Lateral Movement Detection Using Distributed Data Fusion

Ahmed Fawaz*, Atul Bohara†, Carmen Cheh†, William H. Sanders*
*Department of Electrical and Computer Engineering, †Department of Computer Science
University of Illinois at Urbana-Champaign
Email: {afawaz2, abohara2, cheh2, whs}@illinois.edu

***Abstract*—Attackers often attempt to move laterally from host to host, infecting them until an overall goal is achieved. One possible defense against this strategy is to detect such coordinated and sequential actions by fusing data from multiple sources. In this paper, we propose a framework for distributed data fusion that specifies the communication architecture and data transformation functions. Then, we use this framework to specify an approach for lateral movement detection that uses host-level process communication graphs to infer network connection causations. The connection causations are then aggregated into system-wide host-communication graphs that expose possible lateral movement in the system. In order to provide a balance between the resource usage and the robustness of the fusion architecture, we propose a multilevel fusion hierarchy that uses different clustering techniques. We evaluate the scalability of the hierarchical fusion scheme in terms of storage overhead, number of message updates sent, fairness of resource sharing among clusters, and quality of local graphs. Finally, we implement a host-level monitor prototype to collect connection causations, and evaluate its overhead. The results show that our approach provides an effective method to detect lateral movement between hosts, and can be implemented with acceptable overhead.**

I. Introduction

Resiliency is the ability of a system to maintain proper service when facing abnormal changes. In this work, we focus on resiliency to intrusions in networks and distributed systems. With the growth in size and complexity of these systems, it has become practically impossible to prevent all intrusions. Resiliency provides an additional layer of security by detecting the intrusions and controlling the effects of these intrusions while maintaining system service.

Monitoring the operation of a system is essential to achieving resiliency. Monitoring information is used to estimate the current state of the system, detect intrusions, and drive response actions.

In any real-life system, the volume of information that is required to construct a system-wide state can grow rapidly, imposing significant challenges on any analysis for resiliency, such as intrusion detection [1], [2]. Previous approaches that tried to process the large volume of information for intrusion detection have not been adopted in practice. Most of these approaches require significant manual effort and domain expertise to build models for intrusion detection. Moreover, since the models are built based on what has already been observed in the system, detection of previously unseen intrusions may fail. These approaches are also unable to detect attacks that happen over a long period of time, such as long-lasting targeted attacks and coordinated attacks. This is evidenced by the fact that many attacks are detected long after a significant loss has already been incurred [3].

To address the problem of information overhead in constructing a system-wide state, we propose a distributed data fusion framework. This framework formally specifies how information should be collected, exchanged, and transformed to detect certain intrusions and possibly respond to them. This framework also allows us to reduce resource overhead on a central location and provide a robust processing architecture.

We demonstrate the use of the proposed fusion framework in detecting lateral movement behavior. Observed in many long-lasting targeted attacks such as Advanced Persistent Threats (APTs), lateral movement is the phase of an attack in which the attacker tries to expand control over other machines in a network starting from one compromised machine. Detection of lateral movement imposes significant challenges in terms of information overhead, and requires coordination among multiple entities in a system.

In our approach, monitoring and fusion agents at different levels across the system coordinate to detect lateral movement. The agents are arranged in a hierarchy from the host level, to the cluster (group of hosts) level, to the global level.

The host-level agents collect process information from the kernel and infer *causation* relationships between incoming and outgoing network connections. A causation relation implies that there is a dependency between the incoming and outgoing connections. Use of kernel-level information allows us to infer the connection causations more accurately than we could by just using timing information or port numbers. The higher-level cluster agents use abstracted data from host-level agents and construct a graph of lateral movement. Finally, the global agent uses the information from the cluster agents to generate a global view of lateral movement in the system.

We demonstrate that the distributed fusion enables lateral movement detection in large systems by distributing the storage and processing overhead among multiple clusters. We also show that fairness and locality of information among clusters can be achieved using different types of host clustering approaches.

Our contributions can be summarized as follows:

- We propose a distributed data fusion framework for system resiliency, and formalize different fusion require-

ments within our framework (Section II).
- We show the utility of our framework by developing agent-based monitoring and fusion mechanisms to detect lateral movement behavior in an enterprise system (Sections III, IV).
- We propose a method to infer *communication causation events* by collecting and analyzing kernel-level process activities on a host (Section IV).
- We perform a trace-based simulation experiment to evaluate the lateral movement detection approach in terms of scalability, fairness of distributed processing, and quality of local states at higher-level agents (Section V).
- We use DTrace on an OS X machine to implement a prototype host-level data collection and processing agent, and we evaluate its overhead (Section V).

II. Data Fusion Framework

Data fusion, in a general sense, is defined as a set of tools and techniques to combine data originating from different sources. Data fusion aims to obtain information of greater quality [4]. Data fusion is required for intrusion resiliency to obtain a holistic view of the system state that can be acted upon without overwhelming the analyses. This leads us to define the components and formalism of our data fusion framework as follows.

A. Components of Data Fusion

The main building blocks of our fusion framework are the *agents* that are responsible for monitoring and fusion in a computing system. More precisely, agents perform data collection, transformation, and transmission. The communication structure of these agents is defined by a fusion architecture.

a) Data collection: Agents collect data by monitoring a local computer system or communicating with other agents. The data available to an agent at time t represent the local state of the agent at time t. For example, we can implement an agent to collect kernel-level activities on a host machine.

b) Data transformation: The agents can apply transformation functions on their local state to convert it into different representations. In most cases, the purpose of transformation is to decrease the level of complexity of the system state representation. We say that this process increases the *level of abstraction*. The highest level of abstraction is the entire system view, while the lowest level could be a very detailed view of what an individual agent observes. A higher-level abstraction provides a more concise view of a larger part of the system at the cost of some information loss.

c) Data transmission: The agents send a transformed representation of local state to other agents. Triggering events or *triggers* determine when the data are transmitted. The triggers can be periodic, i.e., based on a function of time, or state-specific, i.e., based on a function of the agent's current state.

d) Fusion architecture: The fusion architecture defines how the agents communicate in order to disseminate the data among themselves to achieve a specific goal. Agents can communicate to one central server or among themselves in a distributed manner.

The agents can be divided into multiple levels of hierarchies to build a hierarchical fusion architecture. The agents that are higher in the hierarchy receive data from lower-level agents and apply transformations to fuse and convert the low-level data to the right level of abstraction.

Different hierarchical structures provide a tradeoff between communication overhead and robustness. A centralized structure is simple and incurs less communication overhead. However, the central collection agent at the root of the tree is a single point of failure. A fully distributed architecture, on the other extreme, is more robust to failures but comes with additional communication overhead. The number of levels in the hierarchy and the communication protocol at each level can be chosen based on the goal of fusion. We describe and evaluate these variations in later sections.

B. Data Fusion Framework

Definition The fusion framework $\mathcal{F}$ is defined as a quadruple $(G, f, g, \mathbb{T})$.

G : A directed graph $G(V, E)$ that defines the fusion architecture. The set of vertices, $V = \{1, 2, \ldots, n\}$, represents the agents in the system. The set of edges, $E \subseteq V \times V$, represents the communication structure of the agents.

f : A set of transformation functions, $\{f_i | \hat{x_i} = f_i(x_i), \forall i \in V\}$, applied by an agent to fuse and abstract local data. The output of the function f_i is $\hat{x_i}$, the current state of the agent i.

g : A set of transformation functions, $\{g_j | \hat{x_j}' = g_j(\hat{x_j}), \forall j \in E\}$, that an agent can apply before sending the local state to another agent located on the tail endpoint of edge j.

$\mathbb{T}$: A set of temporal propositions, $\{T_j, \forall j \in E\}$, that represents the triggering events that cause the agent to send data along edge j to the agent at the tail endpoint.

Fig. 1 shows examples of different ways in which the fusion framework can be instantiated. Fig. 1a shows a centralized architecture in which the root of the tree is a collection agent that receives data from its child agents. It can combine and increase the level of abstraction of the data received from the children. Fig. 1b shows an extension of the centralized architecture with multiple levels of hierarchy. In this three-level structure, the lowest-level agents are divided into multiple clusters, with each cluster having one leader to collect information from the lowest-level agents. The cluster leaders then send their information to a global leader who is at the root of the tree. Finally, Fig. 1c shows a variation of the hierarchical architecture in which the agents in the topmost level communicate in a distributed manner.

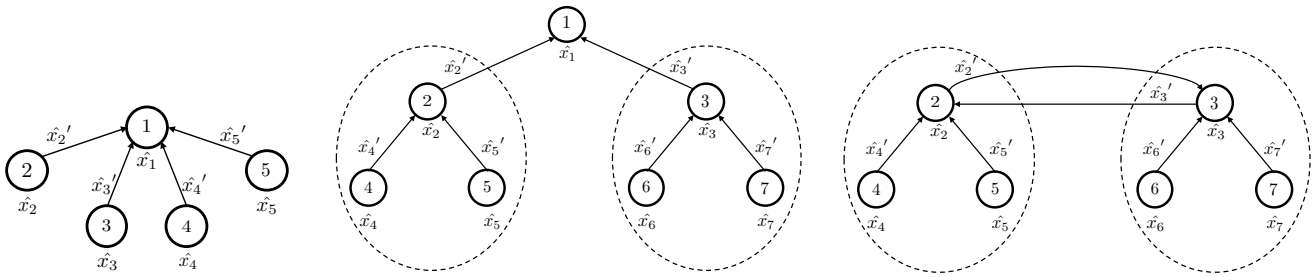

Fig. 1: Examples of fusion framework, $\mathcal{F}$, with different types of hierarchical fusion architectures, G: (a) 2 levels with centralized communication, (b) 3 levels with centralized communication, and (c) 2 levels with distributed communication at the top level. At agent i, the local state $\hat{x_i}$ is transformed into $\hat{x_i}' = g(\hat{x_i})$ and sent to another agent. Grouping of hosts into clusters is shown by the dotted boundaries. Depending upon the underlying network topology, different clustering algorithms can be used for different trade-offs among scalability, fairness of distributed processing, and quality of local state.

III. Overview of Lateral Movement Detection

Using the fusion framework proposed in Section II, we describe our approach to detect lateral movement. We first define lateral movement and argue for the necessity of detecting such behavior. Then, we give an overview of our approach.

A. Lateral Movement

Even though cyber intrusions take many forms, most forms have a common anatomy, described by Lockheed Martin Corp.'s intrusion kill chain [5]. The phases of attack in this kill chain are reconnaissance, weaponization, delivery, exploitation, lateral movement, and actions on objectives. Usually, an attacker performs the delivery with social engineering, spear-phishing, or malware. Exploitation is done by gaining privileged access on a host and establishing command and control (C2). Then, the attacker performs *lateral movement*, which involves use of legitimate services to move to other hosts in the network. This allows the attacker to expand his or her control over the system and eventually reach a set of target machines. After lateral movement, the attacker maintains access to the system to eventually achieve some malicious goal, such as data exfiltration or service disruption.

We focus on the lateral movement phase because if we are able to detect and thwart an attack in this phase, we can prevent the system from sustaining more serious damage. Recently, lateral movement played an important part in the Ukrainian power grid control network compromise. After using spear-phishing to gain access to the internal network, the attackers used VPN and remote desktop to move laterally through the system and control hosts in order to trip breakers in substations [6].

In this work, we detect all lateral movement chains, whether they are malicious (e.g., part of an intrusion kill chain) or benign (e.g., a system-wide administrative task [7]). Since an attacker may hide lateral movement by using the benign chains of events in the system [8], it is difficult to distinguish between benign activity and malicious behavior that hides within such activity. Therefore, we believe that finding all possible chains of events that are related to each other is the first step towards discovering evidence of malicious lateral movement.

B. General Overview of Approach

In our work, we use a hierarchical fusion architecture to fuse host process communication information and network connection information to track lateral movement as it happens in the system.

Intuitively, lateral movement can be thought of as a targeted walk on a network graph such that sequential pairs of steps are causally related. To track this walk, we need to detect series of ordered network connections between hosts. However, not all network connections between hosts will be part of lateral movement. It is typical to use known port numbers or timing to correlate incoming and outgoing connections in a hosts. This leads to a high number of false positives and false negatives. To improve the accuracy of correlation, we monitor inter-process communications to establish a causation relation between incoming and outgoing connections on each host across the system.

Fig. 2 shows the overview of our approach for using distributed data fusion to detect lateral movement. Each host in the system maintains a process communication graph (PC-graph) by monitoring inter-process communication. The host uses the PC-graph to infer causation between incoming and outgoing connections. When a connection causation event is detected, the host contracts the PC-graph and sends an update to a higher-level collection agent. In a network, hosts are clustered into sets; each cluster has a leader that collects connection causation events. The leader then uses the events to build a host communication graph (HC-graph). The HC-graph tracks related connections between hosts and, thus, tracks lateral movement. Finally, as the network grows in size, maintenance of a full HC-graph becomes infeasible. We then create a third-level abstraction in which a global leader collects abstracted views of the HC-graphs from the clusters in the network.

To summarize, the global agent stores the most abstracted view (cluster communication); the cluster leaders store a more detailed view (host communication); and host-level agents store the least abstracted view (process communication). As we go up the hierarchy, the purview increases, but the level

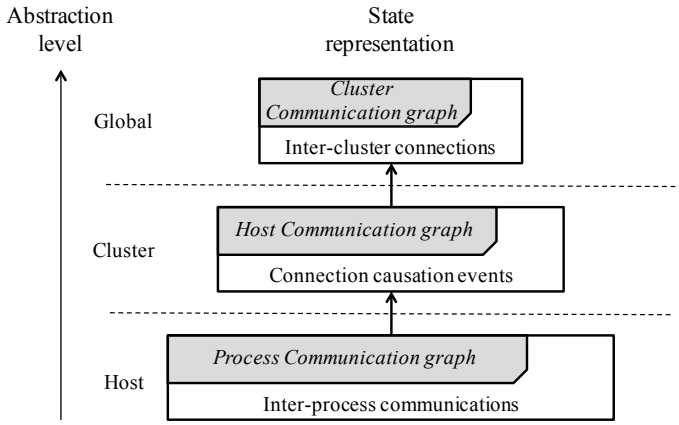

Fig. 2: Overview of the lateral movement detection architecture. A host-level monitor collects inter-process communication events and generates a Process Communication (PC) graph. It extracts connection causation events from the PC-graph and sends them to a cluster leader. The cluster leader fuses the events to construct a Host Communication (HC) graph. The cluster leader abstracts the HC-graph and updates a global leader with inter-cluster connection events. The volume of information decreases as abstraction level increases.

of detail about the communication becomes more abstracted. Thus, in essence, our data fusion architecture generates a global communication graph.

IV. Lateral Movement Detection

Based on the high-level overview of our approach given in Section III-B, we now detail our detection technique. First, we introduce the data models used to represent system state. Then, we propose two different fusion architectures for generating the state. Finally, we describe how agents on the host generate the information that is required for fusion.

A. Data Model

Lateral movement behavior is a chain of communication events; we track these communication events and represent them as system state. In this section, we formally define communication events and the communication causation relation. We use the communication causation events to define a communication causation graph (CC-graph). We also formally define the host communication graph (HC-graph) and the transformation from a CC-graph to an HC-graph.

We consider communication to be any event by which two entities exchange information. Communication flows are directional by nature; nevertheless, information flow might be symmetric. Communication occurs on multiple levels in a system: at the network level between hosts, and at the host level between processes.

Definition The universe of communication events is defined as $\mathcal{C}$. A communication initiated from A to B is defined as a communication event $c = \overrightarrow{AB}$, where $c \in \mathcal{C}$. The communication event is initiated at time $t(\overrightarrow{AB})$ and has a unique ID $h(\overrightarrow{AB})$[1].

We define the system state according to the *causation* relation between connection events. In lateral movement, an attacker starts from a host X, moves to a host Y, and then uses host Y to target other hosts. This malicious connection from Y to the other hosts is *caused* by the malicious connection from X to Y. The notion of connection causality in lateral movement refers to the existence of a flow of events that signify a dependency between an outgoing connection and an incoming connection. Instead of using only temporal ordering among network events to establish causality, we use host-level process communication to find a dependency path between the process that sent an outgoing connection, and the process that receives a connection. By using host-level process communication information, we increase the accuracy of the inferred connection causalities. We formally define connection *causation* below. In Section IV-D, we describe the procedure to infer communication causation relations using process communication graphs.

Definition Let $x, y \in \mathcal{C}$ be communication events. *Connection causation* is a relation of the form $x \trianglerighteq y$ such that y was caused by x. This form $x \trianglerighteq y$ is also termed a *causation event*[2].

Now we construct the *Communication Causation graph* that directly represents the lateral movement behavior.

Definition The *Communication Causation graph* (CC-graph) is a directed graph $CCG = (V, E, \ell_V)$ where the set of vertices is the observed communication events in the system, and an edge connecting two vertices signifies a causation relation between the two communication events. The function $\ell_V : V \to h(\mathcal{C})$ labels the vertices with an ID taken from the set of connection IDs.

The CC-graph represents connection events as vertices and is thus unbounded in the number of vertices, since the connections in a system are unbounded. State-of-the-art theoretic analysis of dynamic graphs only considers insertion and deletion of edges, whereas graphs with changing vertices (e.g., CC-graphs) are not fully studied. Thus, we propose to transform the CC-graph to a *Host Communication graph* that has a static number of vertices.

Definition The *Host Communication graph* (HC-graph) is a directed graph $HCG = (V, E, \ell_V, \ell_E)$ where the set of vertices represents hosts in the system, and the set of edges represents connections. The function $\ell_V : V \to \Sigma_V$ labels a vertex with an ID taken from the set of host IDs Σ_V, and $\ell_E : E \to \Sigma_E$ labels an edge with an ID and a bit array b.

For every host, the edge ID on an incoming connection represents a bit mask on the bit array of every outgoing edge.

[1] A connection is easy to identify; e.g., in TCP, the hash of port numbers and sequence numbers in a SYN message is unique. Moreover, $t(c)$ is relative to a universal system clock.

[2] If $x \trianglerighteq y$ then $t(x) < t(y)$, and if $x \trianglerighteq y \trianglerighteq z$ then $t(x) < t(y) < t(z)$.

When causation event $x \trianglerighteq y$ is added, the mask is used to set on edge y the bit positions represented by the ID of edge x.

In most cases, any two incident edges in the HC-graph define a causation relation between two unique connections. However, when a host is reused during lateral movement, some incident edges in the graph will not correspond to valid causation relations. Those ambiguities are clarified using the unbounded bit arrays on the edges of the HC-graph.

To transform a CC-graph to an HC-graph, we take each causation event $c_k = \overrightarrow{H_i H_j} \trianglerighteq \overrightarrow{H_j H_k}$ and add it to the HC-graph as two directed edges e_1, e_2, such that:

- $e_1 = (v_i, v_j)$ s.t. $(\ell_V(v_i) = H_i) \wedge (\ell_V(v_j) = H_j)$.
- $e_2 = (v_j, v_k)$ s.t. $(\ell_V(v_j) = H_j) \wedge (\ell_V(v_k) = H_k)$.
- $\ell_E(e_1) = (id_1, b_1)$, where id_1 is unique to e_1.
- $\ell_E(e_2) = (id_2, b_2)$, where $b_2[id_1] = 1$ and id_2 is unique to e_2.

The bit array used to represent the causation events in the HC-graph is unbounded. In practice, we want a bounded scheme to represent those events. Depending on the usage of the HC-graph for detection, we list a few possible methods to encode the causation events: (1) **Unbounded**; (2) **Limited history**, in which the size of the bit array is fixed and old data are overwritten during an overflow; (3) **Probabilistic**, in which a Bloom filter is used for each edge to encode the causation events; and (4) **Nondeterministic**, in which no explicit causation relations are stored on the edges. In the nondeterministic case, the structure of the HC-graph still guarantees that there exists a valid HC-graph to CC-graph transformation.

Finally, a valid HC-graph should simulate a CC-graph. The simulation relation is defined as a one-to-one mapping from an HC-graph to a CC-graph, in which the connection IDs are unique but not exactly reproduced.

Proposition 1. *A valid HC-graph simulates a CC-graph.*

Proof. We define a function that maps an HC-graph to a CC-graph. For each pair of incident edges $e_1(v_0, v_1), e_2(v_1, v_2) \subseteq E \times E$ in the HC-graph, if the pair represents a valid causation event $e_1 \trianglerighteq e_2$, then two vertices E_1, E_2 connected by an edge $e'(E_1, E_2)$ are added to the CC-graph. □

B. Centralized Architecture

We can detect lateral movement by joining pieces of the local graph together to construct a global graph of attacker movement. Using our fusion framework, we can define a 2-level fusion architecture (Fig. 1b). In that setting, monitoring agents on all hosts maintain a local process communication graph, which is contracted whenever a causation event occurs. Each agent then sends the causation event. We now describe the fusion algorithm that uses the received causation events to maintain a set of HC-graphs.

We model the system state as a set of HC-graphs $\mathcal{G} = G_1, G_2, \cdots, G_k$. This state represents possible lateral movement chains as they are being tracked by the central collection agent. Each time a new causation event is received, the state is updated. In particular, Algorithm 1 updates the state by first searching for the connection IDs in each of the graphs (line 2). If neither of the connection IDs is found, a new graph is created and added to the state (line 15). If only one of the connection IDs is found, a new edge is added to the corresponding graph (line 8). If both connection IDs are found, then the graphs are merged (line 11).

Algorithm 1 System State Maintenance

Input: $\mathcal{G} = \{G_1, G_2, \cdots, G_k\}$
Input: $c = x \trianglerighteq y$
1: **procedure** HC-GRAPH-MAINTAIN
2: $G_i \leftarrow$SEARCH($\mathcal{G}$, x)
3: $G_k \leftarrow$SEARCH($\mathcal{G}$, y)
4: **if** $G_i = \varnothing$ & $G_k \neq \varnothing$ **then**
5: ADDEVENT(G_k, c)
6: **end if**
7: **if** $G_i \neq \varnothing$ & $G_k = \varnothing$ **then**
8: ADDEVENT(G_i, c)
9: **end if**
10: **if** $G_i \neq \varnothing$ & $G_k \neq \varnothing$ **then**
11: $G_m \leftarrow$MERGE(G_i,G_k,c)
12: $\mathcal{G} \leftarrow \mathcal{G} \setminus \{G_i, G_k\} \cup G_m$
13: **end if**
14: **if** $G_i = \varnothing$ & $G_k = \varnothing$ **then**
15: $\mathcal{G} \leftarrow \mathcal{G} \cup$ NEWGRAPH(c)
16: **end if**
17: **end procedure**

The collection agent also maintains a hash table for storing the connection IDs and the graph in which the connection is represented. The SEARCH function checks this hash table for the connection.

The ADDEVENT function takes as input an HC-graph G_k and a causation event $c = \overrightarrow{H_i H_j} \trianglerighteq \overrightarrow{H_j H_k}$. It then inserts two edges $e_1(H_i, H_j)$ and $e_2(H_j, H_k)$ into G_k. We use a shorthand $G'_k = G_k + c$ for the ADDEVENT function.

The causation events allow the collection agent to receive out-of-order messages, because timing information is encoded in the semantics of the relation. The MERGE function is used to merge two lateral movement graphs when a causation event is received out of order. It takes as input two HC-graphs and adds all the edges from one graph to the other.

Proposition 2. HC-GRAPH-MAINTAIN *has a constant amortized runtime complexity,* $\mathcal{O}(1)$.

Proof. The HC-GRAPH-MAINTAIN algorithm is an online algorithm. The SEARCH function uses a hash table that has $\mathcal{O}(1)$ amortized runtime complexity. The ADDEVENT function runs in constant time $\mathcal{O}(1)$. The MERGE function has a runtime $\mathcal{O}(\min(|E_1|, |E_2|)$. The worst-case running time occurs when we merge graphs every time the size of the input doubles. That is, we merge graphs of sizes $1, 2, 4, \ldots, n/2, n$, where n is the number of causation events. The total worst-case running time is $1+2+4+\ldots+n/2+n < 2n$. Thus, the worst-case amortized runtime complexity of the algorithm is $\mathcal{O}(1)$. □

Finally, we validate the state maintenance algorithm by proving inductively that each operation should keep the state (i.e., all the HC-graphs) valid.

Proposition 3. HC-GRAPH-MAINTAIN *generates a set of valid HC-graphs.*

Proof. For $|\mathcal{G}| = 0$ (i.e., no graphs in the state), a new graph containing the vertices and edges involved in the causation is added. An HC-graph with one causation event can simulate a CC-graph. Assume we have $|\mathcal{G}| = n$ valid graphs. When we receive a new event e, there are three cases:

- The search leads to one match, and the event is added to the respective graph. The addition of the event maintains the validity of the graph as the event is added to the simulated CC-graph.
- The graphs G_1 and G_2 are to be merged. Since a graph can be seen as a series of event additions, $G_2 = \sum_i e_i$, MERGE inserts the sequence of events of G_2 into G_1. Since adding an event preserves validity, adding a sequence of events also preserves validity.
- No matches exist, so a new graph is created with a single event. This is similar to the base case.

□

C. Multilevel Hierarchical Architecture

The centralized fusion architecture works well for smaller systems. However, as the system size increases, the local computing and network bandwidth requirements at the central location may lead to poor performance of the overall system. The central agent also becomes a single point of failure.

To address the weaknesses of the centralized architecture, we propose a hierarchical architecture. In this setup, we group the host-level agents into multiple clusters. One agent in the cluster is chosen as the cluster *leader* that receives causation events from all host-level agents in the same cluster. A cluster leader applies suitable transformations to the received events and shares this information with other cluster leaders or a global leader. The hierarchical architecture is thus an extension of the centralized architecture, since it consists of multiple centralized clusters of agents that coordinate with each other on the next level of abstraction.

To show that our data model and fusion framework can be extended to specify a hierarchical architecture, we describe a three-level hierarchical architecture, shown in Fig. 1b. It is easy to see that the same method can be applied to other types of hierarchical architectures. As in the centralized case, the host agents in the three-level hierarchical architecture maintain a process communication graph and send causation events to the cluster leader. Each cluster leader acts as the centralized collection agent for its cluster and uses Algorithm 1 to maintain HC-graphs. At a cluster leader, the external hosts (i.e., hosts that are not in the cluster) in an HC-graph will have either no incoming edges (a source vertex) or no outgoing edges (a sink vertex). From the definition of the HC-graph, since a cluster leader can receive causation events only from hosts within the cluster, it can never merge two graphs that have an external host in common.

A cluster leader abstracts its HC-graphs and generates a *Cluster Communication Graph* (CL-graph), which we define as follows.

Definition A *Cluster Communication Graph* (CL-graph) is a graph $CLG = (V, E, \ell_V, \ell_E)$, where the set of vertices represents cluster leaders and an edge $(c_1, c_2) \in E$ represents the connection from a host in cluster c_1 to a host in cluster c_2. The function $\ell_V : V \to \Sigma_V$ labels the vertices with an ID taken from the set of cluster IDs Σ_V, and $\ell_E : E \to \Sigma_E$ labels the edges with connection IDs between the two hosts.

Transformation function at cluster leader: The transformation function applied by a cluster leader to abstract the local state (HC-graph) into the higher-level CL-graph is as follows. On receiving a new causation event from the host-level agent, the cluster leader c looks for an outgoing connection in the HC-graph. If there is one, it does a backwards traversal to find all the incoming connections present in the same HC-graph; also, the cluster leader collapses consecutive vertices that share the same cluster ID into a single vertex representing that cluster ID, while retaining connection IDs on the edges. This algorithm generates inter-cluster connection chains with an incoming connection to cluster c, causing an outgoing connection from cluster c. These inter-cluster connection chains are sent to the global leader.

Trigger events: We only consider one trigger event: the existence of an outgoing connection from the cluster c that is caused by an incoming connection to the cluster.

Transformation at the global leader: The global leader fuses the inter-cluster connection chains received from different cluster leaders. It runs an algorithm similar to Algorithm 1 at the cluster-level abstraction. The CL-graph maintained at the global leader exposes the system-wide lateral movement chains.

We believe that the multilevel hierarchical architecture is suitable for most real-life systems and provides a balance between communication overhead and robustness, because it adopts the scale-out model rather than the scale-up model of the centralized architecture. The scalability benefits will be shown in a detailed evaluation presented in Section V.

D. Causation Event Generation on Hosts

The fusion architecture uses the causation events emitted by host-level agents to build the HC-graphs. A host-level agent uses process communication events to infer causation relations. We use process communication instead of pure timing information between connections in order to reduce false causation relations. For example, a causation event is inferred when a process in a host receives a connection, forks a new process, and then creates an outgoing connection to a new host.

To infer causation events, we build a *Process Communication graph* that contains time-ordered remote and local process interactions.

There are three types of elements in a Process Communication graph: a local process, a remote process, and a file. A process is identified by a unique identifier, and not by the OS-supplied PID, which is reused; a remote process is identified by the port number and addresses of the remote host; and a file is identified by the unique ID assigned by a file system, and not by the file name.

We consider three types of process communication events: 1) a network connection with a remote process, 2) a transient communication (local interprocess communication, memory operation, or file read), and 3) a permanent state change (file or Registry writes). All these events, except file reads and writes, are bidirectional.

Definition The *Process Communication graph* (PC-graph) is a graph $PCG = (V, E, t)$, where the set of vertices corresponds to processes and files, an edge connecting two vertices represents a process communication, and the function t labels the edges with a time interval $[t_i, t_j]$ of the communication.

The time interval $[t_i, t_j]$ denotes the time when the communication channel was established, t_i, and the time of the last observed communication, t_j. When a communication channel such as shared memory between processes cannot be observed, as opposed to sockets and pipes, we assume that the channel is active at all times.

A host-level agent monitors processes and collects the relevant communication events. When a new event is observed, the PC-graph is updated through addition of a process/file vertex with the relevant attributes, or through addition or updating of an edge with the observed time. The timestamps of the communication events are sampled using the system's provided timestamps.

We use the PC-graph to infer causation events. A causation event is emitted when a valid path is found. A valid path starts from an incoming connection edge, passes through a set of communicating processes whose periods of activity are sequential or overlapping, and ends on an outgoing connection edge.

Definition A valid path is defined as $p = r_{in}, \ldots, p_i, e_i, p_j, e_{i+1}, p_k, \ldots r_{out}$ where

- The outgoing connection, r_{out}, happens before the incoming connection r_{in}, i.e., $t(r_{in}) < t(r_{out})$,
- $t(e_i)$ and $t(e_{i+1})$ overlap, and
- $t(e_i)$ happened before $t(e_{i+1})$.

We find these causation events using Algorithm 2. The algorithm, a modified Depth First Search (DFS), walks the graph starting from an outgoing connection edge until an incoming connection edge is found. At every step, the algorithm picks edges such that the properties in the valid path are satisfied.

Algorithm 2 Modified Depth First Search (DFS)

```
1: procedure MOD-DFS(G,v,E)
2:     if v is an incoming connection then
3:         trigger causation event
4:     end if
5:     if all edges to v are visited then
6:         label v as discovered
7:     end if
8:     for edge k=(w,v) in time-ordered set G.E(v) do
9:         if vertex w is not discovered AND edge k is not visited AND
   edges (k,e) satisfy the definition in Section IV-D then
10:            DFS(G,w,k)
11:        end if
12:    end for
13: end procedure
```

The algorithm has a worst-case runtime complexity of $\mathcal{O}(|E| + |V|)$. However, the PC-graph is not fully connected, and has many disconnected sub-graphs. So in the average case, MOD-DFS will only walk smaller sub-graphs. Section V-C shows that the overhead of our prototype host-level agent is less than 10%.

To reduce the memory overhead of the host-level agent, we prune the PC-graph to remove those processes that no longer affect running processes. When a process is terminated or a file is deleted, the agent removes the process from the PC-graph if all reachable processes from the terminated process have terminated. However, if the process has a path to a file write event, then the process is not removed until the file itself is removed or overwritten.

V. Evaluation

We used trace-based evaluation to study the performance trade-offs of the proposed lateral movement detection approach. We focused mainly on evaluating our hypothesis that the resource overhead on the leader can be distributed using the multilevel hierarchical fusion architecture for lateral movement detection. First, we implemented the algorithms presented in Section IV. Next, we simulated lateral movement over a network topology, and ran our fusion algorithms using the simulation traces. We evaluated the scalability of the hierarchical fusion by implementing different clustering techniques and computing fairness and locality metrics. Finally, we evaluated the performance overhead of a prototype implementation of the host-level agent to confirm its practicality.

A. Experiment Setup

We modeled a generalized lateral movement as a set of two-state Markov chains at every node. The transition probabilities between the states are affected by the state of neighboring nodes. The complete system dynamics are expressed in equation 1.

$$P_i = (I + \beta A)P_{i-1} \tag{1}$$

where $P_i \in \mathbb{R}^{n \times 1}$ is a probability vector describing whether a node has been visited at time i, $A \in \{1, 0\}^{n \times n}$ is the network topology, and β is the rate of node traversal. The rate of traversal describes the aggressiveness of the attacker during lateral movement; a high value of β denotes an aggressive attacker, signifying fast lateral movement. We model the skill level of an attacker, α, as the probability that a host in the system is exploitable by the attacker.

The simulation starts by using the attacker's skill level to pre-select hosts that are vulnerable. Then, the simulation runs the dynamic model from equation 1, and generates a trace of causation events to evaluate the proposed framework for lateral movement detection. Table I shows the values of the simulation parameters that we used in our experiment. The network topology is a generalized random graph (GRG) with probability γ. As mentioned in Section IV-C, the trigger event is an outgoing connection from a cluster, and the transformation function is the cluster-level abstraction. For the centralized fusion architecture, all the events are processed by

a centralized collection agent. For the hierarchical architecture, the nodes are assigned to distinct clusters.

TABLE I: Simulation Parameters

Parameter	Value	Parameter	Value
Number of nodes (n)	5,000	GRG probability (γ)	0.027
Aggressiveness (β)	1/10.0	Skill set (α)	0.7
Simulation time (T)	10,000	Simulation runs	50

We evaluated the performance and effectiveness of four different host-clustering methods, given below.

- **Random:** Randomly divide the nodes into a fixed number of clusters.
- **Page rank:** Divide the neighborhood of high-page-rank nodes into clusters.
- **Hierarchical clustering:** Divide the graph by greedily optimizing the modularity metric.
- **Spectral clustering:** Identify connected sets of nodes as clusters by computing the graph Laplacian and clustering the top k eigenvectors.

B. Results

In the centralized data fusion architecture, one global collection agent or leader performs all the event processing. This collection agent has a complete view of the system state, but it requires a lot of resources to handle the events from the whole system and is a single point of failure. On the other hand, the hierarchical architecture distributes the load among the clusters. However, every cluster has a limited view of the system, and the global leader agent has a more abstracted view of the whole system.

In order to study the trade-off between performance and quality of system view at cluster leaders in the hierarchical architecture, we varied the number of clusters for different clustering methods and computed the following metrics: 1) resource usage at the global leader, 2) resource usage at cluster leaders, 3) resource fairness among clusters, and 4) locality of graphs at cluster leaders.

To study the resource usage on the global leader, we measured the number of messages sent from the cluster leaders to the global leader. The results in Fig. 3a show that when there is only one cluster, which is the centralized case, all of the messages are processed by the global agent. The number of messages is reduced by a factor of 100 when there are two clusters. For all the clustering methods, the number of messages always decreases as the number of clusters increases. However, the number of messages sent for random clustering is about 10 times higher than for the other clustering methods. The reason is that the trigger event occurs more frequently in random clustering because of the random manner in which nodes are clustered.

Next, we measured the average size of the graph (number of messages) maintained by each cluster leader. As shown in Fig. 3b, for all the clustering methods, the average graph size decreases as the number of clusters increases. The random clustering generates smaller graphs at the cluster leaders than

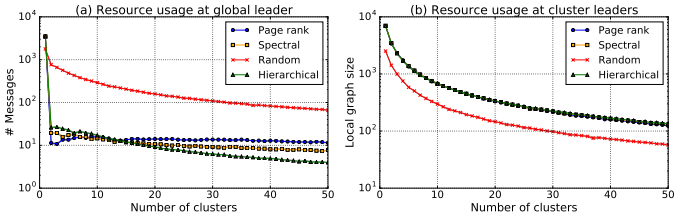

Fig. 3: Evaluation results for (a) number of messages received at the global leader, and (b) local graph size at cluster leaders. The results are averaged over 50 runs of simulation. The envelope around each line is the 95% confidence interval.

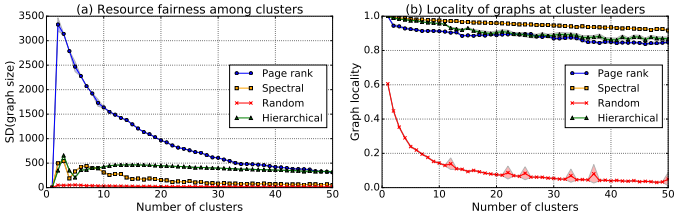

Fig. 4: Evaluation results for (a) standard deviation of local graph size at cluster leaders, and (b) the locality of the graph at cluster leaders. The results are averaged over 50 runs of simulation. The envelope around each line is the 95% confidence interval.

the other clustering methods do. That happens because the random clustering uniformly divides the network into clusters, so each cluster leader will have a small graph.

Ideally, the hierarchical architecture should distribute the communication and processing workload fairly among the clusters. To evaluate the fairness, we measured the standard deviation of the set of graph sizes for all the clusters. Standard deviation describes the distribution of the measurements around the average. If the standard deviation is low, then the resource usage per cluster is balanced. If the standard deviation is high, then the resource usage per cluster is not balanced, and some clusters use more resources than others to maintain the local HC-graphs. As shown in Fig. 4a, we observed that the number of clusters and fairness are not correlated, and that fairness is affected by the clustering method. For each clustering method, when the number of clusters is more than 30, the fairness converges to a singular value. The reason is that each cluster has a very small number of nodes, and thus the local graph is also very small. Moreover, the page rank clustering performs poorly in terms of fairness, giving the highest standard deviation of all the methods because it produces unbalanced clusters due to the nature of the generated random graph.

Finally, we evaluated the locality of the HC-graphs at a cluster leader. The locality metric aims to measure the quality of the graph for use in local response actions. Local response mechanisms use the local HC-graph, whose information is limited to cluster-level causation events. Local response to lateral movement would benefit from HC-graphs that contain more information pertaining to lateral movement within the cluster. Thus, we define the graph locality as the fraction of vertices in the graph that represent internal hosts within the cluster. A larger number for graph locality means the cluster

TABLE II: Overhead (measured using `top`) of the host-level agent implemented using DTrace and Python.

Time period	CPU overhead	DTrace events	Memory usage	Data rate	Graph size
3 hrs	9.09%	328,377	19 MB	1.8 KBps	$\|V\| = 4,584$ $\|E\| = 14,607$

has a better view of the lateral movement within itself. As shown in Fig. 4b, we observed that random clustering has the least locality of the graphs at cluster leaders. The other methods have almost the same locality values. The reason is that random clustering does not consider the network topology when creating the clusters. We also observed that the number of clusters has very little effect on the locality of the graphs.

In summary, for a network size of 5,000 nodes, use of 20–30 clusters achieves an optimal balance among resource usage, fairness, and quality of the local state. With more than 30 clusters, the benefits of the hierarchy start to diminish. All of the clustering methods that utilize underlying graph topology perform well in terms of reducing the resources needed for the global leader. Finally, while random clustering has the best fairness measures, it has the worst quality of local graphs.

C. Host-level Agent Implementation

We used DTrace on OS X to implement a prototype of the host-level agent. The prototype uses probes from the `syscall`, `proc`, and `fsinfo` providers in the kernel to collect process communication information. Information collected from DTrace is piped to a Python program that maintains the PC-graph and prunes the graph when a process is terminated. *python-igraph* is used to implement the PC-graph as a labeled directed graph. We evaluated the implementation over a MacBook Pro (Mid-2012) with 2.6 GHz Intel Core i7 and 16 GB of memory, used as a workstation inside a university network. Table II shows the resources used by the host-level agent. The results show that the CPU overhead is at 9.09% and data are produced with an average rate of 1.80 KBps. This indicates that such a host-level agent, despite the lack of explicit optimization of the collection of host-level activity, is lightweight and uses little resources, making it suitable for practical use.

VI. Related Work

Effective intrusion detection is a necessary component of secure and resilient systems. Multi-sensor fusion techniques [4], [9], [10], [11] aim to improve accuracy and performance of intrusion detection by combining security information from multiple sources. Most of these approaches, however, rely on heuristics and predefined rules about the properties of the underlying system and the behavior of attacks, which change very frequently. These approaches generally fail to detect many sophisticated attacks, such as zero-days and long-lasting targeted attacks [12].

Lateral movement detection can help in detecting intrusions in early phases and preventing significant damage to the system [13], [6]. However, a sophisticated lateral movement attack is difficult to detect, since the attack is targeted and usually uses normal network operations to spread slowly and silently across the system.

Thus, lateral movement behavior is very different from worm propagation, which quickly spreads far and wide. Thus, worm detection techniques [12], [14], [15] that rely on detecting changes in host behaviors or network communication structures may not successfully detect lateral movement. As we show in this paper, more general behavioral features, which provide a holistic view of system-wide activities, are needed to detect such an attack. The distributed fusion framework proposed in this paper enables us to detect such long-lasting and slow-moving attacks in large-scale systems.

The concept of establishing causality among network connections has previously been used to profile normal network communications in order to detect worms. The techniques proposed in the literature detect worms by using anomalous timing and port distributions [12], signatures of known worm packets [14], and rare connections between hosts [15]. However, these methods fail in the context of lateral movement because the attacker may change behavior or use known services and paths to spread through the system. Instead, our approach for finding connection causation chains relies on looking at the kernel-level activities on the hosts and thus allows us to infer the causations more accurately.

Finally, the work in [16] describes an approach to quantitatively determine the level of exposure to certain types of lateral movement that are based on Pass-The-Hash attack. It can be used to help configure a network to minimize exposure to these types of attacks.

Our work, on the other hand, uses high-level behavioral patterns to detect lateral movement in large-scale systems. In particular, our approach collects host-level activities and correlates them with network communications. The hierarchical collection and fusion of this information across all the hosts, which is based on our distributed data fusion framework, then provides the basis for scalability and performance of lateral movement detection.

VII. Discussion and Future Work

Our fusion framework provides a basis for collecting and combining information to detect all types of lateral movement behavior. In this paper, we do not aim to distinguish between malicious and benign lateral movement, since the benign lateral movement can also be used by attackers to achieve malicious goals. Moving forward, we can introduce domain knowledge, such as normal data workflow in the system, into our fusion framework to distinguish between malicious and benign lateral movement.

Evaluation of the security of our fusion framework is not the main focus of this paper, and is being addressed in our ongoing work. Instead, here we discuss the security implications of the fusion framework and the ways to address them.

Our fusion framework relies heavily on the ability of the agents to report accurate information in order to detect lateral movement in the system. This implies that we need to ensure

the security of the information stored and transmitted among the agents. We believe that implementation of such secure agents is feasible in practice through use of technologies such as TrustZone [17]. These technologies encapsulate programs in a secure environment with encryption of memory and disk to provide integrity and authenticity.

In addition to making the agents secure, we also need to consider the situations in which the information collected by the agents is incorrect. The underlying host or network may provide wrong information to the agents because of the effect of an intrusion. We consider this implication in two different cases, loss of information and incorrect information.

The first case, loss of information, may happen when an attacker uses covert channels for communication. For example, information can be encoded in an infinite number of ways using storage or timing channels [18]. Although we cannot claim to detect such well-designed schemes, our framework can be extended to account for the missing information by inferring causation chains based on some probabilistic measures. We plan to incorporate a model for missing data in our fusion framework, similar to what has been shown in [19], [20].

The second case can arise because of malware such as rootkits that can modify the kernel-level state and feed false information to the agents. We believe that we can address this issue by including additional domain knowledge in the fusion framework.

VIII. Conclusion

Intrusion resilience through response and recovery presents a practical solution for system security. To achieve resiliency, we need to monitor the system to estimate the security state, and then select responses that maintain a resiliency metric related to service and intrusions. We present a flexible framework for distributed data fusion aimed at addressing intrusion resilience. The framework defines different components of data fusion: data transformation, dissemination, and abstraction. We use the framework to define a method that uses agents in the system to detect lateral movement. Our method merges host-level communication causation events to create host communication graphs. The merge algorithm exploits the semantics of the causation relation to avoid requiring time ordering on the host-level events. Then, in order to avoid having a centralized collection agent, we cluster the agents into a hierarchy in which each cluster leader maintains local host communication graphs and sends abstracted updates to the global collection agent. We evaluate the performance gains from clustering the agents and distributing the workload for different clustering approaches. Our results show that clustering methods that utilize network topology achieve a good balance between performance and quality of state. We also implement a prototype of the host-level agent and show that the agent is lightweight and suitable for practical use.

This work is the first step towards resiliency for lateral movement. We plan to continue the work by finding methods to detect malicious lateral movement, and then devise response actions to contain such activity.

Acknowledgment

We would like to thank Jenny Applequist for editing the manuscript. This material is based in part on research sponsored by the Air Force Research Laboratory and the Air Force Office of Scientific Research, under agreement number FA8750-11-2-0084.

References

[1] R. Bhatti, R. LaSalle, R. Bird, T. Grance, and E. Bertino, "Emerging trends around big data analytics and security: Panel," in *Proc. of the 17th ACM Symposium on Access Control Models and Technologies*. ACM, 2012, pp. 67–68.

[2] S. Suthaharan and T. Panchagnula, "Relevance feature selection with data cleaning for intrusion detection system," in *Proc. of IEEE Southeastcon*, March 2012, pp. 1–6.

[3] Verizon, "2015 Data Breach Investigation Report," http://www.verizonenterprise.com/DBIR/2015/, 2015.

[4] T. Bass, "Intrusion detection systems and multisensor data fusion," *Commun. ACM*, vol. 43, no. 4, pp. 99–105, Apr. 2000.

[5] E. M. Hutchins, M. J. Cloppert, and R. M. Amin, "Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains," *Leading Issues in Information Warfare & Security Research*, vol. 1, p. 80, 2011.

[6] SANS, "Analysis of the cyber attack on the Ukrainian power grid," http://www.nerc.com/pa/CI/ESISAC/Documents/E-ISAC_SANS_Ukraine_DUC_18Mar2016.pdf, 2016.

[7] D. R. Ellis, J. G. Aiken, A. M. McLeod, D. R. Keppler, and P. G. Amman, "Graph-based worm detection on operational enterprise networks," MITRE Corporation, Tech. Rep. MTR-06W0000035, 2006.

[8] T. Sager, "Killing advanced threats in their tracks: An intelligent approach to attack prevention," SANS Institute, Tech. Rep., 2014. [Online]. Available: https://www.sans.org/reading-room/whitepapers/analyst/killing-advanced-threats-tracks-intelligent-approach-attack-prevention-35302

[9] M. Almgren, U. Lindqvist, and E. Jonsson, "A multi-sensor model to improve automated attack detection," in *Proc. 11th International Symposium on Recent Advances in Intrusion Detection*. Springer-Verlag, 2008, pp. 291–310.

[10] J. Li, D.-Y. Lim, and K. Sollins, "Dependency-based distributed intrusion detection," in *Proc. DETER Community Workshop on Cyber Security Experimentation and Test*. USENIX Association, 2007.

[11] D. Xu and P. Ning, "Correlation analysis of intrusion alerts," *Intrusion Detection Systems*, vol. 38, pp. 65–92, 2008.

[12] V. Sekar, Y. Xie, M. K. Reiter, and H. Zhang, "Is host-based anomaly detection + temporal correlation = worm causality," DTIC Document, Tech. Rep., 2007.

[13] TrendMicro, "Lateral Movement: How do threat actors move deeper into your network?" http://about-threats.trendmicro.com/cloud-content/us/ent-primers/pdf/tlp_lateral_movement.pdf, 2013.

[14] D. R. Ellis, J. G. Aiken, K. S. Attwood, and S. D. Tenaglia, "A behavioral approach to worm detection," in *Proc. 2004 ACM Workshop on Rapid Malcode*. ACM, 2004, pp. 43–53.

[15] N. Kawaguchi, Y. Azuma, S. Ueda, H. Shigeno, and K. Okada, "ACTM: Anomaly connection tree method to detect silent worms," in *Proc. 20th International Conference on Advanced Information Networking and Applications*, vol. 1, April 2006, pp. 901–908.

[16] J. R. Johnson and E. A. Hogan, "A graph analytic metric for mitigating advanced persistent threat," in *Proc. 2013 IEEE International Conference on Intelligence and Security Informatics*, June 2013, pp. 129–133.

[17] ARM, "TrustZone," https://www.arm.com/products/processors/technologies/trustzone/index.php, 2009.

[18] E. Couture, "Covert channels," SANS Institute, Tech. Rep., 2010. [Online]. Available: http://www.sans.org/reading-room/whitepapers/detection/covert-channels-33413

[19] P. Ning, D. Xu, C. G. Healey, and R. S. Amant, "Building attack scenarios through integration of complementary alert correlation methods," in *Proc. 11th Annual Network and Distributed System Security Symposium*, 2004, pp. 97–111.

[20] P. C. Pinto, P. Thiran, and M. Vetterli, "Locating the source of diffusion in large-scale networks," *Physical review letters*, vol. 109, no. 6, p. 068702, 2012.