SCAHunter: Scalable Threat Hunting through Decentralized Hierarchical Monitoring Agent Architecture

Mohiuddin Ahmed[1], Jinpeng Wei[1], and Ehab Al-Shaer[2]

[1] University of North Carolina at Charlotte, NC, USA
[2] Carnegie Mellon University, Pittsburgh, PA, USA
{mahmed27,jwei8}@uncc.edu, ehabalshaer@cmu.edu

Abstract. This paper presents a scalable, dynamic, flexible, and non-intrusive monitoring architecture for threat hunting. The agent architecture detects attack techniques at the agent level, classifies composite and primitive events, and disseminates seen attack techniques or subscribed event information to the upper-level agent or manager. The proposed solution offers improvement over existing approaches for threat hunting by supporting hierarchical event filtering-based monitoring, which improves monitoring scalability. It reduces memory requirement and communication overhead while maintaining the same accuracy of threat hunting in state-of-the-art centralized approaches. We provide a distributed hierarchical agent architecture and an approximation algorithm for near-optimal agent hierarchy generation. We also evaluated the proposed system across three simulated attack use cases built using the MITRE ATT&CK framework and DARPA OpTC attack dataset. The evaluation shows that our proposed approach reduces communication overhead by 43% to 64% and memory usage by 45% to 60% compared with centralized threat hunting approaches.

Keywords: Threat hunting, Intrusion detection, Hierarchical event monitoring

1 Introduction

In recent years, there has been an increase in cyber attacks including advanced persistence threats (APTs) and ransomware [8], and the techniques used by the attacker have reached an unprecedented sophistication [7]. According to Sophos threat report [7], APT and ransomware attacks increased from 37% in 2020 to 78% in 2021. These attacks evade signature-based intrusion detection systems by exploiting the zero-day vulnerability, whitelisted applications and threat emulation tools (Metasploit, Cobalt Strike, Mimikatz). They use a low and slow approach to avoid triggering anomaly detection while working on the attack goals such as exfiltration and encryption. Due to the diverse and sprawling nature of organizational network, and time-consuming nature of attack investigation, attackers can dwell in the system for extended periods. Mandiant reports that the

Accepted in Computing 2023, will be available in ”Springer Lecture Notes in Networks and Systems (LNNS, volume 739)” after 26th September 2023.

global average dwell time of the adversary is 24 days [6]. The damage incurred by the adversary on an organization increases exponentially with increasing dwell time. According to the IBM security threat report [4], data breach damage from ransomware attacks increased from $3.86 million in 2020 to $4.24 million in 2021, and the time to identify and contain the data breach is, on average, 287 days. The high threat detection time indicates that traditional IDS does not make the breakthrough in real-time threat hunting.

Considering the shift in threat actors and unprecedented sophistication in adversary activities, the organization deploys Endpoint Detection and Response (EDR) solutions and System Information and Event Management (SIEM) solutions to record, monitor continuously, and analyze low-level system logs in end-host devices. The EDR solutions detect threats by matching low-level system events against a knowledge base of adversarial TTPs (Tactics, Techniques, and procedures). MITRE ATT&CK framework [1] provides a knowledge base of TTPs developed by domain experts by analyzing real-world APTs. The SIEMs collect low-level system logs or alerts through a collector, sensor, or EDR agent installed in the end-host devices to the manager (central server). A threat hunter uses SIEM to analyze and correlate collected logs to detect adversary activities during the threat hunting process proactively. While such centralized event correlation facilitates causality analysis of attacker activities, it presents the following challenges in threat hunting at a large-scale distributed system:

- **On-demand monitoring:** Existing researches [17, 21, 25] try to monitor everything to give data visibility as much as possible, which is not necessary for detecting a TTP. For example, while an EDR solution tries to detect PowerShell execution of malware, it is not necessary to monitor other data sources (registries, processes, file operations); instead, monitoring the PowerShell command is enough to detect PowerShell execution TTPs [1].
- **Event storage and communication overhead:** The centralized threat hunting process continuously collects monitored logs to the central server, which incurs high memory usage and communication overhead to transfer events to the central server. This approach introduces scalability issues on the monitored network.
- **Efficient event correlation:** To detect attacker TTP, existing solutions and research [20] use single event matching on the end-host devices and generate alerts. Unfortunately, such a single event matching approach generates many false alerts, causing the alert fatigue problem [19] in threat hunting. For example, adversary and benign users can use Windows command shell execution TTP to execute an executable on the system. Detection based on the single event matching will generate many false alerts.

Recent works on threat hunting use causality analysis [17, 21, 24, 26], cyber threat intelligence [25], and MITRE ATT&CK technique detector [25] to reduce mean-time-to-know during the post-breach threat hunting process. These causality analysis approaches incrementally parse low-level audit logs generated by system-level logging tools (e.g., Sysmon, Event tracing for Windows, and auditd) into causal graphs (provenance graphs). The causal graph encodes the dependency between subjects (processes, threads) and objects (files, registries, sockets) to provide the historical context the threat hunter needs to correlate attacker activities and understand alerts. In [5,26], the authors developed detectors or rules for attack techniques and mapped each detector/rule to the MITRE technique. According to a recent survey about EDR solutions by Gartner, all top 10 EDR solutions use MITRE ATT&CK framework to detect adversary behaviors [3]. Such causality analysis is promising for network-wide alert correlation and cyber intelligence.

However, the performance of causality analysis is a limiting factor for real-time threat hunting because of the significant graph construction time ranging from hours to days [21] and the large size of the audit logs (terabytes of logs generated within week [21]). To improve the graph construction time, prior works [24, 26] applied different graph reduction and compression techniques. In [21], the author reduces memory usage during causality analysis by storing the most recent part of the causal graph on the main memory and the unused casual graph on disk. In [17], the authors generate a host-specific causal graph and network-specific causal graph and perform multi-host analysis only if any host-based sub-graph crosses a predefined risk score. Though graph reduction and compression and hierarchical storage reduce memory usage to store monitored events, the causality analysis on the compressed graph has the following limitations. Prior works perform centralized analysis and want to give data visibility as much as possible. Thus, they try to monitor all data sources in the end-host devices, which raises the issue of monitoring scalability and communication overhead (collecting logs on a central server) in threat hunting. Provenance graph expansion for multi-host analysis in Holmes [26] and Steinerlog [17] creates dependency expansion. Graph alignment in multiple hosts [25] increases the threat detection time exponentially with the increase of event logs and network size.

To overcome the limitations (monitoring everything, memory requirement, communication overhead, and many false alerts) of existing tools and research works, we propose a monitoring architecture (Figure 1) using a hierarchical event filtering approach that reduces monitoring load and communication overhead and provides efficient event correlation that can potentially reduce false alerts. The adversary activities follow precedence, meaning that most of the attack techniques have preconditions [11], which are other attack techniques. For example, without performing the initial compromise or execution technique, an attacker will not be able to perform the discovery and command and control technique. Similarly, an attacker cannot perform a collection or exfiltration technique without performing a discovery technique. Following the attack technique association, the SCAHunter provides on-demand monitoring of the data sources corresponding to the monitored attack signature. We provide on-demand monitoring by instrumenting ETW (event tracing for Windows) through ETW API so that the lower-level agents only log signature-specific events. Similarly, auditd for Linux and Endpoint Security for Mac OS can be used for providing signature-specific on-demand monitoring.

Additionally, events/logs can be correlated in the end host devices if monitored events or logs correlate with them. If a set of events from a set of different hosts are required for the correlation of a monitored signature, a middle-level host nearby (in terms of hop-count) the corresponding monitored hosts can be used for the correlation task. Event correlation at the intermediate host will reduce the memory requirement and communication overhead since only the correlated events will be forwarded to the upper-level agents or manager. It will improve scalability and performance by using hierarchical monitoring and distributed correlation while reducing the monitoring intrusiveness.

Finally, hierarchical event filtering will reduce the number of false alert generation problems in the current research works. Single event matching will generate many false alarms because of similarity with benign user activities. However, it is highly unlikely that a sequence of adversary TTPs will match with benign user activities. For example, execution of payload through services.exe or sc.exe (T1569.002) [1] can be used by the benign user; however, remote execution of payload through sc.exe is highly suspicious behavior. Our proposed hierarchical filtering correlates service creation and remote execution in a middle host, thus generating less number of alerts by distributed event correlation. However, generating the agent hierarchy is an NP-hard problem, which we solve using an approximation algorithm (Algorithm 1) based on the geographical host distribution and predefined monitoring capacity of agents. Our proposed approach does not monitor every data source; instead, it monitors what is required to detect the current attack stage and adds a new monitoring task on-demand based on the threat hunting progress.

Contribution. Our first contribution is to provide a distributed hierarchical monitoring agent architecture that optimizes monitoring tasks to reduce resource usage and communication overhead. Our second contribution is to provide an approximation algorithm to generate a near-optimal agent hierarchy, so that event correlation tasks are distributed among the hosts. Our third contribution is to develop an ETW-based agent to monitor signature-specific events so that on-demand monitoring is supported. Our last contribution is to demonstrate the threat hunting process using our proposed agent architecture. We evaluated our proposed architecture using log data generated by running three test scripts provided by Red Canary Atomic Red Team [2], and we created attack signatures for the test scripts following the MITRE ATT&CK technique description during the evaluation. We also evaluated our proposed approach using DARPA OpTC attack dataset [15]. To compare our approach with the existing centralized event monitoring approaches for threat hunting, we also implemented centralized event monitoring using Splunk.

This paper is organized as follows: Section 2 surveys existing research work on threat hunting and intrusion detection system, Section 3 formalizes signature generation and scalable threat hunting with SCAHunter, Section 4 explains SCAHunter by describing each component of the system, attack signature decomposition algorithm, an approximation algorithm to generate near-optimal agent hierarchy used by the agent architecture for subscribe-publish based event monitoring and correlation, and also provides a threat hunting demonstration using the SCAHunter, Section 5 provides implementation details and evaluation with simulated attack use cases and OpTC attack dataset, and Section 6 summarizes our contributions and future research tasks. The remainder of this paper will use logs, alarms, and events interchangeably. It will also interchangeably use monitoring tasks, composite events, and subscribed events.

2 Related works

Causality Analysis. Sleuth [24], DeepHunter [28], Nodoze [20], OmegaLog [22] and CoNAN [31] used provenance graph generation and centralized analysis on the aggregated logs for attack detection and investigation. Holmes [26] uses correlation among information flow to detect an APT, which can be possible only if the threat hunter aggregates events before performing correlation. Domino [32] combines alerts from different NIDS to detect attacks globally, using a single hierarchy level, i.e., manager-agent architecture. Kelifa et al. [16] proposed a misbehavior detection mechanism for wireless sensor network (WSN) based on clustered architecture where a cluster head is selected based on static metrics monitored by the monitoring nodes. Collaborative IDS (CIDS) [30] aggregate alerts from lower-level IDS to manager IDS, and the manager performs graph-based and network-based analysis to detect intrusions. All of those researches aggregate logs in a central server and perform corresponding analysis, which requires monitoring of all events and incurs communication overhead to transfer the generated events to the manager.

In Swift [21], the author reduces memory usage during causality analysis by storing the most recent part of the causal graph on the main memory and the unused casual graph on disk. In [17], the authors generate a host-specific causal graph and network-specific causal graph and perform multi-host analysis only if any host-based sub-graph crosses a predefined risk score. Though graph reduction and compression and hierarchical storage reduce memory usage to store monitored events, the causality analysis on the compressed graph has the following limitations. They try to monitor all data sources in the end-host devices, which raises the issue of monitoring scalability and communication overhead (collecting logs on a central server) in threat hunting. Provenance graph expansion for multi-host analysis in Holmes [26] and Steinerlog [17] creates dependency expansion. Graph alignment in multiple hosts [25] increases the threat detection time exponentially with the increase of event logs and network size.

Event Monitoring. Several centralized and distributed monitoring approaches and tools have been proposed (e.g., [9,10,12–14,23]) in other domains. Although they have various design-specific goals and objectives, they are not scalable for distributed monitoring, lack the flexibility to express the monitoring demands, and require monitoring every data sources. Those proposed approaches either monitor only network traffic [27], apply to network fault diagnosis [10, 18], or maintain a static agent hierarchy [27]. Though hierarchical monitoring systems exist in other domains (fault detection, malicious sensor node detection), they are not suitable for threat hunting since those systems are suitable for a specific use case.

Adversarial Tactics, Techniques and Procedures. MITRE ATT&CK framework [1] published a public knowledge base of TTPs consisting of 24 tactics, 188 techniques, and 379 sub-techniques used by the APT. Every MITRE ATT&CK tactic published is a high-level attacker goal in a specific kill chain phase. Every MITRE ATT&CK technique consists of one or more procedures the attacker can use to achieve a specific goal, whereas each procedure is a unique way to achieve the corresponding goal. MITRE ATT&CK framework also provides 129 APT groups and a subset of the publicly reported technique used by each APT Group. MITRE also provides the data source to monitor to detect a specific attack technique. Holmes [26] uses MITRE ATT&CK TTP to build a set of rules and perform rule matching on the collected logs. This approach will fail if the initial compromise is not detected and they use centralized analysis, which incurs high memory and communication overhead. Additionally, they analyze alerts after generation, whereas SCAHunter reduces the number of alerts generated by hierarchical filtering.

3 Problem Formalization

To formulate the scalable threat hunting with hierarchical agent architecture, we formulate the attack signature and agent hierarchy generation in this section. We formalize attack signatures and event attributes and values, events, event subscription predicate, and event subscription rule (attack signature).

Basic notation for events, attributes, and subscriptions. In distributed event monitoring systems, event producers (application collecting event logs from specific data source) frequently generate events to report aspects of the monitored system state. Let's represent events reported by producers as $E = \{e_1, e_2, .., e_n\}$. Each event e_i consists of a set of attributes $a_{i,j}$ that has assigned values $v_{i,j}$ such that i and j represent the event and attribute indices, respectively. For example, the event e_i that has M attributes is defined as follows: $e_i = \{(a_{i,1}, v_{i,1}), (a_{i,2}, v_{i,2}), .., (a_{i,M}, v_{i,M})\}$. Event consumers may submit multiple subscriptions s_i to request monitoring and reporting of the occurrence of specific event instances or correlation of event instances. The set of consumers' subscriptions can be represented as $S = \{s_1, s_2, .., s_n\}$. Each subscription s_i consists of a *logical expression* on *event attributes*. For example, if a *CA* subscribes with the following, $S_i = \{(pName, "cmd.exe") \wedge (isChild, true)\}$, the *CA* is asking for subscription of all *cmd.exe* processes that are children of another process.

Formulation of Event Fragmentation. By following the formulation of events and attributes, an occurrence of event i at time t is defined as combination (conjunctive) of attribute values as follows:

$$\begin{aligned} e_i^t = (a_{i,1} = v_{i,1}^t) \wedge (a_{i,2} = v_{i,2}^t) \wedge, .., \\ \wedge (a_{i,M} = v_{i,M}^t) = \bigwedge_{j=1}^{M} a_{i,j} = v_{i,j}^t \end{aligned} \tag{1}$$

If we receive a sequence of events of the same event type c within a specified interval T, the event history h_{cl} of event type c during the interval T can be formally represented as follows:

$$h_{cl} = \bigwedge_{l \in L} \bigvee_{t,k \in T} e^t_{l-1} e^k_l \qquad where\ l \neq 1,\ t < k \tag{2}$$

which can be further formalized using the event attributes and values as follows:

$$h_{cl} = \bigwedge_{l \in L} \bigvee_{t,k \in T} \bigwedge_{j \in M} (a^t_{l-1,j} = v^t_{l-1,j})(a^k_{l,j} = v^k_{l,j}) \qquad where\ l \neq 0,\ t < k \tag{3}$$

L is the number of events in the event sequence requested, M is the number of attributes in the event type c. For $l = 1$, Equation 2 and Equation 3 are reduced to the following:

$$h_{c1} = \bigvee_{t \in T} e^t_1 \tag{4}$$

$$h_{c1} = \bigvee_{t \in T} \bigwedge_{j \in M} (a^t_{1,j} = v^t_{1,j}) \tag{5}$$

For instance, a consumer requests for type c event sequence $\{e_1, e_2, e_3\}$ in time interval $T = [1 - 10]$ and the producer reported events e_1 at $t = 2$, e_2 at $t = 6$ and e_3 at $t = 9$, then the equations 2 and 3 will be evaluated true. However, if either any of the three event is not reported or their reporting times are not in order with the sequence, the equations 2 and 3 will be evaluated false.

Formulation of Event Subscription Predicate. An event subscription predicate is a set of *logical predicates* that matches the *attribute values* of the one or more event instance occurrences. Therefore, we define the event predicate as a logical expression defined by the user that will be evaluated based on the attribute values of the event occurrences. Each predicate will be evaluated to `true` if and only if the event attribute values satisfy the predicate logical expression. For example, the predicate $p_1 : (pName, =, "cmd.exe")$ will evaluate to `False` if the event generated is not corresponding to the cmd.exe process, but the predicate, $p_2 : (memAllocated, >, 65)$ will evaluate to `True` if the event is corresponding to a memory allocation and a memory of size greater than 65B is allocated. Therefore, we can formally define the event predicate as follows: $p_{ijk} = (a_{ij}, op, v_k)$, where a_{ij} is the attribute j of event i, v_k is any value of type integer or string that can match the value of the attribute, and *op* is a logical operator (such as $=, <, >, \leq, \geq$) or set operator (such as $\supset, \not\subset, \subseteq, \not\subseteq$, etc). Semantically, if *op* is $=$, then $p_{ijk} \Leftrightarrow a_{ij} = v_k$.

A predicate can specify a relationship between (same or different) attributes of same or different events to detect the occurrence of an *event correlation.* Formally, a predicate that defines a relationship between attribute k in two different events, i and j, can be specified as follows: $p_{ijk} = (a_{ik}, op, a_{jk})$, where a_{ik}, and a_{jk} is the attribute k of events i and j, respectively.

Formulation of Event Subscription Rule. By following the formulation of event subscription predicates, a user can define an Event Subscription Rule (ESR) as the logical Boolean expression of a conjunctive normal form (CNF) using multiple predicates to match an attack signature (subscription) occurrence, S_i, as follows:

$$\begin{aligned} ESR(e_1,..,e_n) = (p(e_i) \lor .. \lor p(e_j)) \land (p(e_i) \\ \lor .. \lor p(e_j)) \land .. \land (p(e_i) \lor .. \lor p(e_j)) \end{aligned} \tag{6}$$

which can be formalized in the concise format as follows:

$$ESR(e_1,..,e_n) = \bigwedge_{i \in \mathbb{N}: i \leq n} \; \bigvee_{j \in \mathbb{N}: j \leq n} p(e_j) \tag{7}$$

Therefore, given the attack signature or subscription request S_i as ESR and a network topology, the goal of the hierarchical monitoring architecture is to generate optimal agent hierarchy such that communication cost and memory usage is reduced, and task is distributed among the agents. The optimal agent hierarchy generation problem can be formalized as follows-

$$\begin{aligned} minimize \sum agent\_count(S_i), \\ minimize \sum Data\_Source\_Count(S_i) \end{aligned} \tag{8}$$

subject to

$$monitoring\_cost < threshold$$

To develop the hierarchical monitoring architecture, we have to address the following problems: 1) Given the subscription request and network topology, generate the optimal number of middle-level agents, maintaining monitoring capacity constraints of each agent, 2) Given the subscription request, determine the optimal number of data sources to monitor, 3) develop monitoring agent for the specific data source. The following section provides the agent architecture, optimal agent hierarchy, and data source monitor generation.

4 Distributed Hierarchical Monitoring Agent Architecture Overview

In this section, we describe each component of SCAHunter as shown in Figure 1. Our SCAHunter consists of three types of agents- console agent or manager (CA), composite event detector agent (CEDA), event filtering agent (EFA), and data sources to monitor. The user of this agent architecture (or a cyber threat hunter) provides a single event or group of events or correlated event subscription requests (i.e., attack signatures) using the CA. The CA decomposes the subscription request based on the formalization provided in Section 3. The CA also generates the required number of CEDAs using the decomposed subscription request.

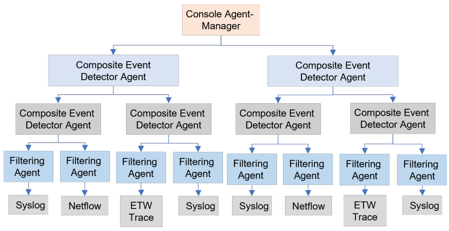

Fig. 1. Distributed Hierarchical Monitoring Agent Architecture

It determines appropriate EFAs, and agent hierarchy such that communication overhead is minimum and subscription task monitoring is distributed across the hosts in the network. Since the optimal agent hierarchy generation is an NP-hard problem, our CA uses an approximation algorithm to generate a near-optimal agent hierarchy. Then, the CA sends the decomposed event subscription requests to the corresponding CEDAs and EFAs through a dedicated configuration channel. Upon receiving a subscription request from the CA, an EFA will start monitoring corresponding data sources. Whenever the EFA detects a subscribed event, it publishes detected events to the upper-level (parent) CEDA through task-specific channels as alerts containing task id and event details. The CEDA or EFA also replies to the CA through a dedicated configuration channel to activate the next monitoring task and deactivate the detected monitoring task to detect a subscription request while supporting on-demand monitoring.

4.1 Console Agent (CA) or Manager

The Console Agent takes an attack signature as input (ESR) from the cyber threat hunter at the beginning of the threat hunting process. This agent's first task is to decompose the subscription request received from the threat hunter using the formalization described in Section 3. The CA's second task is to determine how many CEDA levels to generate and how many CEDAs to develop and where to place those generated CEDAs based on the decomposed event subscription request, the data source to monitor, and the location of end-host devices. It also needs to determine EFA(s) where the event subscription request will be sent. After determining the CEDAs and EFAs, the CA generates the corresponding CEDA and configures them. After a decision as a reply from CEDA is available, it informs the threat hunter about the detected TTPs or attack techniques.

Since the number of generated CEDAs will impact resource usage and communication overhead within the CA and the network, appropriate CEDA number, level, and place are required. We can formulate the determination of the proper number of CEDAs required as *Generate a minimum number of CEDAs that can cover all required EFAs for serving the event subscription request.* We can formulate this problem as a set-cover problem that is NP-complete. One way to solve this problem is to use heuristics on each CEDA or EFA's predefined monitoring capacity and the geographical distribution of end-host devices. Specifically, we propose an approximation algorithm (AHG in Algorithm 1) based on these heuristics.

4.2 Composite Event Detector Agent (CEDA)

Composite event detector agent reduces network communication overhead (traffic flow) between the CA and EFAs by replying to the upper-level CEDA or the CA if a monitoring task or subscribed event is detected. The hierarchical agent architecture can have multiple levels of CEDAs. If the CA creates one level of CEDAs, each CEDA's child is an EFA, and its parent is the CA. On the other hand, if the CA creates two levels of CEDAs, the child of lower-level CEDA is an EFA, and the parent of it is a higher-level CEDA, and the parent of the higher-level CEDA will be the CA.

4.3 Event Filtering Agent (EFA)

The event filtering agent is the lowest level of the agents in the SCAHunter. It monitors different data sources for events requested in the received event subscription request. These agents are static: we generate them initially and continue to work until closed or subscription requests are deleted. MITRE ATT&CK framework [1] provides around 38 different data sources to monitor for detecting different attack TTPs. Thus, to detect all attack techniques and sub-techniques provided by the MITRE ATT&CK framework, we have to develop less than 38 different event filtering agents. We developed EFAs to monitor the following data sources: ETW trace, Netmon, and Sysmon.

4.4 Agent Communication Protocol

Since every agent in SCAHunter may consume event logs or alerts from multiple other agents or data sources, SCAHunter uses a publish-subscribe communication pattern for group communication among the agents. The CA, CEDA, and EFA agents may work as producers and consumers. The CA configures CEDA and EFA agents by publishing configuration info to the corresponding agents through the configuration channel. It also consumes alerts from lower-level CEDA or EFA agents through task-specific channels. CEDA and EFA publish detected monitoring tasks as alerts to the upper-level CEDA or CA through task-specific channels. We use a publish-subscribe communication pattern to facilitate the above-mentioned group communication among the agents. Though the proposed agent architecture employs a hierarchical structure for event monitoring and detection, it is a virtual hierarchy constructed by CEDAs and EFAs' group communication over publish-subscribe communication protocols. Using publish-subscribe communication protocols in agent communication will improve agent hierarchys' robustness in agent failures or network partitioning.

Algorithm 1 Agent Hierarchy Generation Algorithm **AHG**(S, MT, A)

Input: attack signature S, monitoring task list MT, agent hierarchy A
Output: agent hierarchy A

```
if $size(MT) == 1$ then
    return $A \cup \{MT\}$
end if
for all pair$(m_i, m_j)$ where $m_i \in MT, m_j \in MT, m_i! = m_j$ do
    $score, correlationScore \leftarrow$ correlation between $m_i$ and $m_j$, (score, $m_i, m_j$)
end for
sort correlationScore based on $score$
$covered \leftarrow \phi$, $capacity \leftarrow \phi$, $ind \leftarrow 0$, $cluster \leftarrow \phi$, $index \leftarrow 0$
for all $(score, m_i, m_j) \in correlationScore$ do
    if $m_i \in covered$ and $m_j \in covered$ then
        continue
    else if $m_i \in covered$ then
        $index, m_i \leftarrow$ find cluster index containing $m_i$, None
    else if $m_j \in covered$ then
        $index, m_j \leftarrow$ find cluster index containing $m_j$, None
    end if
    if $m_i! = None$ and $m_j! = None$ then
        $cluster_{ind}, capacity_{ind}, covered \leftarrow m_i \cup m_j, 2, covered \cup m_i \cup m_j$
    else if $m_i! = None$ and $capacity_{ind} + 1 <= threshold$ then
        $cluster_{ind}, capacity_{ind}, covered \leftarrow cluster_{ind} \cup m_i, capacity_{ind} + 1, covered \cup m_i$
    else if $m_j! = None$ and $capacity_{ind} + 1 <= threshold$ then
        $cluster_{ind}, capacity_{ind}, covered \leftarrow cluster_{ind} \cup m_j, capacity_{ind} + 1, covered \cup m_j$
    else if $m_i! = None$ then
        $cluster_{ind}, capacity_{ind}, covered \leftarrow m_i, 1, covered \cup m_i$
    else
        $cluster_{ind}, capacity_{ind}, covered \leftarrow m_j, 1, covered \cup m_j$
    end if
    $ind \leftarrow ind + 1$
end for
$A \leftarrow A \cup \{cluster\}$
return AHG(S,cluster, A)
```

4.5 ESR Decomposition and Agent Hierarchy Generation.

A cyber threat hunter provides attack signatures as ESR to the CA for the subscription of related events. However, the CA needs to decompose the provided ESR to generate the required CEDAs and determine corresponding EFAs. The first step in the signature decomposition process is to determine primitive events. Since any predicate containing a relationship between event attribute and specific value is a primitive event predicate (PE_i) to monitor, determining primitive events from signature is trivial. Similarly, for any predicate containing a relationship among multiple identical or different event attributes, we can consider such predicate as a composite event predicate (CE_i) or correlation task. Primitive event decomposition Algorithm 2 ($PED(S)$) takes an attack signature as a monitoring task and converts it to a predicate list (line 1). Next, the PED algorithm extracts primitive events and correlation tasks based on the predicate's event attributes, value, and relationship (lines 3-9).

Given the decomposed primitive event predicates, PE_i, and correlation tasks (composite event predicate CE_i) to monitor, the agent hierarchy generation Algorithm 1, AHG (MT, A), determines the intermediate agents (CEDAs) to monitor correlation among different primitive events. If $MT_{i,j}$ denotes the monitoring task of agent i at level j, we can formalize the intermediate agent generation with the following recurrence relation:

$$MT_{i,j} = \begin{cases} \bigcup_{k \in corrSet} PE_k, & \text{if } j == 0 \\ \bigcup_{k \in corrSet} MT_{k,j-1}, & \text{otherwise} \end{cases} \tag{9}$$

Where $corrSet$ is a set of monitoring tasks of lower-level monitoring agents which are highly correlated either by event attribute name or value. For the EFA agent, each primitive event predicate, PE_i, is a monitoring task of the corresponding agent. Therefore, we can determine the monitoring task for agent i at level j by clustering monitoring tasks of a group of agents at level $j-1$ based on the correlation among the monitoring tasks of the agent at level $j-1$. The AHG algorithm calculates the correlation score for every pair of monitoring tasks in MT (lines 4-6) and sorts the estimated correlation score (line 7). Next, the AHG algorithm greedily selects the pair with the highest correlation score.

It adds each monitoring task in the pair to a cluster if it is not present in the existing cluster yet (e.g., for the pair (m_i, m_j), if m_j is present in an existing cluster but m_i is not, the algorithm tries to add m_i to the cluster of m_j) (lines 17-22). The AHG algorithm creates a new cluster if the existing cluster's size has reached a threshold (lines 23-27). This way, the algorithm greedily clusters all monitoring tasks with a high correlation. In the end, each cluster represents the CEDA of the current agent level. The generated clusters for the current level will be the monitoring tasks for the next level (line 31). The above process continues until only one monitoring task remains (line 1). Since every CEDA is performing a part of the monitoring task, the monitoring task is distributed across the hosts in the network. Moreover, since monitoring tasks in a CEDA have a high correlation, all the predicates related to those monitoring tasks can be calculated with optimal communication; thus, the SCAHunter reduces overall communication overhead. This agent hierarchy generation algorithm starts by taking all primitive events as monitoring tasks.

Algorithm 2 Primitive Event Decomposition Algorithm **PED**(S)

Input: attack signature, S
Output: primitive event to monitor, MT and correlation task, CT

```
$predicates \leftarrow$ convert signature $S$ to predicate list
$MT, CT \leftarrow \phi, \phi$
for all $p_i \in$ predicates do
    if $p_i$ contains relation among event attribute and value then
        $MT \leftarrow MT \cup \{p_i\}$
    else
        $CT \leftarrow CT \cup \{p_i\}$
    end if
end for
return $MT \cup CT$
```

The agent hierarchy generation starts by calling $AHG(S, PED(S), \phi)$, or Algorithm 1, with the attack signature S to monitor, the decomposed monitoring tasks calculated by PED(S), or Algorithm 2, in linear time, and an empty cluster list. Algorithm 2 decomposes the given attack signature S to monitor in linear time. Since the AHG algorithm runs recursively to cluster primitive tasks, each recursive call will reduce the problem size to $N, N/c, N/c^2, N/c^3, ..., 1$, where c is the monitoring capacity of each CEDA agent and N is the number of decomposed monitoring tasks (Algorithm 2). Thus, there will be $log_c N$ number of recursive calls to the AHG algorithm. In each of those recursive calls, there will be $N^2 + N^2 log_2 N^2 + N^2 = 2N^2 + 2N^2 log_2 N \approx O(N^2 log_2 N)$ work in the worst case. Thus the run time of the AHG algorithm is $log_c N \times O(N^2 log N) \approx log_2 c \times log_c N \times O(N^2 log N) \approx log_2 N \times O(N^2 log_2 N) = O(N log_2 N)^2$. Moreover, the AHG algorithm uses $O(N^2)$ additional memory to generate the agent hierarchy.

4.6 Distributed Hierarchical Monitoring Use Case Demonstration

This section will demonstrate threat hunting using SCAHunter. We use a hypothetical attack signature $(p_1 \wedge p_2 \wedge p_3)$ that contains three primitive event predicates $(p_1, p_2$, and $p_3)$ to monitor. We demonstrate signature decomposition, hierarchy generation, and threat hunting in a network consisting of three hosts $(H_1, H_2$, and $H_3)$. We also assume that each generated CEDA agent has a monitoring capacity of two and each PE_i (primitive event predicate) corresponds to separate data sources. Thus, we can say that each PE_i is a monitoring task of EFA. At a high level of abstraction, our signature decomposition Algorithm 2 groups event predicates based on event type and host to generate PE_i, which is the monitoring task of EFA. Next, the AHG algorithm takes all generated PEs as monitoring tasks $(MT_{i,j} \in MT_j)$ and generates monitoring tasks $(MT_{k,j-1} \in MT_{j-1})$ for higher-level agents by clustering $MT_{i,j}$ based on the similarity among $MT_{i,j}$. For our signature, $MT_0 = \{p_1, p_2, p_3\}$.

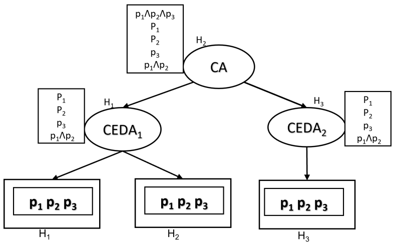

Fig. 2. Generated agent hierarchy

The AHG Algorithm 1 calculates correlation score for each pair of $MT_{i,j}$. For our case, let's assume p_1 has a higher correlation with p_2 than p_3. Thus, AHG algorithm clusters p_1 and p_2 together first. Since each agent has a monitoring capacity of 2, p_3 requires a separate cluster. Thus, $MT_1 = \{p_1 \wedge p_2, p_3\}$. At the next step, AHG algorithm 1 recursively generates monitoring tasks for the next level. In our case, MT_1 is the monitoring task and AHG algorithm clusters each $MT_{1,j}$. Though $MT_{1,1}$ and $MT_{1,2}$ does not have any similarity between them, we will cluster them together because monitoring capacity allows for up to 2 monitoring tasks. Thus, $MT_3 = \{p_1 \wedge p_2 \wedge p_3\}$ which is the attack signature to monitor. At this step, Algorithm 2 stops and returns the monitoring tasks for each level. The monitoring tasks for each CEDA will be $MT = \{p_1, p_2, p_3, p_1 \wedge p_2\}$. The overall result of our algorithms is illustrated in Figure 2.

The hierarchy generation algorithm will determine how many CEDAs to generate and where to place them and what are the communication channels. Since only one monitoring task corresponding to an attack signature will be active at a time, in the beginning only p_1 will be active in all CEDAs and p_1 needs to be monitored in all hosts. Since our use case has three hosts and each agent has a monitoring capacity of 2, we can generate $CEDA_1$ to monitor MR in H_1 and H_2 and $CEDA_2$ to monitor MR in H_3. We launch $CEDA_1$ in H_1 or H_2 and $CEDA_2$ in H_3. $CEDA_1$ communicates with $\{H_1, H_2\}$ using task specific channels $(c1_{p1}, c1_{p2}, c1_{p3})$, and $CEDA_2$ communicates with H_3 using task specific channels $(c2_{p1}, c2_{p2}, c2_{p3})$ and the CA communicates with H_1, H_2, H_3 through configuration channel and with $CEDA_1, CEDA_2$ through task specific channels $(ca_{p1}, ca_{p2}, ca_{p3})$. Now, the CA will activate first monitoring task in $CEDA_1$ and $CEDA_2$ and start the appropriate EFA agent in all hosts to monitor specific ETW provider corresponding to p_1. Suppose $CEDA_1$ detects p_1, it will notify the CA by sending a detection alert containing detected MR_i through a configuration channel. If monitoring of p_1 is not required for any other signatures, the CA will stop monitoring p_1 in $CEDA_1$ and $CEDA_2$. It will also stop consuming event logs from data source corresponding to p_1 in all hosts by removing corresponding ETW providers from active session of EFAs. The CA also activates the next monitoring task, p_2, in all CEDAs through configuration channels. It also activates EFA corresponding to p_2 if it is not already active. This process continues until all p_1, p_2 and p_3 is detected. This threat hunting approach through SCAHunter reduces event storage requirements through on-demand monitoring, and the hierarchical structure of the agent communication ensures optimal communication overhead.

5 Implementation and Evaluation

We implemented the SCAHunter using Python in a Windows 10 machine with 64GB RAM and multiple VM settings (two hosts, three hosts, four hosts, and five hosts). We simulated attacker and benign user activities on all the VM settings. This section provides implementation details of the CA, CEDAs, and EFA agents and evaluates the SCAHunter using three simulated use cases and DARPA OpTC attack dataset [15].

5.1 Implementation Details

We implemented the console agent (CA) using Python 3. The CA consists of a signature decomposition and agent hierarchy generation module, configuration module, signature filtering module, and communication module. Given an attack signature in ESR format and network topology as an adjacency list, the signature decomposition module decomposes ESR into primitive predicates and composite predicates. We described the decomposition process in Section 4.5. It also generates agent hierarchy based on an approximation algorithm 1. After agent hierarchy generation, the configuration module configures all required agents and monitoring tasks in the corresponding hosts. It also configures the publish-subscribe communication protocol so that agents can communicate with a group of required agents. A CA agent works as both a subscriber (consumer) and publisher (producer). And CA consumes (subscribe) alerts corresponding to monitoring tasks from lower-level CEDAs or EFAs. The signature filtering module stores monitored tasks in a map data structure to maintain constant time filtering. We implemented the composite event detector agent (CEDA) using python 3. The CEDA consists of a filtering module and communication module. We implemented the filtering module using the same logic as the filtering module of CA while maintaining constant time filtering. CEDAs subscribe (consume) monitored events from lower-level CEDAs or EFAs. It also subscribes for configuration info from the CA.

We implemented the primitive event filtering agent (EFA) using Python 3 and CPython. The EFA consists of an event filtering module, event consumer module, and communication module (Figure 3). The event filtering module stores the decomposed monitoring task in a map (python equivalent dictionary) data structure to do the event filtering tasks in constant time. The event consumer module consumes data from different data sources. To ingest data from ETW trace, we use various etw providers. Windows ETW trace provides around 1000 ETW providers to produce event logs corresponding to different data sources. It also provides wintrace API to collect and manage event logs from ETW. ETW trace program consists of a controller, provider, and consumer. Our EFA agent creates an ETW trace session with a specific ETW provider based on the data source required to monitor. The EFA consumer module consumes event data from the created session through the win32 event tracing API. Interaction with ETW event tracing API is implemented using CPython.

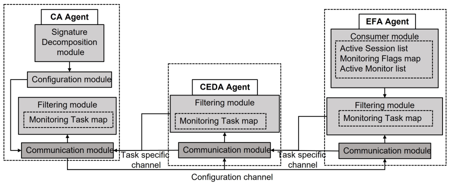

Fig. 3. Implementation of distributed hierarchical monitoring agent architecture

Moreover, we provided ways to monitor specific events from an ETW provider through win32 event tracing API. For example, the Sysmon provider generates event logs corresponding to different data sources such as process creation, network traffic, file access, and registry access. Our EFA agent can collect process creation, network connection, or file access events through event tracing API. Our EFA agent also maintains a list of active ETW sessions. If a new monitoring request comes from the CA, the EFA agent creates a new ETW session if required event logs can not be collected from the currently active session. We implemented Sysmon and Netmon EFA agents and ingested data from ETW Windows Sysmon provider, Winsock providers, and Powershell provider. We also implemented the publish/subscribe communication protocols using a message broker library RabbitMQ [29]. The CA subscribes for monitored signature and publishes configuration info and monitoring tasks to the CEDAs or EFAs using RabbitMQ. The CEDA uses RabbitMQ to ingest alerts for the monitored jobs from the lower-level CEDAs or EFAs and publishes the detected monitoring tasks as alerts to the upper-level CEDA or CA. The EFA agent publishes collected event logs from ETW trace sessions to the upper-level CEDA or CA using RabbitMQ.

To compare our proposed event filtering approach with the existing centralized approaches, we also implemented centralized monitoring using Splunk.

5.2 Evaluation

We evaluate our approach with three simulated attack scenarios (multiple attacker activity sequences expressed in MITRE ATT&CK techniques) and one public attack dataset, the OpTC (Operationally Transparent Cyber) dataset released by DARPA [15]. We use Atomic Red Team's atomic tests [2] to generate attacker activities in a benign user session. The atomic test case includes scripts to execute multiple procedures of each MITRE ATT&CK technique. We generate scripts for a sequence of attacker activities using the MITRE ATT&CK technique by cascading scripts of multiple techniques. In usecase 1, we execute a payload/malware file using Technique PowerShell (T1059.001) [1]. The executed payload performs the account discovery process using technique T1087 [1], and in the end, the malware process executes a scheduled task using technique T1053 [1]. Usecase 2 executes techniques T1189, T1035, T1021.001, T1119, and T1048 [1] using scripts from the Atomic Red Team; we provide the signature for this use case in Appendix A. Usecase 3 uses all the techniques in Usecase 2 plus one defense evasion technique, masquerading (T1036) [1].

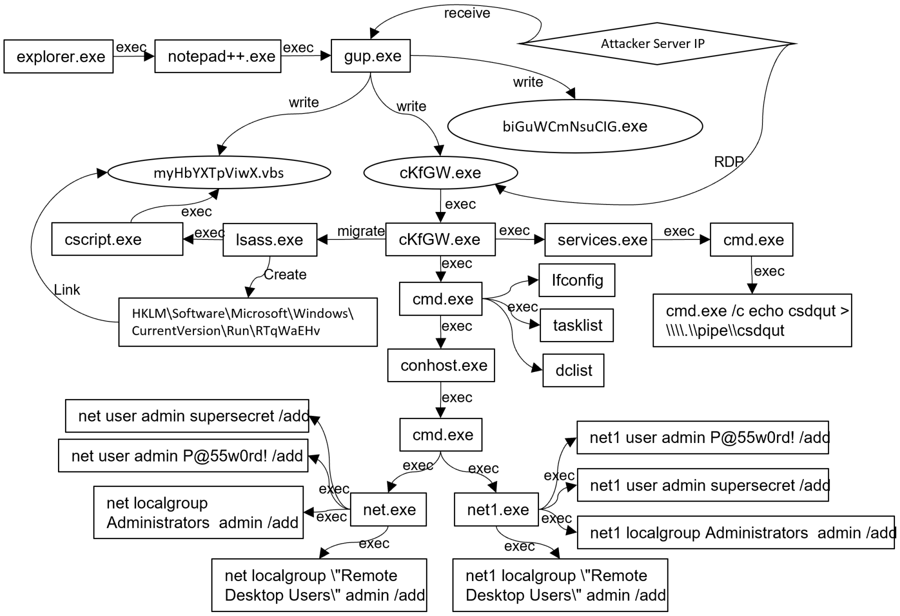
receive
Attacker Server IP
explorer.exe
exec
notepad++.exe
exec
gup.exe
write
write
write
biGuWCmNsuClG.exe
myHbYXTpViwX.vbs
cKfGW.exe
RDP
exec
cscript.exe
exec
exec
lsass.exe
migrate
cKfGW.exe
exec
services.exe
exec
cmd.exe
exec
Create
exec
Ifconfig
Link
cmd.exe
exec
tasklist
HKLM\Software\Microsoft\Windows\CurrentVersion\Run\RTqWaEHv
cmd.exe /c echo csdqut > \\\\.\\pipe\\csdqut
exec
dclist
conhost.exe
exec
net user admin supersecret /add
cmd.exe
net user admin P@55w0rd! /add
net1 user admin P@55w0rd! /add
exec
exec
exec
exec
net1 user admin supersecret /add
net localgroup Administrators admin /add
exec
net.exe
net1.exe
exec
net1 localgroup Administrators admin /add
exec
exec
net localgroup \"Remote Desktop Users\" admin /add
net1 localgroup \"Remote Desktop Users\" admin /add

Fig. 4. Low-level Attacker Activities in OpTC Dataset

The OpTC dataset [15] contains 287 GB event logs collected on five hundred to one thousand hosts over three days. During these three days, the red team performed three APT scenarios. Our evaluation uses one scenario that consists of around 70 GB of log data corresponding to 1.3 billion events. In this APT scenario, the red team updated *notepad++*, which downloads and executes a Meterpreter payload. This payload execution obtains system access through named pipe impersonation. It performs several discovery techniques: system info, installed applications, domain controller, and network shares. It also performs *ARP scanning* to discover neighbor hosts in the same network. Then, the Meterpreter process creates a *Run* registry key and sets the value of the registry key to a downloaded Meterpreter module to establish persistence. Later, this process adds *administrator* and *admin* accounts to the *administrators* and *RDP group*. Finally, the attacker RDPed to the compromised machine from the attacker's server. We develop attack signatures corresponding to each red team activity (Appendix A) and use those signatures to evaluate SCAHunter. The discovered low-level attacker activities are shown in Figure 4. In this figure, nodes represent system entities such as processes, registry keys, commands (rectangles), files (ovals), and sockets (diamonds), and edges corresponding to the attacker's actions represent the information flow and causality. To evaluate the proposed threat hunting approach, we try to answer the following three questions:

1. Does threat hunting with distributed hierarchical monitoring reduce communication overhead and storage requirement compared

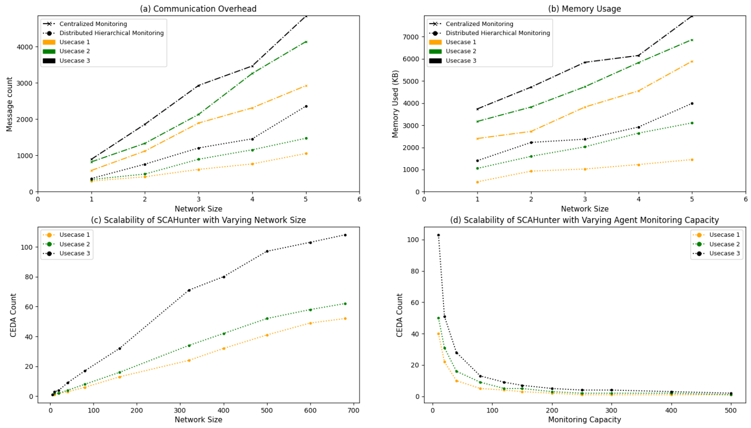

Fig. 5. Performance and Scalability Evaluation of hierarchical monitoring agent architecture based on Simulated Attack Use Cases

to the state-of-the-art centralized threat hunting approaches? Since a centralized approach tries to monitor everything and aggregate logs in a central server, it has to store all of the logs generated by the EFA agents. In contrast, a distributed monitoring approach records logs only for a single monitor corresponding to the current active monitoring task. There can be one active monitoring tasks for a single attack signature. Moreover, CEDAs corresponding to the monitoring tasks will store results of only detected monitoring tasks for the currently active monitor. As a result, the distributed hierarchical monitoring records fewer events than the centralized approach, which is evident in Figure 5 (a) for simulated attack use cases and Figure 6 (a) for OpTC attack dataset.

Specifically, to confirm the decrease in communication overhead, we measure the number of exchanged messages (alerts, configuration messages) among the agents in three simulated attack use cases for both centralized and proposed distributed approaches. During this evaluation, we maintain a fixed monitoring capacity of 12 for each agent while varying the network size from one to five. For the centralized monitoring approach, we collected event logs from only data sources related to the attack signature. As we can see in Figure 5 (a), using the SCAHunter approach, the number of exchanged messages among agents (event count) decreases around 49.74% to 66.9%, 58.04% to 64.58%, and 51.3% to 59.35% for use case 1, use case 2, and use case 3, respectively, compared with the centralized approach. We can also see that the decrease becomes more and more significant as the network size increases. Similarly, we measure the number of exchanged messages among agents using the OpTC attack dataset for centralized and proposed distributed approaches. For the centralized monitoring approach, we collected logs from the following data sources: file creation activity, registry key creation activity, process creation activity, and RDP network connection creation activity. During this evaluation, we maintain a fixed monitoring capacity of 12 for each agent while varying the network size from one to 16. In our proposed approach, the number of exchanged messages among agents decreases by around 43.7% - 64.8% from the centralized approach, as shown in Figure 6 (a). Since our proposed approach forwards event logs corresponding to the current active monitoring task, performs distributed event filtering by event correlation in CEDA agents, and forwards only the detected correlated events to the upper-level agents or CA, the communication overhead is lower compared to the centralized approach as shown in Figure 5 (a) and Figure 6 (a). To show the decrease in memory usage overhead, we measure the memory used by the agents (CA, CEDA, EFA) to store the events corresponding to the detected monitoring tasks in the centralized approach and our proposed hierarchical approach. The amount of memory used is the number of event logs times the average size of each event log, which we estimate as 500 bytes based on event logs from the ETW Sysmon provider. In the proposed hierarchical monitoring approach, the total memory usage decreases around 65.9% to 81.3%, 54.6% to 66.85%, and 49.75% to 62.48% for simulated attack use case 1, use case 2, and use case 3, respectively, from the centralized monitoring approach as shown in Figure 5 (b). Similarly, the proposed SCAHunter approach decreases total memory usage by 45.4% to 60.1% for the OpTC attack dataset compared with the centralized monitoring approach, as shown in Figure 6 (b). Since our proposed approach stores only the event logs related to the monitoring tasks and only a subset of the monitoring

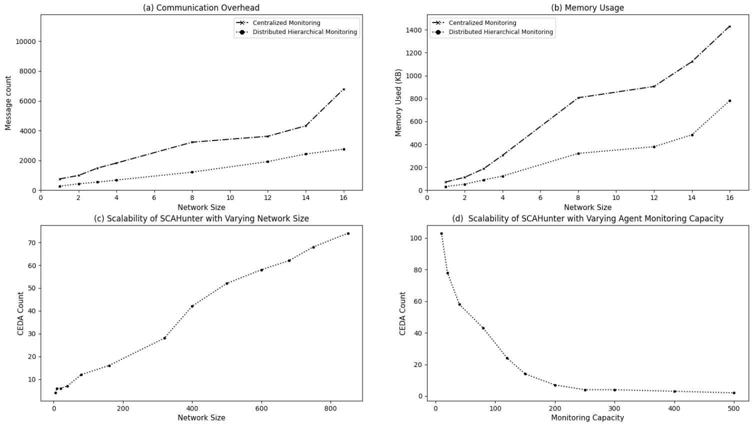

Fig. 6. Performance and Scalability Evaluation of hierarchical monitoring agent architecture based on OpTC Attack Dataset

tasks are active at any moment of threat hunting, the storage required is less for the hierarchical monitoring approach than the centralized approach, as shown in Figure 5 (b) and Figure 6 (b).

2. Can threat hunting with distributed hierarchical monitoring detect multi-step attacker activities with the accuracy of state-of-the-art centralized threat hunting approaches? Since the centralized approach monitors all required data sources for an attack signature and aggregates generated logs in a central manager, the accuracy of the centralized approach depends on the quality of the attack signature. On the other hand, since the distributed approach decomposes attack signatures such that monitoring tasks are distributed among hosts, an efficient event correlation is required in addition to the signature quality. For the DARPA OpTC attack dataset, both the centralized and SCAHunter threat hunting approaches can detect low-level attacker activities, shown in Figure 4. For example, the signature $a.Operation == NewFileWrite \land x.event\_id == process\_creation \land x.imagePath == a.newFile-Path$ is used to detect Meterpreter's payload downloading activity.

For the simulated attack use cases, the centralized approach can detect all three attack signatures. However, the distributed hierarchical monitoring approach initially missed one of the attack signatures. Further investigation of low-level logs reveals that this is due to the on-demand nature of our proposed approach, which does not monitor everything, instead, it activates a monitoring task only if all the previous monitoring tasks in the attack signature are detected. Therefore, there is a time lag in our approach, which we can mitigate by using a memory buffer. In our new implementation, we keep event logs of all required data sources of the current signature for thirty seconds. This user-defined period of log recording is introduced to cover the time lag between detecting one monitoring task and activating the next monitoring task. After introducing this log buffer, our approach detects previously missed attack signatures.

3. Is the proposed hierarchical monitoring architecture scalable? We examine the scalability of threat hunting with the hierarchical monitoring approach as the size of the network topology and the monitoring capacity increase. We measure the number of required CEDAs for the three simulated attack use cases and the OpTC dataset in a simulated network whose size varies from 1 to 800 hosts. In Figure 5 (c), we see that with a 20% to 25% increase in network size (e.g., from 320 to 400), the number of required CEDAs increases 19.5% to 33.3%, 12.4% to 28%, and 23% to 32.73% respectively, for use case 1, use case 2, and use case 3. Similarly, In Figure 6 (c), we see that with a 20% to 25% increase in network size, the required number of CEDAs increases by 8.3% to 23.8% for OpTC attack dataset. From Figure 5 (c) and Figure 6 (c), we see that the number of required CEDAs grows linearly with the network size and tends to flatten out. In all these experiments, we set the monitoring capacity to 12.

We also measure the number of required CEDAs for the three simulated attack use cases and the DARPA OpTC attack dataset with a varying monitoring capacity (from 1 to 600) in a fixed simulated network of size 50. With a 25% increase in the agent monitoring capacity, the number of required CEDAs decreases 11.3% to 19.6%, 12.4% to 28%, and 17.1% to 32.9% respectively, for use case 1, use case 2, and use case 3, as shown in Figure 5 (d). Similarly, In Figure 6 (d), we see that with a 25% increase in the agent monitoring capacity, the required CEDA decreases by 4.7% to 38.4% for OpTC attack dataset. The number of required CEDAs is inversely co-related with monitoring capacity, as shown in Figure 5 (d) and Figure 6 (d).

Since the number of required CEDAs is linearly increasing with the increasing size of the network topology, as shown in Figure 5 (c) and Figure 6 (c), we need to generate around 10% to 33% more CEDAs to support the threat hunting using hierarchical monitoring which is a feasible number of CEDAs. However, the number of required CEDAs is inversely co-related with monitoring capacity, as shown in Figure 5 (d) and Figure 6 (d). Since the number of required agents to monitor required tasks will decrease with the increasing monitoring capacity, the CEDA generation with the provided approximation algorithm is scalable.

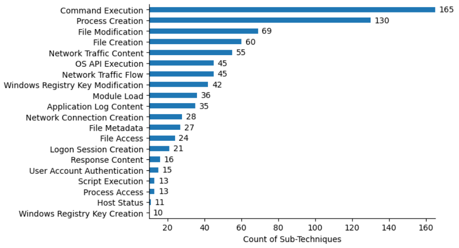

Fig. 7. Data source to technique coverage

Moreover, Figure 7 shows the technique coverage of the top 20 data sources among 81 data sources. We can see that Command Execution and Process Creation cover 36% to 42% techniques, while File Modification and File Creation cover around 32% techniques. Since most of the data sources cover multiple techniques, developing EFAs for around 25% of the data sources will cover all remaining techniques, which is feasible.

6 Conclusion, Limitations and Future Work

This paper describes a distributed hierarchical monitoring architecture for threat hunting using the MITRE ATT&CK framework as a guideline of attacker activities. We formalize the scalable threat hunting problem and provide an approximation algorithm to generate agent hierarchy, evaluation with three simulated attack use cases and DARPA OpTC attack dataset [15] with varying network size and monitoring capacity, and a threat hunting demonstration using the SCAHunter. The monitoring application for threat hunting in distributed large-scale systems must be scalable to handle many event producers (EFA) and selective monitoring of event producers, highly re-configurable to handle on-demand monitoring requests. An efficient event filtering mechanism must be used to reduce event traffic propagation and monitor intrusiveness.

The SCAHunter improves monitoring scalability using an approximation algorithm by the near-optimal composite event detector (CEDA) and primitive event detector (EFA) hierarchy generation. The proposed SCAHunter is scalable since we generate the minimum number of CEDAs that can communicate with the required EFAs considering the hop count among the CEDAs and EFAs. It also reduces event traffic propagation and monitoring intrusiveness by using the hierarchical structure and optimal generation of agent hierarchy. RabbitMQ ensures the security of agent communication.

Although we developed multiple primitive event detector agents, EFAs based on ETW trace, we need to develop additional EFA agents to cover all the data sources. Since the single-point-of-failure calls for applying the consensus mechanism, such as the leader election algorithm, which is another research problem to solve, we did not consider the single-point-of-failure for SCAHunter. Our agent architecture will not be optimal if a new attack signature added to the system overlaps with the existing monitored signatures. We plan to explore this topic in the future.

References

1. Adversarial tactics, techniques & common knowledge. https://attack.mitre.org/. [Online; accessed 15 march, 2022]
2. Atomic red team: Mitre attack technique detector. https://github.com/redcanaryco/atomic-red-team/. [Online; accessed 15 march, 2022]
3. Endpoint detection and response solution survey. https://www.gartner.com/reviews/market/endpoint-detection-and-response-solutions. [Online; accessed 3 August, 2022]
4. How much does a data breach cost? https://www.ibm.com/security/data-breach. [Online; accessed 15 July, 2022]
5. Logrythm:threat hunting use cases. https://logrhythm.com/use-cases/. [Online; accessed 15 April, 2022]
6. Special report m-trends 2021. https://www.mandiant.com/resources/m-trends-2021. [Online; accessed 15 July, 2022]
7. The state of ransomware 2022. https://www.sophos.com/en-us/content/state-of-ransomware. [Online; accessed 15 July, 2022]
8. Symmantec. attack listing. https://www.symantec.com/security-center/a-z
9. Ahmed, M., Al-Shaer, E.: Measures and metrics for the enforcement of critical security controls: a case study of boundary defense. In: Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security, pp. 1–3 (2019)
10. Al-Shaer, E., Abdel-Wahab, H., Maly, K.: Hifi: a new monitoring architecture for distributed systems management. In: Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003), pp. 171–178 (1999)
11. Al-Shaer, R., Spring, J.M., Christou, E.: Learning the associations of mitre att&ck adversarial techniques. In: 2020 IEEE Conference on Communications and Network Security (CNS), pp. 1–9 (2020). DOI 10.1109/CNS48642.2020.9162207
12. Alam, M.M., Wang, W.: A comprehensive survey on data provenance: State-of-the-art approaches and their deployments for iot security enforcement. Journal of Computer Security (Preprint), 1–24 (2021)

13. Alsaleh, M.N., Wei, J., Al-Shaer, E., Ahmed, M.: Gextractor: Towards automated extraction of malware deception parameters. In: Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop, SSPREW-8. Association for Computing Machinery, New York, NY, USA (2018). DOI 10.1145/3289239.3289244. URL https://doi.org/10.1145/3289239.3289244
14. Andreolini, M., Colajanni, M., Pietri, M.: A scalable architecture for real-time monitoring of large information systems. In: 2012 Second Symposium on Network Cloud Computing and Applications, pp. 143–150. IEEE (2012)
15. Arantes, R., Weir, C., Hannon, H., Kulseng, M.: Operationally transparent cyber (optc) (2021). DOI 10.21227/edq8-nk52. URL https://dx.doi.org/10.21227/edq8-nk52
16. Benahmed, K., Merabti, M., Haffaf, H.: Distributed monitoring for misbehaviour detection in wireless sensor networks. Security and Communication Networks **6**(4), 388–400 (2013)
17. Bhattarai, B., Huang, H.: Steinerlog: Prize collecting the audit logs for threat hunting on enterprise network. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, ASIA CCS '22, p. 97–108. Association for Computing Machinery, New York, NY, USA (2022). DOI 10.1145/3488932.3523261. URL https://doi.org/10.1145/3488932.3523261
18. Boem, F., Ferrari, R.M., Keliris, C., Parisini, T., Polycarpou, M.M.: A distributed networked approach for fault detection of large-scale systems. IEEE Transactions on Automatic Control **62**(1), 18–33 (2016)
19. Hassan, W.U., Bates, A., Marino, D.: Tactical provenance analysis for endpoint detection and response systems. In: 2020 IEEE Symposium on Security and Privacy (SP), pp. 1172–1189 (2020). DOI 10.1109/SP40000.2020.00096
20. Hassan, W.U., Guo, S., Li, D., Chen, Z., Jee, K., Li, Z., Bates, A.: Nodoze: Combatting threat alert fatigue with automated provenance triage. In: Network and Distributed Systems Security Symposium (2019)
21. Hassan, W.U., Li, D., Jee, K., Yu, X., Zou, K., Wang, D., Chen, Z., Li, Z., Rhee, J., Gui, J., Bates, A.: This is why we can't cache nice things: Lightning-fast threat hunting using suspicion-based hierarchical storage. In: Annual Computer Security Applications Conference, ACSAC '20, p. 165–178. Association for Computing Machinery, New York, NY, USA (2020)
22. Hassan, W.U., Noureddine, M.A., Datta, P., Bates, A.: Omegalog: High-fidelity attack investigation via transparent multi-layer log analysis. In: Network and Distributed System Security Symposium (2020)
23. Hong, P.T.N., Le Van, S.: An online monitoring solution for complex distributed systems based on hierarchical monitoring agents. In: Knowledge and Systems Engineering, pp. 187–198. Springer (2014)
24. Hossain, M.N., Milajerdi, S.M., Wang, J., Eshete, B., Gjomemo, R., Sekar, R., Stoller, S., Venkatakrishnan, V.: {SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data. In: 26th {USENIX} Security Symposium ({USENIX} Security 17), pp. 487–504 (2017)
25. Milajerdi, S.M., Eshete, B., Gjomemo, R., Venkatakrishnan, V.: 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, pp. 1795–1812 (2019)
26. Milajerdi, S.M., Gjomemo, R., Eshete, B., Sekar, R., Venkatakrishnan, V.: Holmes: real-time apt detection through correlation of suspicious information flows. In: 2019 IEEE Symposium on Security and Privacy (SP), pp. 1137–1152. IEEE (2019)

27. Sacerdoti, F.D., Katz, M.J., Massie, M.L., Culler, D.E.: Wide area cluster monitoring with ganglia. In: CLUSTER, vol. 3, pp. 289–289 (2003)
28. Wei, R., Cai, L., Zhao, L., Yu, A., Meng, D.: Deephunter: A graph neural network based approach for robust cyber threat hunting. In: J. Garcia-Alfaro, S. Li, R. Poovendran, H. Debar, M. Yung (eds.) Security and Privacy in Communication Networks, pp. 3–24. Springer International Publishing, Cham (2021)
29. Wood, A.: Rabbit MQ: For Starters. CreateSpace Independent Publishing Platform, North Charleston, SC, USA (2016)
30. Wu, Y.S., Foo, B., Mei, Y., Bagchi, S.: Collaborative intrusion detection system (cids): a framework for accurate and efficient ids. In: 19th Annual Computer Security Applications Conference, 2003. Proceedings., pp. 234–244. IEEE (2003)
31. Xiong, C., Zhu, T., Dong, W., Ruan, L., Yang, R., Chen, Y., Cheng, Y., Cheng, S., Chen, X.: Conan: A practical real-time apt detection system with high accuracy and efficiency. IEEE Transactions on Dependable and Secure Computing (2020)
32. Yegneswaran, V., Barford, P., Jha, S.: Global intrusion detection in the domino overlay system. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences (2003)

Appendix A

In order to build the attack signature in terms of low-level system events, we follow the technique detectors provided by LogRythm [5]. Following the rules developed by LogRythm, technique T1189 can be detected by inspecting IDS or antivirus logs. Technique T1035, T1021.001, and T1119 [1] can be detected by analyzing Windows event logs or SysMon logs as mentioned by MITRE [1]. Moreover, technique T1048 can be detected by analyzing network traffic and looking for uncommon data flow in the NetMonitor. Therefore, we can define the signature in terms of low-level logs for technique T1189, T1035, T1021.001, T1119, and T1048 [1], and use case 2 using ESP formalization provided in Section 3 as follows:

$$\begin{multline} IDSMon.\_event\_id == Malware\_detected \wedge \\ Malware\_detected.malwareFileDir ==' *temp' \wedge \\ SysMon.\_event\_id == object\_access \wedge \\ object\_access.object\_dir ==' *temp' \wedge \\ object\_access.procName =' chrome' \tag{10} \end{multline}$$

$$\begin{multline} SysMon.\_event\_id == service\_creation \wedge \\ service\_creation.command ==' SC\ create' \wedge \\ service\_creation.imagepath == `*temp' \tag{11} \end{multline}$$

$$SysMon.\_event\_id == RDS\_logon\_success \tag{12}$$

$$\begin{multline} SysMon.\_event\_id == process\_created \wedge \\ process\_created.command ==' *.bat' \wedge \\ SysMon.\_event\_id == network\_conn\_created \tag{13} \end{multline}$$

$$NetMon.\_event\_id == uncommon\_data\_flow \tag{14}$$

$$\begin{multline} (Equation\ 10) \wedge (Equation\ 11) \wedge (Equation\ 12) \wedge (Equation\ 13) \wedge (Equation\ 14) \wedge \\ (object\_access.object\_dir == service\_creation.imagePath == \\ malware\_detected.malwareFileDir) \wedge (RDS\_logon\_success.src\_ip == \\ ids\_event\_id.host\_ip) \wedge (RDS\_logon\_success.session\_id == \\ process\_created.session\_id == network\_conn\_created.session\_id) \wedge \\ (uncommon\_data\_flow.network\_protocol == \\ network\_conn\_created.network\_protocol) \tag{15} \end{multline}$$

To build the attack signature of the red team activities in OpTC attack dataset, we investigate notepad++ update process and Meterpreter execution process. To detect Meterpreter payload download activities, we can look for new file create and write event logs and process creation using the newly created file.

$$\begin{aligned} &a.Operation == NewFileWrite \wedge x.event\_id == process\_creation \\ &\qquad \wedge x.imagePath == a.newFilePath \end{aligned} \tag{16}$$

Named pipe impersonation can be detected by looking for any cmd.exe process which is a child of services.exe and the commandline arguments of the cmd.exe process contains echo and pipe keywords.

$$\begin{aligned} &b.processName == cmd.exe \wedge b.parentProcess == services.exe \\ &\qquad \wedge b.Commandline \in echo \wedge b.Commandline \in pipe \\ &\qquad \wedge b.process\_id == x.process\_id \end{aligned} \tag{17}$$

The system info, installed applications, domain controllers and network share discovery activities can be detected by looking for command line arguments running form the cmd.exe process which is spawn from Meterpreter process.

$$\begin{aligned} &(c.commandline \in [local\_system\_info\_enumeration\_command \\ &\qquad \cup installed\_application\_enumeration\_command(tasklist) \\ &\qquad \cup domain\_controller\_enumeration\_command(dclist) \\ &\qquad \cup network\_share\_enumeration\_command(Get\_SmbShare)] \\ &\qquad \wedge c.process\_id == b.process\_id) \end{aligned} \tag{18}$$

We can detect the registry key creation and setting the value of the created key to a newly downloaded payload as follows:

$$\begin{aligned} &g.Operation == NewFileWrite \wedge h.Operation == RegistryKey\_create \\ &\qquad \wedge g.newFilePath == h.registryKeyCreated.value \\ &\qquad \wedge g.process\_id == h.process\_id == x.process\_id \end{aligned} \tag{19}$$

Creating new user account and adding it to specific group can be done by using net utility. Thus the signature will be:

$$\begin{aligned} &i.Commandline \in net\_userAccount\_add\_command \\ &\qquad \wedge i.process\_id == b.process\_id \wedge i.processName == Net.exe \end{aligned} \tag{20}$$