An Edit-Distance Algorithm to Detect Correlated Attacks in Distributed Systems

Sule Simsek

Abstract**—**Intrusion detection systems (IDS)are crucial components of the security mechanisms of today's computer systems. Existing research on intrusion detection has focused on sequential intrusions. However, intrusions can also be formed by concurrent interactions of multiple processes. Some of the intrusions caused by these interactions cannot be detected using sequential intrusion detection methods. Therefore, there is a need for a mechanism that views the distributed system as a whole. L-BIDS (Lattice-Based Intrusion Detection System) is proposed to address this problem. In the L-BIDS framework, a library of intrusions and distributed traces are represented as lattices. Then these lattices are compared in order to detect intrusions in the distributed traces.

Keywords**—**Attack graph, distributed, edit-distance, misuse detection.

I. Introduction

An intrusion is defined as any set of actions which compromise the integrity, confidentiality, or availability of information and/or resources [4]. Intrusion detection in computer systems has been an active field of research for over three decades. Intrusion detection techniques are classified into two main groups based on the detection mechanism, Anomaly Detection and Misuse (Signature-based) Detection. In this paper, the focus is on one form of misuse detection, distributed-system based misuse detection in which the system inspects message traffic between the processes of a distributed system and detects intrusions that violate the system architecture's security policy. Although the current research focuses on intrusions caused by sequential events, intrusions can also be formed by concurrent interactions of multiple processes. Some of the intrusions caused by these interactions cannot be detected by using sequential intrusion detection methods alone on each process. Therefore, there is a need for a mechanism that is capable of capturing the concurrent interactions of processes as well as detecting intrusions caused by these concurrent interactions. Taking into account the concurrent structure of events is important since distributed systems interact between processors. The distributive lattice [16] is an attractive framework to represent concurrent processor interactions since it is capable of capturing causal interactions between the system components as well as the concurrent interactions between them [3].

Unfortunately, security threats increase exponentially with multiple vulnerabilities [19]. The representation model (lattices) to illustrate these vulnerabilities in this work also grows exponentially. However, this problem can be overcome by limiting the size of the lattice through different methods such as reducing the size of the lattice by using computation slicing method which slices the lattice into a concise sub-lattice [17] and detecting the conjunctive predicates [8], [2] of the lattice.

In the remainder of this paper: Section 2 discusses the related work on attack graphs, distributed intrusion detection systems, and sequence-based intrusion detection. Section 3 motivates the reader to the importance of the problem. In Section 4, fundamental concepts regarding the temporal properties of distributed systems is explained. Section 5 outlines the architecture of the proposed method, L-BIDS. In Section 6, the problem is analyzed and the paper concludes with discussion and future plans.

Sule Simsek is with the Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, USA 65409, email: simsek@mst.edu

II. Related Work

Representing the attacks as graphs is becoming a popular research area. Lippmann *et al.* [22] used attack graphs to show how attackers progress through an enterprise network; [11] represented attacks as graphs to simulate intruders' actions during the design and exploitation stages of computer networks; Gao*et al.* [7] monitored the programs and modeled sequential system call behavior as execution graphs; Tupper and Zincir developed security metrics [19] that use data from attack graphs to decide the attackability of the network.

Over the past years, different methods have been applied to distributed intrusion detection. [24] combined distributed intrusion detection with anomaly-based clustering techniques for completing global information extraction of intruder actions; Wu *et al.*[23] presented BDI(Belief-Desire-Intention) model of intelligent agents to improve the robustness hierarchical distributed intrusion detection system. [21] designed an multi agent-based distributed intrusion detection system which survives the attacks by hiding the main source of the network from the attacker. [9] presented an event sequence based distributed intrusion detection system in which the intrusion detection performance rate is elevated by focusing on the timespace precision, time synchronization and network delay.

Sequence relations between the events of an intrusion have been studied widely. Lee *et al.* applied data mining [14], Forrest *et al.* used human immunology system inspired techniques [10], Lane and Brodley used limited memory models to construct the normal profile of temporal data [13], and [20] represented normal program behavior using system call frequencies.

III. Motivation

In order to elaborate on the differences between sequential and concurrent (distributed) systems and intrusions that occur in these systems we provide two intrusion examples,

one sequential and one distributed. The first example is the *sunsendmailcp* intrusion of the *sendmail* UNIX program. In the *sunsendmailcp* intrusion, first the *sunsendmailcp* script uses a special command line to cause *sendmail* to append an email message to a file. By using this script on a file such as *.rhosts*, a local user may obtain root access [10]. The sequence of events observed in the *sunsendmailcp* intrusion is illustrated in Figure 1. In Figure 1 circles represent events and the vertical line represents the timeline of the events which happen at user1's (*U1*)'s machine. Initially the email message should be appended to a file then the intruder, *U1*, will be able to obtain a root access. The *sunsendmailcp* intrusion of *sendmail* can be detected by sequential intrusion detection systems by observing the sequence of events which constitute the intrusion.

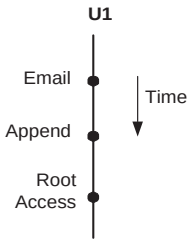

Fig. 1. *sunsendmailcp* intrusion of the *sendmail* UNIX program. Initially, *sunsendmailcp* script causes *sendmail* to append an email message to a file which resides at local user *U1*'s machine then by using this script on *.rhosts*, *U1* may obtain a root access and attack other users' machines in the system.

Intrusions can also be formed by non-sequential events. In other words, the intrusive events may be concurrent with each other. We provide a distributed intrusion example that is resulted from concurrent interactions of processes which is the distributed denial of service (DDoS) attack scenario illustrated in Figure 2. In Figure 2 arrows between processes represent messages. The starting point of the arrowed-line represents the sending of a message, and the ending point of the arrowed-line represents the receiving of a message. The dashed lines represent the propagation of the intrusion. In this correlated, distributed attack, P1 decides to initiate a DDoS attack on P4. In order to achieve its malicious goal, P1 attempts to persuade the other processes in the system to collaborate with it. First P1 sends a message to P3 and receives a message from P3 which confirms P3's willingness to collaborate with P1. Then P1 sends a message to P2. Since all these processes agreed to initiate an attack on P4, they start sending excessive messages to P4 which results in P4 consuming all its resources to try to reply to these excessive messages, therefore results in an attack on the availability of system resources for P4.

On the right side of Figure 3 events on different processes are provided in sequence. The process numbers followed by send (shown by _S after the process number) and receive (shown by _R after the process number) messages represent how a sequential intrusion detection method views the DDoS scenario depicted in Figure 2. Since the intrusive messages sent by P1, P2 and P3 are observed at different processes' timelines, P4 appears to receive three independent messages. This representation fails to show the dependency between

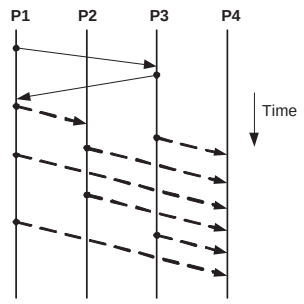

Fig. 2. The distributed denial of service (DDoS) attack scenario which can not be detected with sequential intrusion detection methods. P1, P2, and P3 collaborate on a DDoS attack on P4.

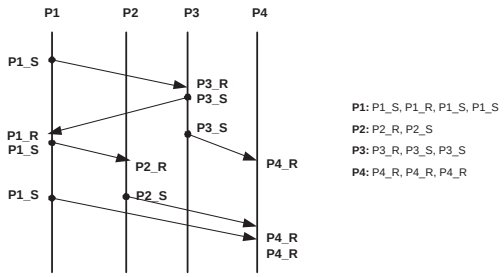

Fig. 3. TimeSpace Diagram of the intrusion scenario including the labeling on send and receive messages (on the left). Observations at each individual process (on the right). The concurrency between intrusive messages can not be captured by sequential intrusion detection methods due to the limited individual process observations.

these virtually unrelated messages on different processes. In other words, the partial order or causality between these intrusive messages can not be detected by sequential intrusion detection methods. On the other hand, by obtaining a global view of the distributed system that observes causality we can detect this intrusion. This is the basis for distributed intrusion detection.

Without the global view of the distributed systems, the intrusions such as the one depicted in Figure 3 gone undetected. In order to capture the relations between multiple messages on different processes and detect intrusions caused by these messages, we propose L-BIDS. The power of the lattice representation is due to the ability of capturing both sequential and distributed intrusions as well as concurrent permutations of the events. Our aim is to represent intrusions and distributed system traces as lattices then compare them to detect to correlated, concurrent intrusions in the distributed system traces.

IV. Preliminaries, Model and Background

Since our aim is to detect distributed intrusions represented in the form of lattices, we provide a brief introduction to the terminology of distributed systems and graph (more specifically lattice) theory in the next sub-sections.

A. Distributed System Preliminaries

"happens-before" Relation. The relation "happens-before" denoted by "$\rightarrow$" on the set of events of a system is the smallest relation satisfying the following conditions:

1) If a and b are events in the same processor, and a comes before b, then a →b; and a & b are sequentially related.
2) If a is the sending of a message by one processor, and b is the receipt of the same message by another processor, then a → b; and a & b are causally related.
3) If a → b and b → c then a → c.
4) If a and b are two distinct events and ¬ (a→ b) and ¬ (b →a); and a & b are concurrent [12]. The symbol ¬ represent the negation logical symbol.

Consistent Global State. The entire system events representing the concurrent states of all processes based on their message interactions.

Vector Time Stamp (VTs). A discrete clock for a process is a monotonically increasing integer by increasing its value at each event occurring at the process. A vector time stamp is a vector of discrete clock values, one from each process in the distributed application [6].

Time-Space Diagram. Modeling of a distributed computation by utilizing the partial order on the events.

Distributive Lattice Representation of the Time-Space Diagram. The collection of all possible consistent states of a distributed computation.

B. Lattice Theory Preliminaries

Lattice. A lattice L is a partial ordered set (poset) P in which every pair of elements (x,y ϵ P) have a greatest lower bound denoted by x $\wedge$ y, and a lowest upper bound denoted by x $\vee$ y [1]. Let *L* be represented as *L(V,T)* such that *V* denote the set of vertices, *T* denote the set of transitions (edges).

Distributive Lattice. Let L be a lattice. L is said to be distributive if it satisfies any of the following identities: (for all x,y,z ϵ L)

1) $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.
2) $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$.
3) $(x \vee y) \wedge (x \vee z) \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$.

V. The Proposed Method

A. Architecture of L-BIDS

Figure 4 depicts the architecture of L-BIDS. In an L-BIDS lattice, nodes represent consistent global states of a distributed system and edges represent the events which are the transitions from one consistent global state to another. The paths in the lattice capture the sequence of events. If the path is split into several forks then we can observe the propagation of distributed intrusion in concurrent global states. In the first phase of L-BIDS; the intrusions are identified as intrusion signatures based on domain knowledge and/or *a priori* knowledge. Then these intrusion signatures are collected into a trace-based audit log. In the second phase, these audit logs are transformed into intrusion signature lattices. In the third phase of L-BIDS, messages between processes of a distributed system are collected at run-time and aggregated into a trace-based audit log. As the fourth phase, this collected distributed trace is transformed into collected trace lattice. The fifth phase of L-BIDS consists of the comparison of the collected trace lattice and current intrusion signature lattice in the library by the lattice matching algorithm. If these two lattices match, an intrusion alert is generated and the intrusion is stopped. On the other hand, if two lattices do not match then the next intrusion signature lattice in the library is accessed and comparison is performed between this intrusion signature lattice and the collected trace lattice. This phase repeats throughout the lifetime of L-BIDS. The transformation of audit logs into distributive lattices is achieved by latGenU program [18] and Graphviz software [5]. The library of intrusions is a dynamic structure in which as the new intrusions are identified this library is updated and extended. The lattice matching phase of L-BIDS is related to the graph edit distance problem[1]. Our goal is to formulate edit operations to apply on the collected trace lattice so that we can obtain a similarity or dissimilarity metric between the collected trace lattice and intrusion lattice.

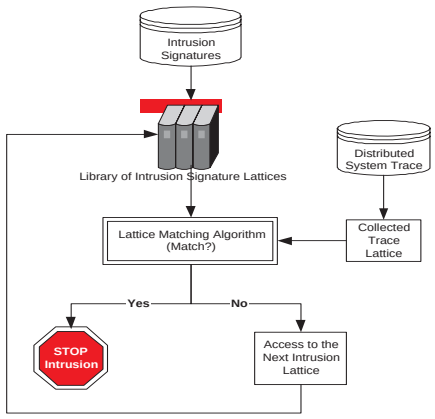

Fig. 4. The architecture of Lattice-based intrusion detection system(L-BIDS)

B. Encoding Scheme

In the distributive lattice representation of the collected distributed trace, the nodes represent consistent global states of a distributed system and edges represent the events which are the transitions from one consistent global state to another. Initially, the nodes of the lattice are composed of VTs of all the processes in the system. Given that the number of processes is *n*, this constitutes a vector of size n^2. Representing the global states with number-dependent VTs is not effective when comparing lattices. Additionally, one state can be represented with specific VTs for a specific scenario and the same state can be represented with different VTs for another scenario. In order to solve this number-dependent and inconsistent representation of global states problem we propose an encoding scheme for lattice edges. This enables us to reason about the meaning of the edges and lattice paths, and ultimately the behavior of the intrusion more effectively and accurately.

[1]Graph edit distance [15] is used to find the similarity between graphs and is defined as transforming one graph into another by performing edit operations and computing their associated costs which resulted in minimum edit costs. A lower cost of transformation represents greater similarity, whereas a higher cost represents greater dissimilarity between the graphs.

The definition of the encoding scheme is given as follows: Each consistent global node in the lattice consists of VTs of all the processes in the system. For a system of *n*-processes, VTs of *n* digits is utilized.

1) For each starting and ending node pairs connected with edges, subtract the starting-node's first digit of VTs which represents the first process' VTs from the ending-node's first digit of VTs.
2) Keep performing the subtraction for all digit-wise numbers of VTs of the starting and ending node pairs. The subtraction resulted in VTs of size *n* in which the non-zero element of the VTs reveals the process which sent/received message.
3) Map the resulting non-zero element of VTs to letters in the alphabet which represent the send and receive events at different processes.

The encoding scheme for events in a distributed system with *n*-processes is formulated as follows:
For $i = 1, 2, \ldots, n$, n is the number of processes.
Let
P$_i$ be the **i**th **process**
Ms$_i$ be the **sending of a message at the i**th **process**
Mr$_i$ be the **receiving of a message at the i**th **process**
Then there exists a one-to-one mapping between the sending and receiving of messages happened at $P_1, P_2, \ldots, P_n$ to the letters in alphabet.

C. Equivalence Edges

Figure 5 represents the rhombus shaped sub-lattice of a distributive lattice. We call the parallel edges which are labeled with the same symbol as "equivalence edges."The advantage of "equivalence edges" (denoted by EE of events send (*S*), receive (*R*)) of the distributive lattices is that: any edit operation which is performed on an edge can simply be performed on the "equivalence edges" of this edge.

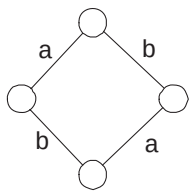

Fig. 5. Two events denoted by *a* and *b* are concurrent. The events denoted by the same symbol which resides at the lower and upper half of the lattice represent the same events (Equivalence edges).

VI. Problem Analysis

The problem of distributed intrusion detection becomes more complicated when an intruder tries to trick the system by sending irrelevant extra messages. Additionally, messages can naturally be interleaved by other processes which results in a similar type of an effect as in the case of the tricking intruder. These irrelevant extra messages within the intrusion scenario can cause the collected trace lattices to be different than the intrusion lattices. However, further comparison of the intrusion lattice and the collected trace lattice can lead to the discovery of essential similarities or equivalences between these lattices. The following discussion relates to the structural lattice differences caused by the embedding of extra irrelevant messages and how to eliminate the effects of these messages. In order to eliminate the effects of these messages, the irrelevant extra messages are classified into three types:

1) Extra messages which preserve concurrency: Embedding of this type of an extra message does not change the relations between the intrusive messages, thus it preserves the concurrency between intrusive messages.
2) Extra messages which transform concurrency into causality: Embedding of this type of an extra message transforms the concurrency between intrusive messages into causality.
3) Extra messages which preserve causality: Embedding of this type of an extra message preserve the causality between intrusive messages[2].

Since our aim is to discover whether the collected trace lattice contains an intrusion, we compare these lattices by using the proposed lattice matching algorithm. The lattice matching algorithms for eliminating the effects of each type of extra message are defined in the following sub-sections. The notation used in these algorithms is given as follows:
Let
LT(L) be the **lattice traversal of a lattice** L
I be the **set of intrusion signature lattices**
i be the **intrusion signature lattice (***i* ϵ *I***)**
c be the **collected trace lattice**
E be the **set of events (***E* = *S* $\rightarrow$ *R***:** *S* **(send event),** *R* **(receive event),** *S***,** *R* ϵ *T*).

A. Preserving Concurrency

In order to transform a concurrency preserving collected trace lattice into an intrusion lattice the following edit operations are proposed as follows:

Delete Edges. Delete *e* (*e* ϵ *E*).

Merge Vertices. Merge v_s^e (the vertex which *e* starts from) with v_e^e (the vertex which *e* ends at) ($v_s^e, v_e^e \epsilon$ *V*).

The recursive function *preserve_cc* is the cost of transforming a collected trace lattice into an intrusion signature lattice by performing the proposed edit operations on *LT(c)* and *E*. The goal is to find the optimum sequence of edit operations which minimizes *preserve_cc*. The application of the edit operations on *LT(c)* and *E* are defined as follows:

1) **DelS:** Apply the "Delete Edges" operation on *S* of *e* and also on all *EE* of *S* in *LT(c)*.
2) **MergeS:** Apply the "Merge Vertices" operation on all v_s^S of *S* and v_e^S of *S* in *LT(c)*.
3) **DelR:** Apply the "Delete Edges" operation on *R* of *e* and also on all *EE* of *R* in *LT(c)*.
4) **MergeR:** Apply the "Merge Vertices" operation on all v_s^R of *R* and v_e^R of *R* in *LT(c)*.

[2]The first type of extra message can also be viewed as this type of extra message. In other words, embedding this type of extra message results in the same differences in the lattices which is also caused by the first type of extra message. Therefore, the series of operations to transform a lattice embedded with the first type of extra message is exactly same as the series of operations to transform a lattice embedded with the third type of extra message.

The associated costs with these operations should be summed up when they are applied on *LT(c)* and *E*. The total cost of these operations (C_{dm}) is defined as follows:
C_{dm}= Cost(DelS)+Cost(MergeS)+Cost(DelR)+Cost(MergeR)

In *preserve_cc*, *E* also represents the candidate set of events which will be deleted from *LT(c)*. The goal is to find specific *e*'s in *E* which minimizes the cost of transforming one lattice into another. The recursive function *preserve_cc* is:

preserve_cc(i, c, E) = min(*preserve_cc(i, c-e, E-e)* + C_{dm},
preserve_cc(i, c, E-e))

Boundary Conditions:

1) if ($E = \emptyset$) and ($c \neq i$)
   preserve_cc(i, c, E)= ∞
2) if ($c = i$)
   preserve_cc(i, c, E)= 0
3) if ($E = \emptyset$) and ($c = \emptyset$)
   preserve_cc(i, c, E)= ∞

The first boundary condition is for the case in which all the candidate events in *E* are considered for deletion and all the operations are also performed on *c*, however, the resulting *c* is still not the same as *i*. Therefore, the cost associated with this case is extremely huge (∞). The second boundary condition applies to the case in which no matter what *E* is, *c* and *i* become the same. In this case, the matching between *i* and *c* is accomplished and the cost of this case is zero. In the third boundary condition, all the candidate events are removed from *c* and *E* and both become empty sets. Since an empty set (*c*) is not equal to a non-empty set (in this case *i*), the associated cost for this case is extremely huge (∞).

As an example consider the time-space diagram of an intrusion signature in Figure 6. Also in Figure 7 the time-space diagram of the concurrency preserving extra message embedded collected distributed trace is depicted. The corresponding distributive lattice of the intrusion signature is depicted in Figure 8. The corresponding distributive lattice of the concurrency preserving extra message embedded collected distributed trace is also illustrated in Figure 9.

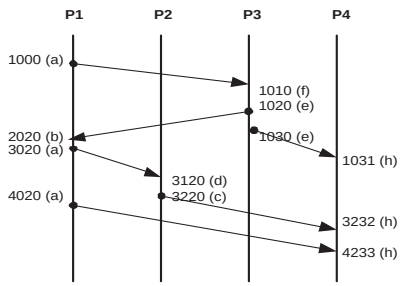

Fig. 6. Time-space diagram of the intrusion scenario.

The proposed edit operations are applied on all combinations of events in the lattice seen in Figure 9. The optimum sequence of edit operations which minimizes *preserve_cc* is found as the one which resulted from applying the proposed operations to the event of *g* followed by *d*. The "Merge Vertices" operation defined in Merge Vertices operation is consistently observed throughout the lattice which results in

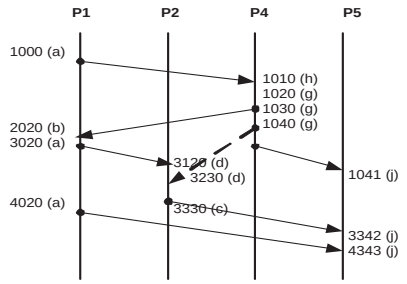

Fig. 7. Time-space diagram of the interleaved intrusion scenario.

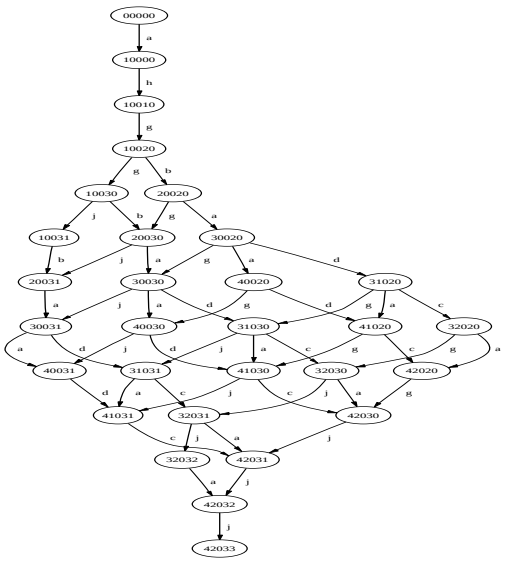

Fig. 8. The corresponding distributive lattice of the intrusion scenario.

a compressed lattice. Therefore, this operation decreases the size (total number of nodes and edges) of the collected trace lattice and make the size of the collected trace lattice exactly same as the intrusion lattice. Applying the "Merge Vertices" operation on the lattice seen in Figure 9 compresses it and reduces the number of its nodes from 42 to 30 which is the number of nodes of the intrusion lattice seen in Figure 8.

B. Transforming Causality into Concurrency

Transforming causality into concurrency requires adding concurrency to a causal system. In order to transform a collected trace lattice into an intrusion lattice by transforming the causality into concurrency the following edit operations are defined as follows:

Delete Edges. Delete $e \epsilon E$.

Move-Up the Edges. Move $e^{'}$ ($e^{'} \epsilon E$) up according to the following rules: If the edge appears before *S* of *e* happens at the same process as *S* and if $e^{'}$ appears after *R* of *e* happens at the same process as *R* then move up $e^{'}$ and attach it to the nearest edge which also happens at the same process as itself. If it does not exist, move $e^{'}$ up and attach it to the root of the lattice.

Update the Vector Timestamps of the Vertices. Update the VTs of the vertices attached to $e^{'}$.

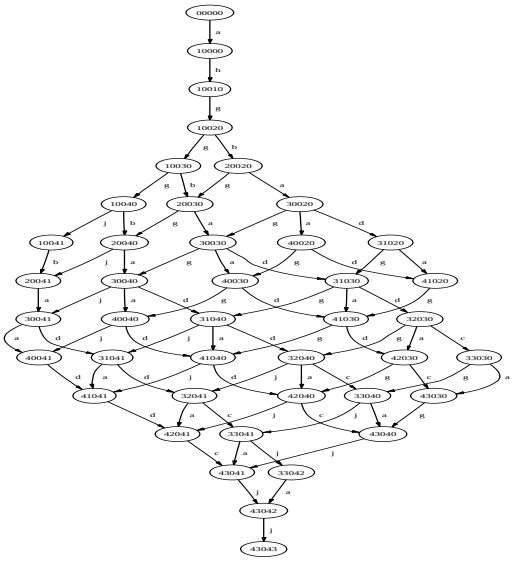

Fig. 9. The corresponding distributive lattice of the interleaving intrusion scenario.

Take the Cross-Product of All Concurrent Vertices. Identify concurrent vertices by examining their VTs and take the cross-product of all concurrent vertices.

transform_cc is the cost of the optimum sequence of edit operations which transforms the collected trace lattice into the intrusion signature lattice. The application of the edit operations on *LT(c)* and *E* are defined as follows:

1) **DelR:** Apply the "Delete Edges" operation on *R* of *e* and also on all *EE* of *R* in *LT(c)*.
2) **MoveR:** Apply the " Move-Up the Edges" operation on all $e^{'}$ of *R* in *LT(c)*.
3) **UpdateR:** Apply the "Update the Vector Timestamps of Vertices" operation on v_e^R of *R* in *LT(c)*.
4) **DelS:** Apply the "Delete Edges" operation on *S* of *e* and also on all *EE* of *S* in *LT(c)*.
5) **MoveS:** Apply the " Move-Up the Edges" operation on all $e^{'}$ of *S* in *LT(c)*.
6) **UpdateS:** Apply the "Update the Vector Timestamps of Vertices" operation on v_e^S of *S* in *LT(c)*.
7) **TakeCP:** Apply the "Take the Cross-Product of All Concurrent Vertices" operation in *LT(c)*.

The associated costs with these operations should be summed up when they are applied on *LT(c)* and *E*. The total cost of these operations (C_{dmut}) is defined as follows:

C_{dmut}= (Cost(DelR) + Cost(MoveR) + Cost(UpdateR) + Cost(DelS) + Cost(MoveS) + Cost(UpdateS) + Cost(TakeCP))

In *transform_cc*, *E* also represents the candidate set of events which will be deleted from *LT(c)*. The goal is to find specific *e*'s in *E* which minimizes the cost of transforming one lattice into another. The algorithm *transform_cc* is:

transform_cc(i, c, E) = min(*transform_cc(i, c-e, E-e)* + C_{dm}, *transform_cc(i, c, E-e)*)

Boundary Conditions:

1) if ($E = \emptyset$) and ($c \neq i$)
   transform_cc(i, c, E)= ∞
2) if ($c = i$)
   transform_cc(i, c, E)= 0
3) if ($E = \emptyset$) and ($c = \emptyset$)
   transform_cc(i, c, E)= ∞

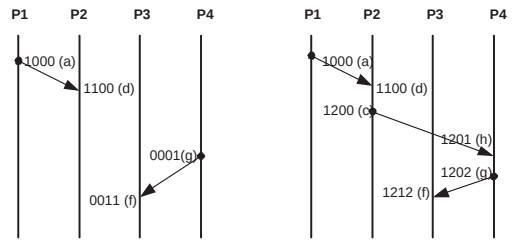

Fig. 10. Time-Space Diagram of the intrusion scenario (on the left) and the Time-Space Diagram of the collected trace scenario (on the right).

The first boundary condition is for the case in which all the candidate events in *E* are considered for deletion and all the operations are also performed on *c*, however, the resulting *c* is still not the same as *i*. Therefore, the cost associated with this case is extremely huge (∞). The second boundary condition applies to the case in which no matter what *E* is, *c* and *i* become the same. In this case, the matching between *i* and *c* is accomplished and the cost of this case is zero. In the third boundary condition, all the candidate events are removed from *c* and *E* and both become empty sets. Since an empty set (*c*) is not equal to a non-empty set (in this case *i*), the associated cost for this case is extremely huge (∞).

In order to elaborate on the application of *transform_cc* we provide a contrived scenario in Figure 10 which shows an intrusion scenario (on the left) and a collected trace scenario (on the right). Figure 11 depicts the corresponding lattices of the scenarios seen in Figure 10. Our goal is to transform the lattice on the right hand-side of Figure 11 into the lattice on the left hand-side of Figure 11 by performing the proposed operations. The sequence of edit operations which minimizes *transform_cc* `is` applied on the collected trace lattice depicted in Figure 11. Figure 12 is obtained by applying the edit operations # 1 and # 2 on Figure 11. Figure 13 is obtained by applying edit operation # 3. Figure 14 is obtained by applying edit operations # 4 and #5. After all concurrent events in Figure 14 are identified and the cross-product of these concurrent events is taken, Figure 14 is transformed into the same graph as shown on the left side of Figure 11. In other words, collected trace lattice is transformed into intrusive lattice, therefore, we can conclude that collected distributed trace is in essence intrusive.

When L-BIDS compares two lattices, it compares the costs obtained from *transform_cc* and *preserve_cc* and chooses the lattice matching algorithm with the minimum cost. The following formula shows this decision:

lattice_matching = min(*transform_cc(i, c-e, E-e)*, *preserve_cc(i, c, E-e)*)

VII. Conclusion and Future Work

It is shown that the proposed lattice edit distance algorithm is capable of eliminating the effects of extra messages in the

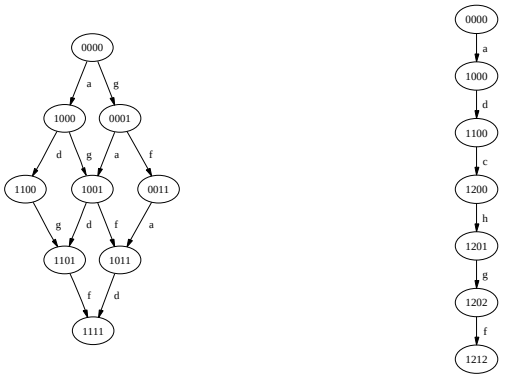

Fig. 11. Lattice Representations of the intrusion scenario (on the left) and the collected trace scenario (on the right).

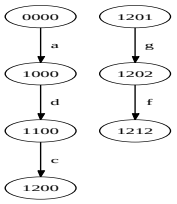

Fig. 12. The extra receive message *h* is deleted and the edges following *h* are moved.

collected trace lattices, therefore, can match two lattices to detect distributed intrusions. Finding edit operations and their associated costs to transform the collected trace lattice into the intrusion lattice, as a means of intrusion detection, provides a quantitative metric for the proposed intrusion detection framework. A higher edit cost reveals dissimilarity between lattices; on the other hand, lower edit cost reveals similarity between lattices. As a future work, we plan to define the lattice traversal algorithm thoroughly and limit the complexity of the lattice comparison by performing the comparisons on regions of lattices rather than entire lattices.

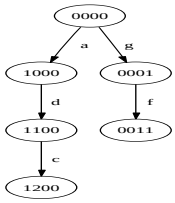

Fig. 13. The VTs of the moved-up edges are updated. Two sub-lattices are connected with each other by attaching the edge *g* to the root of the lattice.

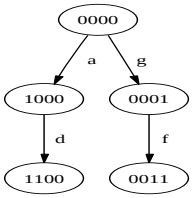

Fig. 14. The extra send message *c* is deleted and the edges following *c* are moved-up (in this case no edges are moved-up).

REFERENCES

[1] G. Birkhoff, *Lattice Theory*, 3rd ed., ser. American Mathematical Societ Colloquium Publications. NY, USA: American Mathematical Society, 1967, vol. 25.

[2] P. Chandra and A. D. Kshemkalyani, "Distributed algorithm to detect strong conjunctive predicates," *Inf. Process. Lett.*, vol. 87, no. 5, pp. 243–249, 2003.

[3] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proceedings of the 1991 ACM Workshop on Parallel and distributed debugging*. New York, NY, USA: ACM Press, 1991, pp. 167–174.

[4] D. E. Denning and P. G. Neumann, "Requirements and model for IDES - a real-time intrusion expert system," SRI International, Computer Science Lab, Tech. Rep., August 1985.

[5] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz - open source graph drawing tools," *Lecture Notes in Computer Science*, vol. 2265, 2002.

[6] C. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," in *Proc. 11th Australian Computer Science Conference*, 1988.

[7] D. Gao, M. Reiter, and D. Song, "Gray-box extraction of execution graphs for anomaly detection," in *Proceedings of the 11th ACM Conf. on Computer and Communications Security*. New York, NY, USA: ACM, 2004, pp. 318–329.

[8] V. Garg and C. Chase, "Distributed algorithms for detecting conjunctive predicates." in *ICDCS*, 1995, pp. 423–430.

[9] L. Guoyuan, H. Hao, and C. Tianjie, "Issue of event sequence in time of distributed intrusion detection system," in *Proceedings of the 2007 Network and Parallel Computing Workshops*. Dalian, China: IEEE Computer Society, 2007, pp. 215–222.

[10] S. A. Hofmeyr and S. A. Forrest, "Architecture for an artificial immune system," *Evol. Comput.*, vol. 8, no. 4, pp. 443–473, 2000.

[11] I. V. Kotenko and M. Stepashkin, "Attack graph based evaluation of network security," in *Comm. and Multimedia Security*, ser. Lecture Notes in Computer Science, vol. 4237. Springer, 2006, pp. 216–227.

[12] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[13] T. Lane and C. E. Brodley, "An empirical study of two approaches to sequence learning for anomaly detection," *Mach. Learn.*, vol. 51, no. 1, pp. 73–107, 2003.

[14] W. Lee and S. J. Stolfo, "A framework for constructing features and models for intrusion detection systems," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 4, pp. 227–261, 2000.

[15] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics - Doklady*, vol. 10, no. 8, pp. 707–710, February 1966.

[16] F. Mattern, "Virtual time and global states of distributed systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. Elsevier Science Publishers B.V., 1989.

[17] N. Mittal and V. Garg, "Techniques and Applications of Computation Slicing," *Distributed Computing*, vol. 17, no. 3, pp. 251–277, 2005.

[18] S. Shivashankaraiah, "Latgenu - lattice generator for unix," Computer Science Department, Missouri University of Science and Technology, Tech. Rep., 2003.

[19] M. Tupper and A. N. Zincir-Heywood, "Vea-bility security metric: A network security analysis tool," in *ARES*, 2008, pp. 950–957.

[20] S. M. Varghese and K. Jacob, "Anomaly detection using system call sequence sets," *Journal of Software*, vol. 2, no. 6, pp. 14–21, 2007.

[21] S. Vongpradhip and W. Plaimart, "Survival architecture for distributed intrusion detection system (dids) using mobile agent," in *NCA*, 2007, pp. 332–338.

[22] L. Williams, R. Lippmann, and K. Ingols, "An interactive attack graph cascade and reachability display," in *Proceedings of the Workshop on Visualization for Computer Security*, 2007, pp. 97–104.

[23] J. Wu, C. Wang, J. Wang, and S. fu Chen, "Dynamic hierarchical distributed intrusion detection system based on multi-agent system," in *WI-IATW '06: Proceedings of the 2006 International Conference on Web Intelligence and Intelligent Agent Technology*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 89–93.

[24] Y.-F. Zhang, Z.-Y. Xiong, and X.-Q. Wang, "Distributed intrusion detection based on clustering," in *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, vol. 4, 2005, pp. 2379–2383.