MTR230165

MITRE TECHNICAL REPORT

MITRE

A Coordination Model for Attack Graphs

Sponsor: MITRE
Dept. No.: L527
Project No.: IRD100.23.C1.0DA

Bedford, MA

Author: Paul D. Rowe
Suresh K. Damodaran
Peter Malinovsky

May 2023

Abstract

Attack graphs have been proven to be useful for modeling multi-stage attacks for vulnerability analysis, though their use in threat emulation has been hindered by multiple challenges. In this paper, we propose a new type of graph, Activation, Guard, and Effect (AGE) graph to support emulation of multi-stage attacks. We describe the abstract syntax and execution semantics of AGE graphs, and provide examples that illustrate the ability of AGE graphs to model attacks and enable attack execution automation.

This work was funded by MITRE's Independent Research and Development Program.

Contents

1 Introduction

An attack graph describes the actions an attacker can take on a target system to induce an event or state, called a goal condition, desired by the attacker. Attack graphs have been used to describe complex multi-stage attacks in multiple domains [14, 26, 28, 10], and to describe defenses [17]. The primary application area of attack graphs has been for vulnerability analysis [24, 11, 20, 17]. Applying attack graphs to automated emulation of multi-stage attacks such as those with the tactics described in MITRE ATT&CK [27] would be an attractive prospect. Threat emulation is useful for simulated penetration testing [9], and for evaluating attacker strategies [1]. However, there are a few challenges that must be addressed to enable such an automator.

The first challenge is that the attack graph execution will need to respect the partial ordering and nondeterminism that arises from the dependencies and logical conditions in the attack graph. The second challenge is that during an attack the target system will likely undergo state changes due to the attacker's actions, the defender's actions, or the actions of users of that system. Therefore, prior to applying the attacker's actions, also called effects in this paper, the automator must always reevaluate the system state before applying an action, both to understand the success of the previous action and to ensure the preconditions of the action are met. Consequently, the attack representation must allow for state changes in the target system that are not predicted at the beginning of the attack by the attacker. The third challenge is that while attack graph representations such as [14, 26] represent attack plans, they only contain preconditions for the actions, and do not contain the attacker's mission requirements such as time or other mission constraints to conduct attacker's actions. The fourth challenge is that sometimes the attack effect may not terminate, or the effect may be fleeting and leaves no permanent trace. The fifth challenge is that the attack representation must be machine readable. This issue has been addressed for the purpose of vulnerability analysis by tools such as MulVal [20], and other ontological techniques for scalability [18].

We are not aware of any general-purpose attack graph representation with precise execution semantics that can be used for automated execution and addresses the challenges above. This paper describes a coordination model, using the novel Activation, Guard, Effect (AGE) graph-based representation, for attack graph automation that addresses these challenges, inspired by

coordination models such as the PTIDES model [31]. This paper focuses primarily on how AGE graphs address the first four challenges described above.

Our contributions are: (1) the definition of the AGE graph and its execution semantics, the first coordination model for attack graphs, and (2) the development of a simulator for attack graph execution based on AGE graphs, along with examples of attack graphs and their execution sequences.

The rest of the paper is organized as follows. An example of a sample attack graph from literature is presented along with an alternate graph more suitable for graph execution in Section 2. Section 3 defines AGE graphs and their execution semantics. We discuss the AGE graph simulator and some examples in Section 4. Background and related work are discussed in Section 5; and we conclude in Section 6.

2 Motivating Example

To motivate the definition of AGE graphs in the next section, we will use the multi-stage attack graph from Zeng et al. [30], reproduced in Figure 1. The target system network topology consists of the Internet; DMZ (demilitarized zone) that has a DNS Server (DS), and a Web Server (WS); and the trusted subnetwork that contains an FTP Server (FS), a SQL Server (SQLS), and an Administrative Server (AS). The attacker exploits some of the vulnerabilities in these servers described in the Common Vulnerabilities and Exposures (CVE) [22] database. We refer to each stage in the attack in Figure 1 as an *attack step*.

As shown in Figure 1, the attacker exploits WS using the vulnerability CVE-2015-1635 that allows remote attackers to execute arbitrary code via crafted malicious HTTP requests. To ascertain the existence of this vulnera-bility in WS, the attacker would have scanned the WS, shown in Figure 1 as the node labeled CVE-2015-1635. Once the attacker gets the ability to exe-cute arbitrary code in the Web Server, the attacker simultaneously scans FS for CVE-2012-2526 & CVE-2013-4465, and SQLS for CVE-2014-1466. These parallel activities will form a *split* in the attack graph. Only one of the two parallel paths attacking FS needs to succeed, represented as an OR condition in the attack graph. The attack step that results in the OR condition being true and the successful exploitation of CVE-2014-1466 of the SQLS are both needed for the attacker to succeed. This fact becomes an AND condition,

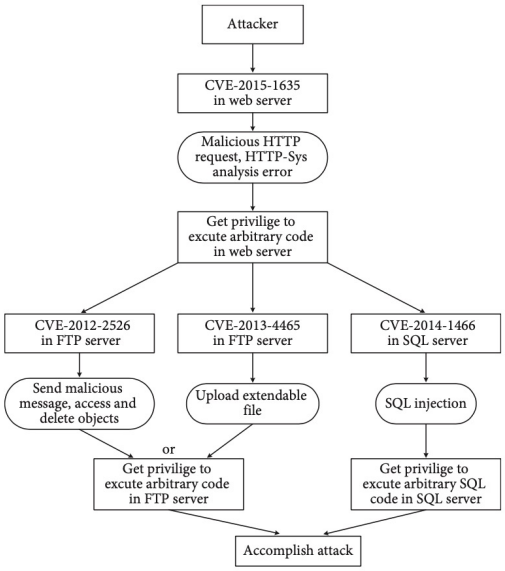

Figure 1: Sample Attack Graph from [30]

though not explicitly marked in Figure 1. We now make some observations on this graph, related to the challenges discussed in Section 1.

There is a partial order of execution in the attack graph. Without the successful exploitation of the Web Server, exploitation of the FS or SQLS cannot take place. Therefore, prior to executing the FS or SQLS exploitation, the attack graph automator must look for an indication of success, or *firing* of the WS exploitation. The target system state may have changed in a way unpredictable by the attacker prior to applying an exploitation to WS or FS. Therefore, a *guard* condition must be evaluated prior to applying an exploitation to ascertain that the right preconditions for its application exist in the target system. However, there is a need to guarantee that the WS exploitation succeeded prior to *activating* the next stage of FS and SQLS exploitations. For the attacker, this guarantee may be just waiting for a certain period of time for the exploitation to succeed, or verifying from system logs that the exploitation code did indeed succeed in gaining the desired access to FS and SQLS. To satisfy a mission requirement, an attacker may even decide to wait for several days. Therefore, the *activation* of the next step of an attack is a decision that the attacker may make based on the attack mission needs, and target system state. In contrast, the *guard* condition is a

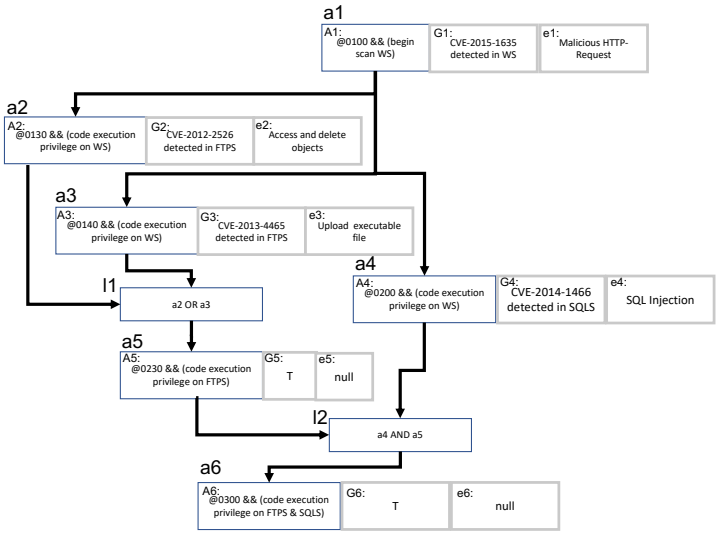

Figure 2: Revised Attack Graph

purely technical precondition, to find the suitable state of the target to apply an effect.

Another important consideration is that while doing threat emulation, it is not possible to guarantee that an attacker effect such as a malware execution will always terminate, and therefore waiting to activate the next stage of attack until a previously applied effect completes is not always a feasible graph execution strategy. Indeed, in most cases, the target system does not send out an alert that the exploitation is successful. Further, if the proof of completion of a time sensitive attack effect disappears before activating the next attack step, once again, waiting to activate the next stage of attack until a previously applied effect completes execution can be problematic. Also note that as the attacker waits, the target system may change and a desired precondition for the next attack step may occur with no effort from the attacker. Therefore, we need a graph representation that addresses these timing issues.

Consider Figure 2, which is an alternate representation of the same attack. Each node representing an attack step in this graph has three rectangles. We refer to the first rectangle in each attack step as an *activation* node. This node incorporates the *activation* condition, such as timing requirements, or indicators for completion of the previous attack step by evaluating the target

system state. For example, the activation node *a2*, has the activation condition *A2*, that indicates the time to start this attack step is at time 0130, if the code execution privilege existed on the Web Server. The *activation* node addresses the third challenge described in Section 1 on incorporating mission requirements in the attack graph. Many of the second rectangles in each node in Figure 2 are readily recognized from the labels in Figure 1. These nodes are the *guard* conditions. The *guard* conditions evaluate the state of the target system, typically with scans or logs, prior to applying an effect. The last rectangle in each attack step is the effect, such as making a malicious HTTP request, for node *a1*. The second and third rectangles are grouped together and named a *guarded effect* node, since this node incorporates both the precondition of the effect, and the effect itself.

As discussed earlier, there is a need to evaluate conditional logic of OR and AND to evaluate the *firing* of the previous attack step. Such conditional logic is represented using an explicit *conditional logic* node. The evaluation of the logic can be subject to timing issues such as the time taken to detect an activation condition, or signal delays from the firing of the previous attack step, and cause nondeterministic execution of the effects. Another way an attack graph exhibits nondeterminism is through the OR logic. For example, it is sufficient to have either of the FS exploits to succeed to reach the next attack step, and therefore, either or both effects may be applied, and sometimes one of the effects may be applied much later. These are situations that occur in practice. The attack graph shows concurrency and synchronization in the execution - the SQLS and FS exploitation may concurrently occur, and the AND condition allows for synchronization. A split is a mechanism in an attack graph to indicate concurrent paths of execution. Also note that as the attack proceeds, some of the guard conditions in previous steps may succeed, and therefore the application of multiple effects can occur in the target system. Therefore, the activation conditions must be chosen carefully to allow for only the effects that are needed at the right time.

In summary, an executable attack graph representation must be capable of depicting the partial order of execution, respect the conditional logic of activations, and allow for concurrency in activating multiple paths of next attack step actions. Figure 2 preserves the partial order of execution, concur-rency, and conditional logic in Figure 1, while also allowing for representing nondeterminism.

Depending on the time scale of an attack and the target system attributes, evaluations of activation conditions or guard conditions may use streaming

analytics on the data streams of logs from the target system [13, 5], or other analytics approaches. Such an evaluation may even be with vulnerability scanning tools such as Nessus [29]. Generically, we refer to these evaluations as *watchpoints* in this paper. The overall purpose of using the watchpoint is to ensure that the target system state or its environment is in the right condition at the right time, prior to applying an effect.

The attack graph representation in the sample example graph has only one starting node and one goal node. However, in reality, there can be multiple start nodes through which an attacker may enter, and there may be multiple goal states, any of which satisfies a successful outcome for an attacker. Such attack graph representations must have precise abstract syntax and semantics to enable machine automation with clearly understood rules, which is the topic of the next section.

3 AGE Graph Semantics

In this section, we present the abstract syntax of AGE graphs and introduce an execution semantics. Before diving into the details of AGE graphs, we note that the material below is parametric with respect to *watchpoints* and *effects*. That is, we treat watchpoints and effects as abstract elements of two separate syntactic classes Ω and $\mathcal{E}$ and create a well-defined execution semantics around these abstract notions. Nevertheless, it will be helpful for the reader to keep some concrete examples in mind.

Definition 1. *A* watchpoint *generates a signal that is sent to (an execution engine of) a graph when corresponding conditions about the target system (and/or its surrounding environment) are met. When the conditions for a watchpoint signal to be sent are met, we say the watchpoint is* triggered. *The syntactic class of watchpoints is denoted* Ω *and we often use* ω *or* ω_i *to denote an arbitrary watchpoint.*

Following the example from the previous section, when the time is past 0130 and the attacker has code execution privileges on WS, a watchpoint would be triggered and sent to the AGE graph. We assume the watchpoints are distinguishable from each other so the AGE graph knows which conditions have been met. The actual mechanism for verifying that the conditions have been met is outside the scope of the current paper, however for time

sensitive attack graphs, analytics detection engines such as those described in [2, 6, 13, 5] could be used.

Definition 2. *An* effect *is a sequence of actions that alter the target system or its environment is some way. The syntactic class of effects is denoted $\mathcal{E}$ and we often use e or e_i to denote an arbitrary effect.*

The effects are the actions that are being coordinated by an AGE graph. As seen in the example from last section, this may be as simple as a single command to upload an executable file, or it could be some set of steps to access and delete a subset of files. By only executing effects once certain pre-conditions have been met (as represented by watchpoints) we aim to enable fined-grained control of when to initiate and when to delay the execution of effects.

3.1 Abstract Syntax

We consider labeled, directed graphs $\mathcal{G} = (N, E, \ell)$ where N is a finite set of nodes, $E \subseteq N \times N$ is a (finite) edge relation among nodes of N, and $\ell : N \to L$ is a labeling function into some set of labels L. We begin by describing the abstract syntax of AGE graphs.

We assume that elements of N have one of the following three forms: $\mathtt{a}_i$, $\mathtt{g}_i$ or $\mathtt{b}_i$ for some natural number i. The set of labels L will also come in 3 types. Nodes $\mathtt{a}_i$ will be labeled with *activation watchpoints* ω that embody the activation conditions discussed in Section 2. Nodes $\mathtt{g}_i$ will be labeled with *guarded effects* which are pairs (ω, e) where ω is a watchpoint, embodying the guard condition discussed in Section 2, and e is an effect. And nodes $\mathtt{b}_i$ will be labeled with Boolean expressions built out of the Boolean connectives $\wedge$ and $\vee$ and identifiers $\mathtt{a}_i$. Nodes of the first type are called *activation nodes*, nodes of the second type are called *guarded effect nodes*, and nodes of the third type are called *logic nodes*. We only consider acyclic graphs, and we impose some extra constraints on the in- and out-degree of nodes according to their labels.

Definition 3. *A graph $\mathcal{G}$ is an* Activation, Guard, Effect *(or AGE) graph iff $\mathcal{G}$ is acyclic and satisfies the following conditions:*

1. *Every activation node has exactly one guarded effect node as a successor.*

2. *Guarded effect nodes have in-degree 1 and out-degree 0. (By item (1), the unique predecessor of a guarded effect node must be an activation node.)*

3. *The immediate predecessors of a logic node are exactly those nodes appearing in its label.*

4. *$\mathcal{G}$ has no edges between logic nodes. That is, for all i and j, $(\mathtt{b}_i, \mathtt{b}_j) \notin E$.*

The graph in Fig. 2 is an AGE graph. In that representation we have grouped activation nodes and guarded effect nodes into triples (with the two parts of the label of a guarded effect node each having its own rectangle). We have suppressed the edges between activation nodes and the unique guarded effect nodes associated with them.

When a logic node has in-degree 0, its label will be $\top$, the always true condition. Condition 4 of Def. 3 is convenient to ensure compact representations, and it eases aspects of the presentation, but is not essential. We could just as easily decompose a single logic node into a subgraph that exhibits the parse-tree structure of the label.

To a graph $\mathcal{G} = (N, E, \ell)$, we also associate a set of *start nodes* $I \subseteq N$. While a natural default choice for I is the set of nodes with in-degree 0, we actually let I be any set of nodes. In particular, it may contain nodes that are not at the start of the graph, which allows us to begin graph execution at arbitrary points. Similarly, we don't always assume that every node with in-degree 0 is a start node. This allows us to explore the consequences of relying on only a portion of the attack.

We additionally identify a set of *goal nodes*. These are activation nodes with out-degree 1, and hence, by the above conditions, their only child is a guarded effect node. Typically, a goal node will be an activation node whose watchpoint verifies that some goal condition has been reached in the target system. Thus, as in Fig. 2, the guarded effect node will often have label ($\top$, Null) that acts as a no-op. We allow graphs with several goal nodes, in which case reaching any of them will be considered a success.

3.2 Semantics

A key goal we have with AGE graphs is to provide a well-defined execution semantics. Before jumping into the formal details, we start with an informal description of how it works. At any stage in an execution, some of the nodes

Figure 3: The activation stages of AGE nodes.

of the graph may be *activated*, others will not yet be activated, and the rest will have already *fired* (or executed). We start with a graph in which the start nodes are activated. When an activation node or guarded effect node is activated, it will fire when the watchpoint in its label is triggered. Guarded effect nodes will also execute their effects when they are fired. All logic nodes start out as activated. An activated logic node can fire whenever the Boolean condition in its label is satisfied. Each node will fire at most one time, so once a node has fired it will never be activated again. As nodes fire, we emit information about the node (such as its watchpoint) into a trace. Logic nodes fire silently, emitting nothing into the trace.

In this way, the conditions that hold in the target system and the environment propel the nodes through stages of being inactive, then active, then fired (see Fig. 3). The sequence of activations is constrained (but not deter-mined) by the partial order implicit in the graph structure modulo the use of disjunction in logic nodes. That is, nodes will not be activated until all their predecessors fire with the exception provided by logic nodes using dis-junction that allow certain subsets of predecessors to fire before progressing. The effects that are executed in guarded effect nodes will typically alter the target system creating a feedback loop that, while constrained by the partial order, will not be entirely predictable.

We now proceed to a more formal treatment of this semantics. The semantics of AGE graphs is given by a labeled state transition system in which the states are triples (B, A, F) as defined below.

Definition 4. *A* graph state *of a graph* $\mathcal{G} = (N, E, \ell)$ *is a triple* $S = (B, A, F)$ *where* $B \subseteq N$ *represents the subset of logic nodes that have not yet fired,* $A \subseteq N$ *represents the set of* activated *guarded effect and activation nodes, and* $F \subseteq N$ *represents the set of* fired *guarded effect and activation nodes.*

A graph state is almost a partition of the nodes of the graph, but not quite. Activation nodes and guarded effect nodes that have not yet been activated are not represented. Also, logic nodes that have fired are not represented. An alternative definition of graph state could account for these nodes, but it

would unnecessarily complicate the semantics. Our labeled state transition system will guarantee that B, A, and F remain disjoint throughout any execution.

Auxiliary functions. To define the execution semantics, we rely on a few auxiliary "access" functions which we define next.

$$\mathsf{wp}_{\mathcal{G}}(n) := \begin{cases} \omega & \text{if } \ell(n) = \omega \\ \omega & \text{if } \ell(n) = (\omega, e) \\ \bot & \text{otherwise} \end{cases} \qquad \mathsf{eff}_{\mathcal{G}}(n) := \begin{cases} e & \text{if } \ell(n) = (\omega, e) \\ \bot & \text{otherwise} \end{cases} \tag{1}$$

We start with the function $\mathsf{wp}_{\mathcal{G}}$ that returns the watchpoint (if any) associated with a node. Similarly, the function $\mathsf{eff}_{\mathcal{G}}$ returns the effect (if any) associated with a node.

Activation nodes and guarded effect nodes will fire when they are activated and their watchpoint is triggered. Logic nodes can fire only if their logical condition (i.e., their label) is satisfied. The logical condition is viewed as representing which combinations of activation nodes must have fired for the logic node to fire. Thus, the logical conditions are evaluated against the set F of fired nodes as follows.

$$\begin{aligned} &F \models \top \\ &F \models \mathsf{a} && \textit{iff } \mathsf{a} \in F \\ &F \models \varphi_1 \wedge \varphi_2 && \textit{iff } F \models \varphi_1 \text{ and } F \models \varphi_2 \\ &F \models \varphi_1 \vee \varphi_2 && \textit{iff } F \models \varphi_1 \text{ or } F \models \varphi_2 \end{aligned} \tag{2}$$

The first line says that when the label of a logic node is $\top$, it is satisfied by any fired set F. The second line says that F satisfies a exactly when a has been fired (and hence is in the fired set). The next two lines define the meaning of conjunction and disjunction in the usual way.

For a given node $n \in N$, we define the following sets.

$$A^{\mathcal{G}}_{\omega} := \{n \in A \mid \mathsf{wp}_{\mathcal{G}}(n) = \omega\} \tag{3}$$

$$\mathsf{next}_{\mathcal{G}}(n) := \{n' \in N \mid (n, n') \in E \text{ and } n' \text{ is not a logic node}\} \tag{4}$$

The first one is all the nodes in A whose watchpoint is ω. The second is the set of all children of a given node that are not logic nodes (that is they are activation nodes or guarded effect nodes) and that have not already fired. It

is the use of $\mathsf{next}_{\mathcal{G}}(n)$ to determine which nodes will be activated when n fires that ensures our semantics will be constrained by the partial order implicit in the graph. We extend functions to sets in the usual way. Namely, for any set X, and any function f we let $f(X) = \{f(n) \mid n \in X\}$. This allows us to apply $\mathsf{wp}_{\mathcal{G}}$ and $\mathsf{eff}_{\mathcal{G}}$ to sets of nodes.

Transition system. Using these notions just defined, we are ready to define the execution semantics of AGE graphs. As mentioned above, the semantics is given as a labeled transition system on graph states (B, A, F). There are two types of transitions: *external* transitions that execute when a watchpoint is triggered, and *internal* transitions that require no external trigger.

An external transition will fire one or more active nodes when their watchpoint is triggered. If the watchpoint triggered is the guard watchpoint of an activated guarded effect node $\ell(\mathsf{g}_i) = (\omega, e)$, then its associated effect e, will also execute. Then an external, labeled transition $(B, A, F) \xrightarrow{l} (B', A', F')$ can occur when a watchpoint ω is triggered and $A^{\mathcal{G}}_{\omega}$ is not empty. The transition satisfies

$$\begin{aligned} l &= (\omega, \mathsf{eff}(A^{\mathcal{G}}_{\omega})) \\ B' &= B \\ A' &= (A \cup \mathsf{next}_{\mathcal{G}}(A^{\mathcal{G}}_{\omega})) \setminus (A^{\mathcal{G}}_{\omega} \cup F) \\ F' &= F \cup A^{\mathcal{G}}_{\omega}. \end{aligned} \tag{5}$$

This says that when ω is triggered, any activated nodes with watchpoint ω will fire, being moved from the set of activated nodes to the set of fired nodes, and replaced by any children that are not logic nodes. The label l emits into a trace which watchpoint was triggered and any effects that are executed as a result of the transition.

An internal transition will fire a single logic node. For a graph state (B, A, F), if $n \in B$ and $F \models \ell(n)$, then the internal transition $(B, A, F) \xrightarrow{\varepsilon} (B', A', F')$ is possible where

$$\begin{aligned} B' &= B \setminus \{n\} \\ A' &= (A \cup \mathsf{next}_{\mathcal{G}}(n)) \setminus F \\ F' &= F. \end{aligned} \tag{6}$$

This says that, when logic nodes fire they are removed from the set of unfired logic nodes, and all successor nodes are activated, provided they have not already fired. (By the conditions in Def. 3, such nodes will not be logic

$$\frac{A^{\mathcal{G}}_{\omega} \neq \varnothing}{(B, A, F) \xrightarrow{(\omega, \mathsf{eff}(A^{\mathcal{G}}_{\omega}))} (B, (A \cup \mathsf{next}_{\mathcal{G}}(A^{\mathcal{G}}_{\omega})) \setminus (A^{\mathcal{G}}_{\omega} \cup F), F \cup A^{\mathcal{G}}_{\omega})} \tag{7}$$

$$\frac{n \in B \qquad F \models \ell(n)}{(B, A, F) \xrightarrow{\varepsilon} (B \setminus \{n\}, (A \cup \mathsf{next}_{\mathcal{G}}(n)) \setminus F, F)} \tag{8}$$

Figure 4: Transition rules for AGE graphs.

nodes.) The logic nodes are not added to the set of fired nodes when they execute, so F remains unchanged.[1] This transition is labeled by ε indicating a silent transition.

These two rules characterize all the possible transitions of graph states. We can summarize this as the two inference rules depicted in Fig. 4.

Traces. The initial graph state of a graph $\mathcal{G} = (N, E, \ell)$ is $(B, I, \varnothing)$, where $B \subseteq N$ is the set of all logic nodes of $\mathcal{G}$, and I is the set of start nodes. An *execution* of a graph $\mathcal{G}$ is a sequence $S_0 \xrightarrow{l_0} S_1 \xrightarrow{l_1} \cdots \xrightarrow{l_{n-1}} S_n$ for $n \geq 0$, such that S_0 is the initial graph state, and for all $i < n$, $S_i \xrightarrow{l_i} S_{i+1}$ is a valid transition. If several transitions are enabled, the execution chooses among them nondeterministically. We can focus on various aspects of an execution by extracting different portions. A *verbose trace* of a graph $\mathcal{G}$ is any sequence $\langle l_0, l_1, \ldots, l_{n-1} \rangle$ such that there exists an execution with those labels. Verbose traces also list all the "silent" label ε. A *trace* is the projection of a verbose trace onto the non-silent transitions. The *triggering sequence* of a trace is the projection onto the first components of the trace. Thus a triggering sequence is a sequence of watchpoints that cause state transitions in the graph. An *effects trace* is the projection of a trace onto the non-empty second components. Effects traces record only the sequence of effects that execute, but not the order of activations. From a trace, we can extract a *firing sequence* that contains the sequence of node firings. Specifically, if we let F_i be the set of fired nodes in state S_i, the activation firing sequence will be $\langle (F_1 \setminus F_0), (F_2 \setminus F_1), \ldots, (F_n \setminus F_{n-1}) \rangle$. As internal transitions will cause $F_i \setminus F_{i-1}$ to be empty, we will typically remove the empty sets from the

[1] This aspect of the semantics assumes Condition 4 of Def. 3 is in effect. If we allow logic nodes to have other logic nodes as predecessors, we must add fired logic nodes to F.

sequence. In the following section, we further remove guarded effect nodes from this sequence to focus on the activation node firing sequence.

4 Examples and AGE Graph Simulator

The target system behavior may change when a watchpoint generates an alert, in either activation nodes, or guarded effect nodes. This behavior can result in zero or more traces of execution of an AGE graph. The discussion in the previous section did not consider timing delays in watchpoint triggering due to target system behavior, or the influence of implementation policy in the automation of an attack graph.

An AGE graph simulator that implements the transition rules described in Section 3 to generate all execution traces that reach the elements of a goal node set from the start node set will be helpful in seeing the result of these delays. The input to the simulator is a machine-readable description of an AGE graph, and the output is a set of execution traces. The simulator can be used to explore reachability from an element of the start node set to an element of the goal node set. The simulator can also be used to see all the unique traces that an AGE graph allows to identify any unexpected or undesirable traces. In this section, we explore some of the applications of the simulator.

By applying the simulator to Fig. 2, we determined that it will permit 22 unique activation node firing sequences, assuming all activation nodes fire, depending on the time taken for firing the watchpoints A2, A3, A4, and A5 in the activation nodes, a2, a3, a4, and a5, respectively. In the real world, the number of firing sequences will be much smaller because the firing of some activation nodes may require the successful execution of effects in the previous attack step. Further, some of the watchpoints may never fire, and the attack may not succeed. Therefore, there could be many more firing sequences, but any such sequence will form a prefix of these 22. For brevity, we examine a few of the interesting sequences in Table 1 below. Sequence 1 in Table 1 shows a case in which triggering of watchpoint A3 is delayed compared to A2, and therefore, a5 gets activated prior to A3 triggering, resulting in the activation firing sequence shown. In sequence 2, A3 only triggers after the goal node a6 is triggered. In sequence 3, A3 triggers before A2, and results in the sequence shown. These sequences do not yet account for the relative delays in firing of the guard conditions in effect nodes. Since

Table 1: Triggering and Activation Sequences for Fig. 2.

Item	Triggering Sequence	Activation Firing Sequence
1	[A1, A2, A5, A4, A3, A6]	[{a1}, {a2}, {a5}, {a4}, {a3}, {a6}]
2	[A1, A4, A2, A5, A6, A3]	[{a1}, {a4}, {a2}, {a5}, {a6}, {a3}]
3	[A1, A3, A2, A4, A5, A6]	[{a1}, {a3}, {a2}, {a4}, {a5}, {a6}]

firing of the guard node as a result of the triggering of the guard condition is a prerequisite for applying the effect in an effect node, the number of unique effect traces that can occur is higher than 22. To reduce the effects traces, the watchpoints in activation nodes could include additional conditions that verify the application of previously enabled effects.

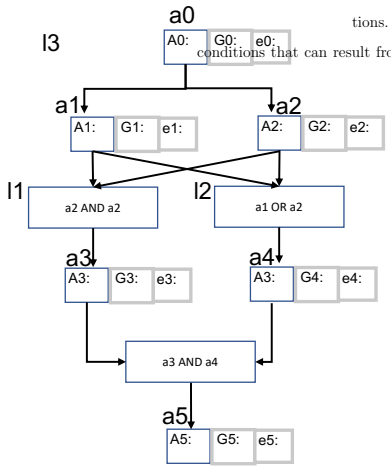

Figure 5: Attack Graph with 1, Race Conditions

One application of the simulator is to flag race conditions in the firing sequences. Having race conditions in the firing sequence is a side-effect of nondeterminism in attack graphs, and therefore it would be useful for the attacker to be aware of their existence. Race conditions in activation firing and effects execution can occur due to the relative delays in firing activation or guard conditions. Fig. 5 contains some examples of race conditions that can result from relative de-lays or automation policy.

We will use the notation *activation node fired (causing watchpoint triggered)* to explain sequences in Table 2. In sequence in the watchpoint triggering sequence shown in Table 2 causes the following activation firing sequence: *a0(A0)* $\rightarrow$ *a1(A1)* $\rightarrow$ *l2* $\rightarrow$ *a4(A3)* $\rightarrow$ *a2(A2)* $\rightarrow$ *l1* $\rightarrow$ *a3(A3)* $\rightarrow$ *a5(A5)*, because both *a3* and *a4* registered the same watchpoint *A3*, though watchpoint *A3* is registered for *a3*, only after *a4* is fired. In comparison, sequence 2 is: *a0(A0)* $\rightarrow$ *a1(A1)* $\rightarrow$ *a2(A2)* $\rightarrow$ *l1* $\rightarrow$ {*a3(A3)*, *a4(A3)* } $\rightarrow$ *a5(A5)*, because the same watchpoint A3 is registered by both *a3* and *a4*. Therefore, when *A3* is triggered both *a3* and *a4* fire. Either *a3* or *a4* could have fired first. Sequence 5 and sequence 4 have identical watchpoint triggering sequences but different activation firing

Table 2: Triggering and Activation Sequences for Fig. 5.

Item	Triggering Sequence	Activation Firing Sequence
1	[A0, A1, A3, A2, A3, A5]	[{a0}, {a1}, {a4}, {a2}, {a3}, {a5}]
2	[A0, A1, A2, A3, A5]	[{a0}, {a1}, {a2}, {a3, a4}, {a5}]
3	[A0, A2, A1, A3, A5]	[{a0}, {a2}, {a1}, {a3, a4}, {a5}]
4	[A0, A1, A2, A3, A3, A5]	[{a0}, {a1}, {a2}, {a4}, {a3}, {a5}]
5	[A0, A1, A2, A3, A3, A5]	[{a0}, {a1}, {a2}, {a3}, {a4}, {a5}]

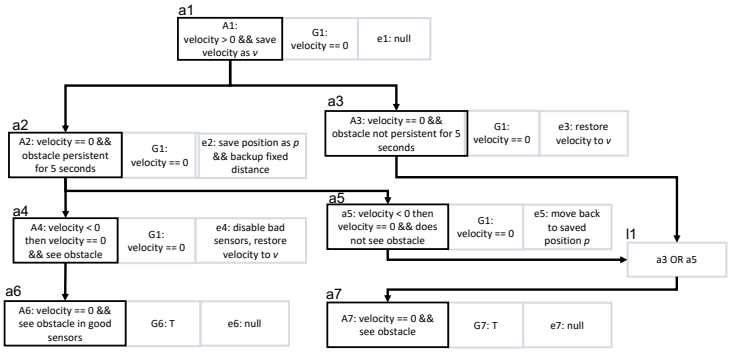

Figure 6: Roomba Attack-Defense Tree

sequences due to the automation implementation picking either *l1* or *l2* first to fire. The sequence 4 occurs because only after *a2(A2)* $\rightarrow$ *l2* $\rightarrow$ *a4(A3)*, does *a3* register *A3*. In sequence 5, *a2(A2)* $\rightarrow$ *l1* $\rightarrow$ *a4(A3)* occurs before *a4* registers *A3*. Fig. 6 is a simplified version of an attack-defense tree [17] that shows how a consumer robot, such as Roomba iCreate2 [12], can be enhanced to defend against sensor attacks. In activation node a1, the velocity of the Roomba is saved periodically until in a2 a stop is detected persistently with an obstacle for 5 seconds. In e2, its position is saved, and the Roomba moves backward for a fixed distance. If there is still an obstacle, then in e4, it assumes the sensors are bad, and disables the bad sensors. If there is no obstacle (in a5), the robot moves back in e5 to the previously saved position in e2. The node a6 is the case when the Roomba detects an obstacle and stops without using bad sensors, and a7 is the case when it stops with data from all sensors. Another parallel path starts with a3, where the obstacle is transient, and e3 restores the velocity saved earlier in a1. Note that this graph has two activation nodes, *a6* and *a7*, in its goal node set.

5 Related Work

The notion of an *attack graph* was proposed in 1998 by Phillips and Swiler [21]. Sheyner et al. [24] used attack graphs to model exploitation of vulnerabilities in a system. They define an attack graph as a data structure used to represent all possible attacks on a network that can take a system from a set of initial states to a set of goal states. Model checking was proposed as a way to see whether an attacker can succeed in an attack. This representation of attack graph was used to do vulnerability analysis by Ingols et al. [11]. Sheyner and Wing discussed tools for automatic generation of attack graphs [25]. Jajodia and Noel [14] describe generation of multi-stage attack graphs.

Ou et al. described MulVAL, a scalable tool that is used to conduct network security analysis across multi-hosts based on multi-stage attacks [20]. The uncertainty of attack success and the environment led to the use of partially observable Markov decision process for attack modeling by Surraute et al. [23]. Kordy et al. defined an attack-defense tree to represent the interactions between the attack and defense [17]. A relatively recent taxonomy of attack graph generation techniques is reported by Kaynar et al. [15]. While the majority of attack graph research focused on enterprise network security, recently, attack graphs have also been used to describe network protocol security [26], and cyber-physical system attacks [10]. Recently approaches make sure attack graphs are machine readable, such as [18]. Yet, these approaches fall short of providing the execution semantics of the machine readable attack graph as in our approach.

The idea of simulated penetration testing, to completely automate a penetration test, was proposed by Hoffmann [9]. The idea of threat emulation to demonstrate and validate the effectiveness of attacks has been around for a century [7]. Cyber-ranges [4] used for Testing and Evaluation of systems can considerably benefit from automated threat emulation of attack scenarios conducted by Applebaum et al. [1]. When a multi-stage attack scenario is enacted on a system, the uncertainty of the system and environment requires frequent *reevaluation* of the system state prior to application of the subsequent stage. For this reason, our attack scenario enactment requires reevaluation of the current target system state prior to applying the attack effects or triggering the next stage.

Since an attack graph may be enacted over a period of time of the attacker's choosing, automated threat emulation of an attack graph may also

run into minutes, hours, or more. Since implementation of watchpoints that produce alerts is essential for executing attack graphs, what techniques can be used to generate alerts becomes a crucial question. One important assumption the watchpoint based alerting makes is that the target system does not change for the duration between the production of the logs that caused the alerts, and the use of the alerts in the AGE graph for making a firing decision. Therefore, the threat emulation process must decide based on the target system attributes and threat emulation goals. The decision on whether to use a batch analytics model implemented in cyber log analysis tools such as Splunk [2], and ELK [6], or use streaming analytics systems such as those described by Haruna et al. [13] depends on the latency needs of threat emulation. The streaming analytics systems have the advantage of lower latency and better accuracy of evaluation when sliding windows are used, compared to batch analytic processing systems.

Gelernter and Carriero clarified in 1992 that a programming language specifies computation, while a coordination language acts as a glue that binds separate activities in an ensemble [8]. A rich set of coordination languages and models have emerged since then. The PTIDES model [31] related model time to real time and used the natural partial order of tasks to achieve deterministic concurrency in real-time applications. The recording of time stamps based on physical clocks has been used to ensure consistency of real-time behavior of Google Spanner, a very large, distributed database [3]. The time-triggered programming language Giotto enforced the concept of logical execution time (LET) as the worst-case execution time estimation for a task excluding input and output times [16]. Recent languages such as Lingua Franca built on the PTIDES model and Reactor model include multiple timelines, again, to achieve deterministic concurrency [19]. While the ideas of partial order of the attack graph nodes is highly relevant, and LET can be a very useful, yet optional, technique in *time-boxing* attacker tactics, the execution of an attack-graph need not exhibit deterministic concurrency. Our coordination model for multi-stage attack modeling does not assume the need for deterministic execution of the attacker tactics, especially in the face of a dynamic environment where defenders and users could alter the system state without the knowledge of the attacker. Our approach is closer to the PTIDES approach [31] in that we do make use of the natural dependencies of the actors to let the attack proceed, if necessary, through multiple concurrent timelines.

6 Conclusions

This paper documents the first use of a coordination model for multi-stage attacks with the AGE graph and its precise semantics. Further, we showed examples of AGE graphs, for attack and defense, created by ourselves or published previously. We ran the AGE graphs through the simulator to highlight some of the nondeterministic properties. An AGE graph may reach one of the goal states of the attack graph only if the target system will permit it. Therefore, it is impossible to claim deterministic concurrency [31, 3, 19] for an AGE graph, though we believe it is possible to prove reachability from a starting node set to an element of the goal set, either using the simulator or through reachability analysis of the AGE graph based on the firing states of the activation nodes or guarded effect nodes. This reachability analysis remains a future research topic, so does automation of an AGE graph. Another area of future research is the extension of AGE graphs with cycles. Detecting the end of the threat emulation is tricky using AGE graphs. Once an AGE graph node in a goal node is fired, there are multiple policy choices for detecting attack termination. Researching these choices remains a future research topic.

References

[1] Andy Applebaum, Doug Miller, Blake Strom, Henry Foster, and Cody Thomas. Analysis of automated adversary emulation techniques. In *Proceedings of the Summer Simulation Multi-Conference*, pages 1–12, 2017.

[2] Ledion Bitincka, Archana Ganapathi, Stephen Sorkin, Steve Zhang, et al. Optimizing data analysis with a semi-structured time series database. *SLAML*, 10:7–7, 2010.

[3] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):1–22, 2013.

[4] Suresh K Damodaran and Jerry M Couretas. Cyber modeling & simulation for cyber-range events. In *Proceedings of the Conference on Summer Computer Simulation*, pages 1–8, 2015.

[5] Suresh K Damodaran and Joshua D Guttman. Systems and methods for declarative specification, detection, and evaluation of happened-before relationships, December 31 2019. US Patent 10,521,331.

[6] Elastic.co. Elasticsearch: The official distributed search & analytics engine. Accessed Jan. 29, 2022.

[7] Gregory Fontenot and Darrell L Combs. Fighting blue: Why first class threat emulation is critical to joint experimentation and combat development. *American Intelligence Journal*, 26(1):24–30, 2008.

[8] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):96, 1992.

[9] Jörg Hoffmann. Simulated penetration testing: From "Dijkstra" to "Turing test++". In *Twenty-Fifth International Conference on Automated Planning and Scheduling*, 2015.

[10] Mariam Ibrahim, Qays Al-Hindawi, Ruba Elhafiz, Ahmad Alsheikh, and Omar Alquq. Attack graph implementation and visualization for cyber physical systems. *Processes*, 8(1):12, 2020.

[11] Kyle Ingols, Richard Lippmann, and Keith Piwowarski. Practical attack graph generation for network defense. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 121–130. IEEE, 2006.

[12] iRobot. irobot create® 2 programmable robot. Accessed Jan. 30, 2022.

[13] Haruna Isah, Tariq Abughofa, Sazia Mahfuz, Dharmitha Ajerla, Farhana Zulkernine, and Shahzad Khan. A survey of distributed data stream processing frameworks. *IEEE Access*, 7:154300–154316, 2019.

[14] Sushil Jajodia and Steven Noel. Advanced cyber attack modeling analysis and visualization. Technical report, GEORGE MASON UNIV FAIRFAX VA, 2010.

[15] Kerem Kaynar. A taxonomy for attack graph generation and usage in network security. *Journal of Information Security and Applications*, 29:27–56, 2016.

[16] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

[17] Barbara Kordy, Sjouke Mauw, Saša Radomirović, and Patrick Schweitzer. Attack–defense trees. *Journal of Logic and Computation*, 24(1):55–87, 2014.

[18] Jooyoung Lee, Daesung Moon, Ikkyun Kim, and Youngseok Lee. A semantic approach to improving machine readability of a large-scale attack graph. *The Journal of Supercomputing*, 75(6):3028–3045, 2019.

[19] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A Lee. A language for deterministic coordination across multiple timelines. In *2020 Forum for Specification and Design Languages (FDL)*, pages 1–8. IEEE, 2020.

[20] Xinming Ou, Sudhakar Govindavajhala, Andrew W Appel, et al. Mulval: A logic-based network security analyzer. In *USENIX security symposium*, volume 8, pages 113–128. Baltimore, MD, 2005.

[21] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *Proceedings of the 1998 workshop on New security paradigms*, pages 71–79, 1998.

[22] CVE Program. Common vulnerabilities and exposures. Accessed Jan. 29, 2022.

[23] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. Pomdps make better hackers: Accounting for uncertainty in penetration testing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[24] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M Wing. Automated generation and analysis of attack graphs. In *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 273–284. IEEE, 2002.

[25] Oleg Sheyner and Jeannette Wing. Tools for generating and analyzing attack graphs. In *International symposium on formal methods for components and objects*, pages 344–371. Springer, 2003.

[26] Orly Stan, Ron Bitton, Michal Ezrets, Moran Dadon, Masaki Inokuchi, Ohta Yoshinobu, Yagyu Tomohiko, Yuval Elovici, and Asaf Shabtai. Extending attack graphs to represent cyber-attacks in communication protocols and modern it networks. *IEEE Transactions on Dependable and Secure Computing*, 2020.

[27] Blake E Strom, Andy Applebaum, Doug P Miller, Kathryn C Nickels, Adam G Pennington, and Cody B Thomas. Mitre att&ck: Design and philosophy. *Mitre Product Mp*, pages 18–0944, 2018.

[28] Travis L Swiatocha. Attack graphs for modeling and simulating sophisticated cyber attack. Technical report, Naval Postgraduate School, 2018.

[29] Tenable. Nessus. Accessed Jan. 30, 2022.

[30] Jianping Zeng, Shuang Wu, Yanyu Chen, Rui Zeng, and Chengrong Wu. Survey of attack graph analysis methods from the perspective of data and knowledge processing. *Security and Communication Networks*, 2019, 2019.

[31] Yang Zhao, Jie Liu, and Edward A Lee. A programming model for time-synchronized distributed real-time systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 259–268. IEEE, 2007.