diin.2007.06.014

[39] P. O. Meredith, D. Jin, F. Chen, and G. Rosu. 2008. Efficient Monitoring of Parametric Context-Free Patterns. In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*. 148–157. https://doi.org/10.1109/ASE.2008.25

[40] P. Ning and D. Xu. 2002. *Adapting Query Optimization Techniques for Efficient Intrusion Alert Correlation.* Technical Report. Raleigh, NC, USA.

[41] Chenji Pan, Jiaping Gui, Jiaju Yan, and Yue Wu. 2012. Markov chain-based routing algorithm in delay-tolerant networks. In *2012 IEEE/IPSJ 12th International Symposium on Applications and the Internet*. IEEE, 161–165.

[42] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from big code. In *ACM SIGPLAN Notices (POPL'15)*, Vol. 50. ACM, 111–124.

[43] Rishabh Singh and Sumit Gulwani. 2015. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification (CAV'15)*. Springer, 398–414.

[44] Saurabh Srivastava, Sumit Gulwani, and JeffreyS. Foster. 2013. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (2013).

[45] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*. IEEE Computer Society, 253–262.

[46] Chengnian Sun, Haidong Zhang, Jian-Guang Lou, Hongyu Zhang, Qiang Wang, Dongmei Zhang, and Siau-Cheng Khoo. 2014. Querying Sequential Software Engineering Data. In *FSE 2014*. ACM, New York, NY, USA, 700–710. https://doi.org/10.1145/2635868.2635902

[47] E Ted, Henry G Goldberg, Alex Memory, William T Young, Brad Rees, Robert Pierce, Daniel Huang, Matthew Reardon, David A Bader, Edmond Chow, et al. 2013. Detecting insider threats in a real corporate database of computer usage activity. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 1393–1401.

[48] Kalyan Veeramachaneni, Ignacio Arnaldo, Vamsi Korrapati, Constantinos Bassias, and Ke Li. 2016. AIˆ 2: training a big data machine to defend. In *2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS)*. IEEE, 49–54.

[49] Mian Wan, Yuchen Jin, Ding Li, Jiaping Gui, Sonal Mahajan, and William GJ Halfond. 2017. Detecting display energy hotspots in Android apps. *Software Testing, Verification and Reliability* 27, 6 (2017), e1635.

[50] F George Wilkie and Barbara A Kitchenham. 2000. Coupling measures and change ripples in C++ application software. *Journal of Systems and Software* 52, 2 (2000), 157–164.

[51] Christian Winter, Martin Steinebach, and York Yannikos. 2014. Fast indexing strategies for robust image hashes. *Digit. Investig.* 11 (May 2014).

[52] Xusheng Xiao, Shi Han, Dongmei Zhang, and Tao Xie. 2013. Context-sensitive delta inference for identifying workload-dependent performance bottlenecks. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA'13)*. ACM, 90–100.

[53] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability* 22, 2 (2012), 67–120.

[54] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterising Logging Practices in Open-Source Software. In *ICSE*.