Managing Attack Graph Complexity Through Visual Hierarchical Aggregation

Steven Noel
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
snoel@gmu.edu

Sushil Jajodia
Center for Secure Information Systems
George Mason University
Fairfax, VA 22030
jajodia@gmu.edu

ABSTRACT
We describe a framework for managing network attack graph complexity through interactive visualization, which includes hierarchical aggregation of graph elements. Aggregation collapses non-overlapping subgraphs of the attack graph to single graph vertices, providing compression of attack graph complexity. Our aggregation is recursive (nested), according to a predefined aggregation hierarchy. This hierarchy establishes rules at each level of aggregation, with the rules being based on either common attribute values of attack graph elements or attack graph connectedness. The higher levels of the aggregation hierarchy correspond to higher levels of abstraction, providing progressively summarized visual overviews of the attack graph. We describe rich visual representations that capture relationships among our semantically-relevant attack graph abstractions, and our views support mixtures of elements at all levels of the aggregation hierarchy. While it would be possible to allow arbitrary nested aggregation of graph elements, it is better to constrain aggregation according to the semantics of the network attack problem, i.e., according to our aggregation hierarchy. The aggregation hierarchy also makes efficient automatic aggregation possible. We introduce the novel abstraction of protection domain as a level of the aggregation hierarchy, which corresponds to a fully-connected subgraph (clique) of the attack graph. We avoid expensive detection of attack graph cliques through knowledge of the network configuration, i.e. protection domains are predefined. While significant work has been done in automatically generating attack graphs, this is the first treatment of the management of attack graph complexity for interactive visualization. Overall, computation in our framework has worst-case quadratic complexity, but in practice complexity is greatly reduced because users generally interact with (often negligible) subsets of the attack graph. We apply our framework to a real network, using a software system we have developed for generating and visualizing network attack graphs.

Categories and Subject Descriptors
K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *unauthorized access.*

General Terms
Security, Human Factors, Management.

Keywords
Network attack modeling, network attack graphs, vulnerability analysis, clustered graphs.

1. INTRODUCTION
The utility of constructing graphs representing actual or possible attacks through a network is well established. Attack graphs represent interactions among individual network vulnerabilities (via attacker exploits), leading to an understanding of collective impact on network security. Attack graphs have application in a number of areas of network security, including vulnerability analysis, intrusion alarm correlation, and attack response. They can be applied in both an offensive (e.g., penetration testing) or defensive (e.g., network hardening) context.

Significant progress has been made in generating attack graphs automatically [1][2][3][4][5][6][7][8][9][10][11]. Previous work to date has focused on attack graph representations and generation algorithms, the network attack modeling necessary for generating attack graphs, and computational scalability. But perhaps the greatest challenge in making attack graph analysis practical for real networks is to manage complexity in user interaction. Visualization is a natural choice for conveying attack graph relationships to the user. Indeed, previous work in attack graph generation typically employs graph visualization to report results. But for realistic networks, the complexity of generated graphs often greatly exceeds the human ability to understand the relationships and take appropriate action. As an example of attack graph complexity, consider Figure 1, which is for a subnet of only 14 machines, with less than 10 vulnerabilities per machine. But the management of attack graph complexity from a usability standpoint has received little attention.

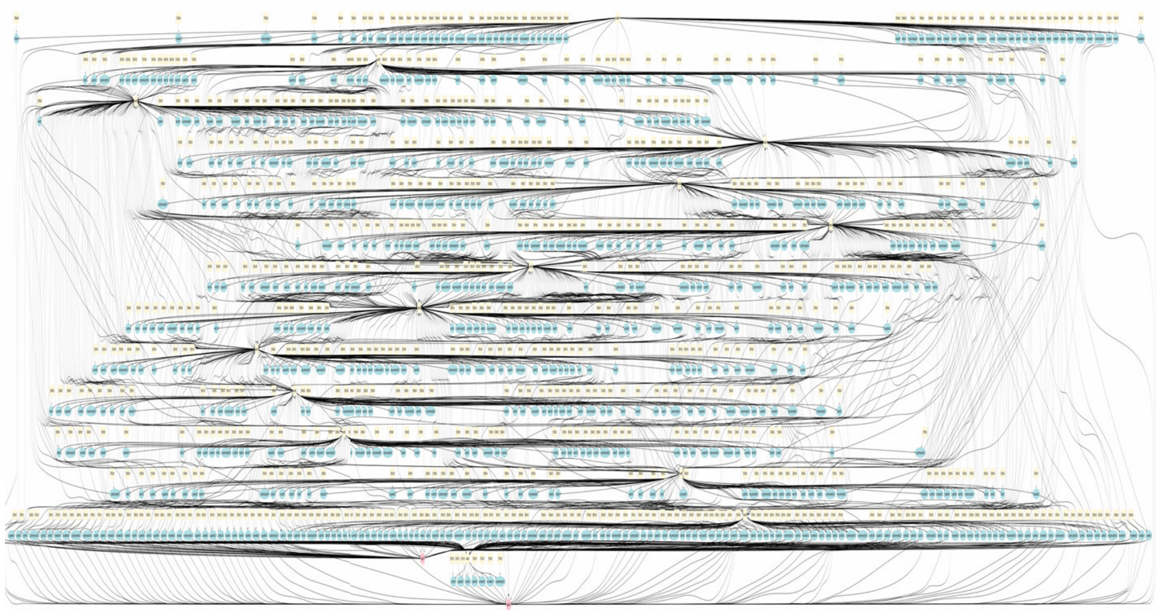

Figure 1. Attack graph for 14-machine subnet.

In this paper, we manage attack graph usability complexity through interactive visualization and hierarchical graph aggregation. We describe a visualization framework in which attack graph elements are aggregated hierarchically, i.e., non-overlapping nested attack subgraphs are recursively collapsed. This aggregation allows attack graph complexity to be compressed to an arbitrary degree, with higher levels of aggregation corresponding to higher levels of abstraction. The user can interactively aggregate or de-aggregate an attack graph according to the task at hand, in either a top-down or bottom-up fashion.

We define a hierarchy of rules, which controls how attack graph elements or subgraphs can be aggregated at each level of the hierarchy. Rules are based on either common attribute values for attack graph elements or attack subgraph connectedness. All our rules can be implemented efficiently, and support automatic aggregation at each level of the hierarchy.

We introduce a novel attack graph abstraction, i.e., the protection domain, which is particularly effective at reducing attack graph complexity while preserving semantics. The protection domain represents a fully connected subgraph (clique) among machines and exploits, for example, as might be found for a set of machines with no traffic filtering among them. Our other aggregation rules reduce complexity in a linear fashion, i.e., they generally map sets of largely independent (sparsely connected) objects to a single vertex. But the protection domain rule reduces complexity in a quadratic fashion, i.e., it maps fully connected sets of objects (machines and exploits among them) to a single vertex. We assume protection domains are known from the network configuration, so that we avoid the expensive detection of attack graph cliques.

Overall, the worst-case computational complexity in our framework is quadratic. However, in practice the complexity is reduced because users generally interact with subgraphs of the attack graph at any given time, and at lower levels of aggregation subgraphs of negligible size are being processed.

In the next section, we review related work in this area. Section 3 then describes the attack graph representation used in our framework, which is designed to be scalable and readily interpretable. Section 4 describes the aggregation rule hierarchy in our framework. In Section 5, we demonstrate novel visual representations for hierarchically-aggregated attack graphs, using a real network and a software system we have developed for generating and visualizing network attack graphs. In Section 6, we summarize our work and draw conclusions.

2. RELATED WORK

The general approach to generating attack graphs automatically is to first build a network attack model comprised of security conditions and rules (exploits) for changing the attack state based on the security conditions. From this, analysis is performed to generate sequences of exploits that lead to an unsafe network state, which can be organized in a graph. Various methods have been proposed for this kind of network attack analysis, including logic-based (symbolic model checker) approaches [1][2][3][4], and explicit graph-based approaches [5][6][7][8][9]. While these approaches allow large complex attack graphs to be generated, such as in Figure 1, almost no attention has been paid to the management of attack graph complexity from the standpoint of human understandability. We go beyond mere generation of attack graphs, turning our focus to their usability and understandability, managing their complexity through interactive visualization and hierarchical graph aggregation.

Early approaches to attack graph generation generally suffered serious scalability problems, because they considered the full exponential state space. More recently [10][11], it has been recognized that in practice, it is reasonable to assume the attacker's control over the network increases monotonically, i.e., the attacker need not relinquish resources already gained in order to further advance the attack. That is, at a reasonable level of modeling fidelity, attacker behavior resolves to monotonic. This monotonicity allows (all possible) network attacks to be represented as a graph of dependencies among exploits and security conditions, rather than as an enumeration of states. The exploit dependency representation has worst-case quadratic (as opposed to exponential) complexity in the number of exploits. We employ exploit dependencies as our basic representation, but we define a hierarchy of rules for aggregating the basic graphs according to network attack semantics, and introduce novel interactive visual representations for aggregated graphs.

There has also been complementary work in the kind of network attack modeling needed for generating attack graphs [12][13][14]. Also, attack graphs have been generated for intrusion alert correlation [15][16][17][18]. However, this work does not address attack graph usability and complexity management as we do.

While various approaches have been proposed for inducing hierarchical structure on graphs, our framework is a form of so-called clustered graphs, first described in [19]. Actually, we apply a generalization of the original clustered graphs [20], in which it is possible to have views across multiple levels of the cluster hierarchy, as opposed to views at a single level only. Software architecture for clustered graphs, with arbitrary numbers of hierarchies per graph and arbitrary numbers of views per hierarchy, is described in [21]. Recent work in clustered graphs is represented by [22].

3. ATTACK GRAPH REPRESENTATION

In terms of feasibility for realistic networks, perhaps the most critical property for attack graph representations is scalability. In early formulations, e.g. [1][2][3][4][5][6][7][8], attack graphs represented transitions of a state machine, where states are network security attributes and state transitions are attacker exploits, resulting in graphs that enumerate transition paths through state space. These state-based graphs have the property that one can simply follow a path through the graph to generate an attack path (sequence of exploits leading from the initial state to the goal state). But such graphs have serious scalability problems, as they can grow exponentially with the number of state variables.

More recently, it has been recognized that under the assumption of monotonically increasing state variables, it is not necessary to explicitly enumerate attack states [10]. Rather, it is sufficient to form a graph of dependencies among exploits and security conditions. Such dependency graphs grow only quadratically with the number of (machine-dependent) exploits, so that it becomes feasible to apply them for realistic networks. The assumption of monotonicity is quite reasonable, corresponding to the conservative assumption that once an attacker gains control of a network resource, he need not relinquish it to further advance the attack. In reality, when non-monotonic attack behavior does occur, it is usually at a very detailed level (more detailed than is warranted for network penetration analysis) and it resolves to monotonic behavior at a suitable level of detail.

Figure 2 is a comparison of state-enumeration and exploit-dependency attack graph representations. The exploit dependency graph is taken from [10] and the state enumeration graph is taken from [3]; both graphs are for an identical network attack scenario. In this example, the exploit dependency graph is clearly more efficient. In the exploit dependency graph, each exploit or dependency appears only once, and no edges appear between independent exploits. For example, in the exploit dependency graph, each of the three exploits *ftp_rhosts(0,1)*, *sshd_bof(0,1)*, and *ftp_rhosts(0,2)* appears only once, and since these exploits are independent, there are no edges between them. In contrast, in the state transition graph, there are edges (state transitions) between these 3 exploits even though the order of these exploits (transitions) is irrelevant, and these 3 exploits each occur multiple times. Also, the attack path *sshd_bof(0,1)* $\rightarrow$ *ftp_rhosts(1,2)* $\rightarrow$ *rsh(1,2)* $\rightarrow$ *local_bof(1,2)* appears twice in the state-enumeration graph and only once in the exploit-dependency dependency graph, as marked with bold arrows in the figure.

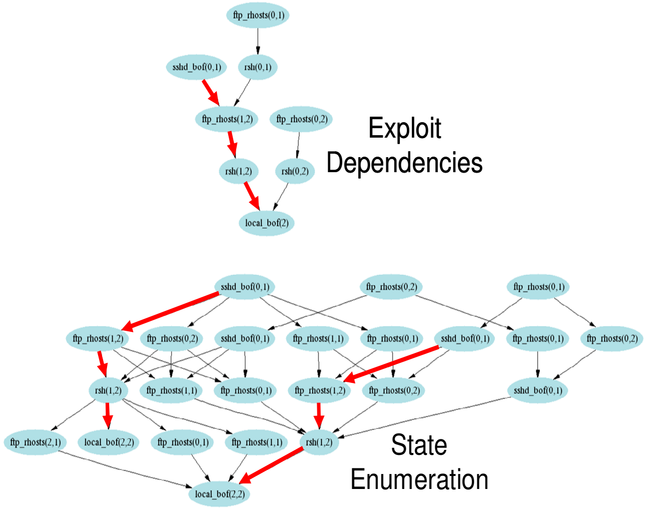

Figure 2. Exploit dependency and state enumeration attack graph representations.

The compact exploit-dependency graph representation contains all the information needed for network attack analysis. For example, all paths (through attack state space) can be generated if needed. This compact representation supports efficient forms of post analysis such as the computation of minimal cost network hardening measures described in [23]. But for us, what is important (other than computational scalability) is that the attack graph can be readily interpreted by humans when it is visualized.

In the original formulation [10] for monotonic network attack logic, graph dependencies are oriented in terms of security conditions rather than exploits. That is, graph vertices represent conditions, which are connected by edges that represent exploits. The top of Figure 3 shows an example of this representation. Here, the attack graph consists of exploits among three machines, with the attack goal of executing commands as superuser on Machine 3. The attack graph is actually very simple in this example, consisting of these two paths: (1) *wuFtpd*(1,2) $\rightarrow$ *wuFtpd*(2,3) and (2) *xdmLoginPrompt*(1,2) $\rightarrow$ *xdmLogin*(1,2) $\rightarrow$ *wuFtpd*(2,3). However, the essential flow of these 2 attack paths is not visually obvious in this representation. The problem is that individual exploits are distributed throughout the graph, rather than being centralized. This makes it difficult to visually follow the distinct steps in the attack. An alternative attack dependency representation has been proposed [11] that is the dual of the one in the top of Figure 3, with security conditions being edges and exploits being vertices. That is, exploits are connected to one another via precondition/postcondition dependency edges, making the graph exploit oriented. In the exploit-oriented representation, the sequences of exploits comprising the attack are visually obvious, since each exploit is centralized in the graph.

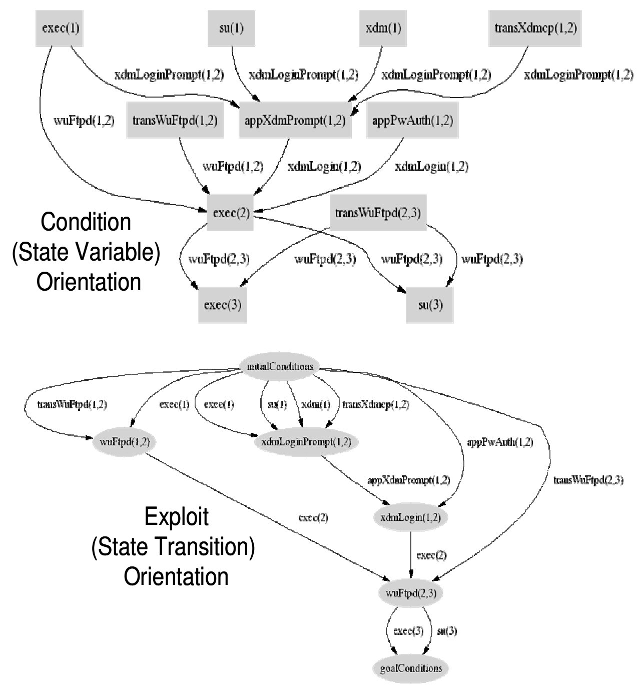

Figure 3. Condition-orientated and exploit-orientated dependency graphs.

The bottom of Figure 3 is the exploit dependency version of the condition dependency graph in the top of Figure 3. For consistency, the exploit dependency representation includes a distinguished "initial conditions" exploit having null preconditions and postconditions equivalent to the initial conditions in the network model. Similarly, goal conditions are handled by a distinguished "goal conditions" exploit having null postconditions and preconditions equivalent to the specified attack goal conditions. The exploit-oriented dependency graph is the basic representation used in our framework. Technically, it is a multigraph, with parallel edges for the preconditions or postconditions of a single exploit.

For interactive visualization, we modify somewhat the basic exploit-oriented representation shown in the bottom of Figure 3. One modification is to remove the initial condition and goal condition exploit vertices, and replace them with individual condition vertices. This helps keep these conditions visually localized with their corresponding exploits, and helps reduce edge-crossing clutter on larger graphs. Another modification is to introduce an intermediate common vertex for duplicate preconditions for an exploit. These duplicate preconditions come from exploits with identical postconditions, forming a logical disjunction (Boolean OR), and we want to make this visually obvious. In this case, we have something of a hybrid between the condition-oriented and exploit-oriented representations, since both conditions and exploits appear as attack graph vertices.

4. ATTACK GRAPH AGGREGATION

In the previous section, we describe how the exploit-dependency representation, in comparison to the state-transition representation, reduces attack graph complexity from exponential to quadratic, which makes attack graph generation computationally feasible for realistic networks. But exploit dependency graphs can still be too complex for humans to easily understand. As a simple example, an attack graph with 100 exploits could have up to $100^2 = 10,000$ edges in the exploit dependency graph. It is clear that a human would be unable to assimilate such a large graph all at once.

For visualizing large attack graphs, a key strategy for managing complexity is to reduce the apparent size of the information space, consistent with the semantics of the problem. We reduce complexity through hierarchical (recursive) graph aggregation, with higher levels of aggregation corresponding to higher levels of abstraction and lower complexity.

While in principle it would be possible to allow attack graph aggregation in a totally arbitrary fashion, it is generally better to take advantage of domain knowledge and constrain aggregation according to the semantics of the network attack problem. Thus we define a hierarchy of rules that determine the conditions under which certain attack graph elements can be aggregated. This rule hierarchy is not overly restrictive, i.e., violations of these rules would yield aggregations of semantically incompatible elements.

Figure 4 shows our aggregation hierarchy, with level-dependent rules for aggregating elements of an attack graph. The user can interactively aggregate or de-aggregate an attack graph, either top-down bottom-up.

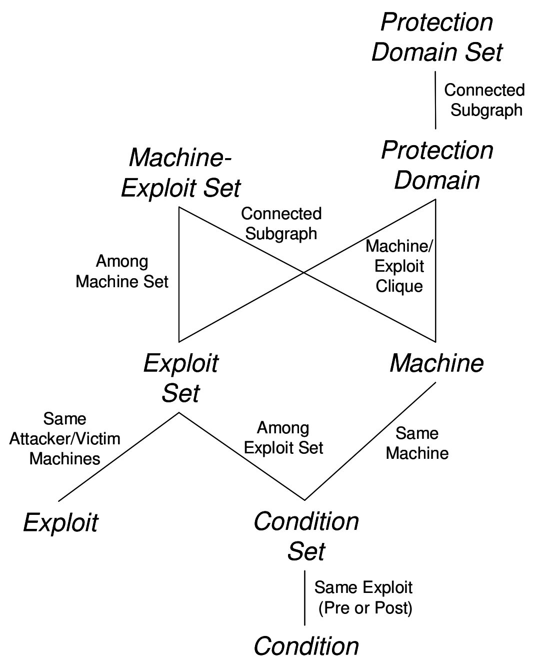

Figure 4. Hierarchy of rules for aggregating attack graph.

At the bottom of the hierarchy in Figure 4, *exploits* and *conditions* are aggregated into *exploits sets* and *condition sets*, respectively. Condition sets are aggregated into either *machine* abstractions or exploit sets. Sets of machines and the exploits among them are aggregated into *machine-exploit sets* or *protection domains*, and protection domains are aggregated into *protection domain sets*. Here, protection domains represent fully connected subgraphs (cliques) among machines and exploits, which we assume are known from the network configuration, so that we avoid expensive clique detection. Aggregation into sets of protection domains requires testing whether the protection domains are connected, which has linear complexity in the size of the graph of aggregated protection domains. For levels of the aggregation hierarchy other than protection domain sets, the rules are based on matching attribute values of attack graph elements (which are known from the network configuration), allowing very efficient automatic aggregation.

Our rule for exploits constrains aggregation to those with the same pair of (possibly identical) attacker/victim machines. That is, only exploits between the same pair of machines are aggregated. This is not overly restrictive, because exploits between different pairs of machines can be aggregated at the machine level. Assuming exploits are sparsely interconnected via conditions (which is usually true in practice), exploit aggregation reduces graph complexity essentially linearly, since the number of condition edges is roughly proportional to the number of exploits between a pair of machines.

Conditions are aggregated into sets if they are either preconditions or postconditions (but not both) of the same exploit. This is not overly restrictive, because in terms of understanding attack paths, there is no reason to mix conditions across exploits, or to mix preconditions and postconditions of an exploit. Also, conditions across exploits can be subsequently included in higher-level machine aggregations. Since sets of preconditions or postconditions for an exploit are independent, condition aggregation reduces attack graph complexity essentially linearly with the number of exploits.

Sets of conditions are aggregated into a machine abstraction if they are all conditions on that machine. Thus, a machine is simply the union of all its conditions. We use the convention that conditions for network connections, which involve both a source (attacker) and destination (victim) machine, belong to the source machine, because we are taking the point of view of the attacker. The rule for aggregating condition sets into machines is not overly restrictive, since for understanding attack paths there is no reason to mix conditions across machines, and such conditions can be subsequently included in higher-level machine set aggregations. Assuming sets of conditions for a machine are largely independent, condition set aggregation to machines reduces attack graph complexity essentially linearly with the number of machines. Condition sets can also be aggregated, along with exploits, to form an exploit set. The rule here is that the aggregated conditions must be from among the exploits being aggregated.

Machines and the exploits among them can be aggregated to a machine-exploit set if they form a connected subgraph, which allows machines to be aggregated across protection domains. However, the clique property cannot be guaranteed for the aggregated set, because it is based on pre-defined protection domains. Also, machines are not aggregated into sets automatically, only manually, since the semantics of an arbitrary set of machines independent of protection domain is unclear without prior knowledge. Restricting machine-exploit sets to connected subgraphs prevents aggregation of machines into sets that have unreachable components within them. This is not overly restrictive, because disconnected attack graph components correspond to attacker unreachability.

The protection domain abstraction represents a set of machines that have unrestricted access to one another's vulnerabilities, so that the attack graph is fully connected among them (if one treats exploit sets between a pair of machines as graph edges). A protection domain is actually a clique, perhaps even a maximal clique, and finding maximal cliques is NP-complete. However, operationally, protection domains can be identified from the network configuration. A common type of protection domain is a set of machines (e.g., a subnet) with no traffic filtering (e.g., firewalls) among them. In this case, protection domains can be obtained from the subnet mask and machine IP addresses. The protection domain abstraction is particularly effective at reducing attack graph complexity, and has clear and useful semantics. While the other abstractions involve aggregation of largely independent (sparsely connected) objects, the protection domain aggregates fully connected sets of objects (machines and exploits among them), reducing complexity quadratically rather than linearly.

At the highest level of abstraction, protection domains can be aggregated into sets. Here, the only restriction is that the protection domains to be aggregated form a connected subgraph. This restriction prevents aggregation of protection domains into sets that have unreachable components within them. This is not overly restrictive, because disconnected attack graph components correspond to attacker unreachability. Like machine sets, protection domain sets are not formed automatically, since the semantics of an arbitrary set of protection domains is unclear without additional prior knowledge.

The aggregation hierarchy guides the interactive visual exploration of an attack graph. Typically, automatic aggregation at all levels might be applied initially. One could then proceed in a top-down fashion, starting at the highest levels of abstraction and progressively moving towards lower levels of details through de-aggregation. The system should also support bottom-up aggregation (either manual or automatic), perhaps even starting at the lowest possible level, i.e., the fully non-aggregated attack graph. To provide full interactive flexibility, the system should allow the user to aggregate or de-aggregate non-uniformly as desired, i.e., to different levels of details for different portions of the attack graph. Also, arbitrary subsets of elements within a particular level could be aggregated, or elements within a level could be aggregated recursively, as long as the aggregation rule for the level is followed. Another desirable property is persistence, in which the particular structure of an aggregated subgraph persists when it is subsequently de-aggregated.

Figure 5 shows two possible modes of interactive de-aggregation. One mode is to de-aggregate by presenting the contents of the de-aggregated vertex completely isolated from the vertex's original context, i.e., to use de-aggregation as a filter. In this case, the vertex serves as a filter criterion, with the remainder of the attack graph being removed. Another possible mode of de-aggregation is to present the contents of the de-aggregated vertex within the full context of the remainder of the graph. While both modes are possible, even within a single interactive system, we focus on in-context de-aggregation, although our framework applies equally to the two modes.

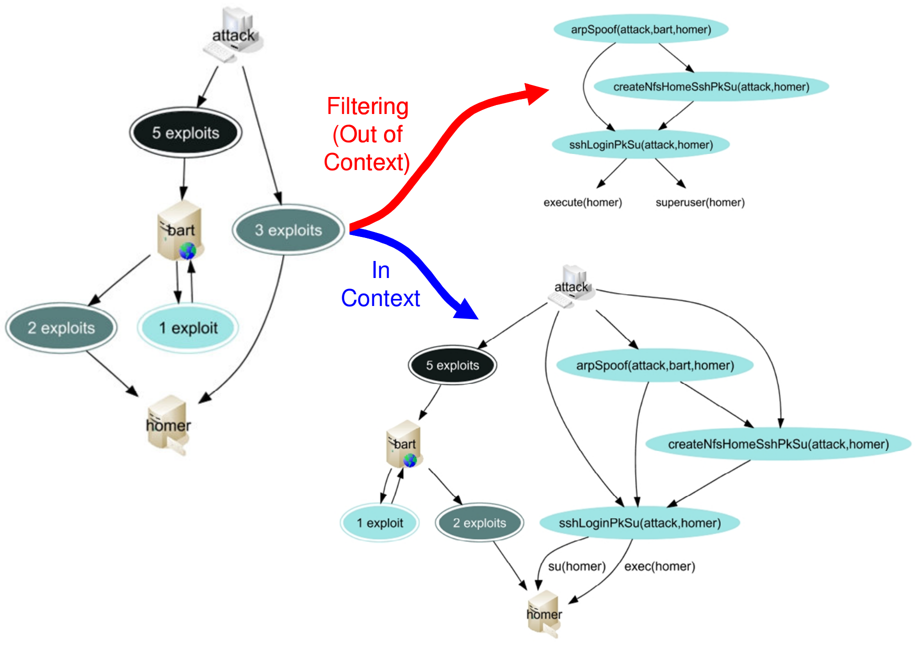
attack
5 exploits
bart
3 exploits
2 exploits
1 exploit
homer
Filtering (Out of Context)
In Context
arpSpoof(attack,bart,homer)
createNfsHomeSshPkSu(attack,homer)
sshLoginPkSu(attack,homer)
execute(homer)
superuser(homer)
attack
5 exploits
arpSpoof(attack,bart,homer)
createNfsHomeSshPkSu(attack,homer)
bart
1 exploit
2 exploits
sshLoginPkSu(attack,homer)
su(homer)
exec(homer)
homer

Figure 5. Filtering versus de-aggregation in context.

5. VISUALIZING AGGREGATED GRAPHS

The previous section describes our hierarchical attack graph aggregation, with rules for aggregating objects at each level of the hierarchy. This allows attack graph abstraction at various levels, from high-level overviews to low-level details. However, the effectiveness of interactive attack graph exploration depends critically on the particular visual (as opposed to mathematical) representations used. This section describes the visual representations in our framework. To do this, we apply a software system we have implemented for generating and interactively visualizing network attack graphs, which was originally described in [11].

Our system builds a model of network vulnerabilities and connectivity automatically, via the open-source Nessus [24] vulnerability scanner. We have modeled about 550 attacker exploits, in terms of preconditions and postconditions on generic attacker/victim machines, from Nessus vulnerabilities we found to be relevant to progressive network penetration. Not all these modeled exploits might appear in an attack graph, only those represented by actual vulnerabilities on the scanned machines in the network. Moreover, such exploits must be reachable by the attacker given the network connectivity, a starting machine (and user privilege level), and a network attack goal. From the network attack model, our system builds exploit sequences leading from the initial network conditions to the attack goal. In doing this, it matches generic (machine independent) exploits with specific machines in the network model. The analysis begins by building an exploit precondition/postcondition dependency graph in the forward direction, adding executable exploits starting from the initial conditions. The system then traverses the resulting dependency graph in the backward direction, retaining relevant exploits starting from the goal conditions.

Figure 6 shows the non-aggregated version of the attack graph for an example network. In the figure, exploits are represented as ovals, initial conditions are represented as boxes, goal conditions are represented as octagons, and all other conditions are represented as plain text. In this network, there are 16 victim machines organized into 4 subnets, with firewalls restricting connectivity between subnets. From the attack graph, one can see some clustering of attacker exploits, which is caused by the unrestricted connectivity among machines in each subnet and restricted connectivity across subnets. But from the non-aggregated view, it is difficult to understand the exact nature of this clustering and to understand the overall flow of the attack.

In Figure 7, the non-aggregated attack graph in Figure 6 has been aggregated (automatically) to the level of machines and sets of exploits between them. That is, conditions for each machine have been aggregated to a single machine vertex (represented with a machine glyph in the figure), and individual exploits between each pair of attacker/victim machines have aggregated to single vertices. In our framework, aggregation can occur at mixed levels of the hierarchy – the single level shown here is for illustrative purposes. In Figure 7, exploit set glyphs include the number of exploits (in each direction) between each machine pair, and are shaded in proportion to the number of exploits (actually, the greater number between the 2 directions), so that larger sets of exploits appear darker. Also, exploit set glyphs have a second periphery to distinguish them from single exploits. Graph edges have arrowheads if the attack flows in a single direction, and the lack of an arrowhead means exploits can be launched in either direction between a pair of machines. In comparison to the non-aggregated attack graph in Figure 6, this graph is much less complex, and the overall attack flow and exploit clustering by subnet are clearer. However, it is tedious to manually determine that the visual clusters really constitute subnet protection domains (fully-connected subgraphs), and to understand the exact nature of interaction across subnets.

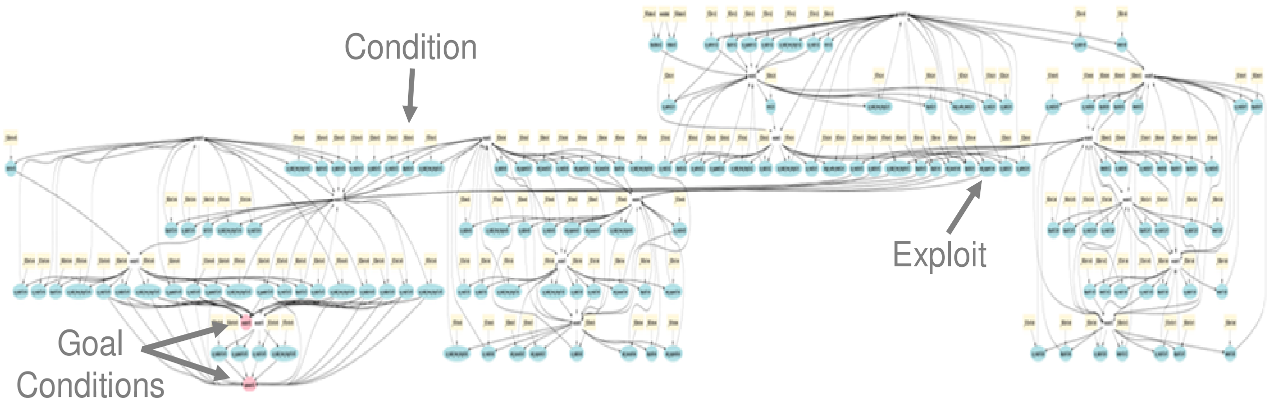

Figure 6. Non-aggregated attack graph.

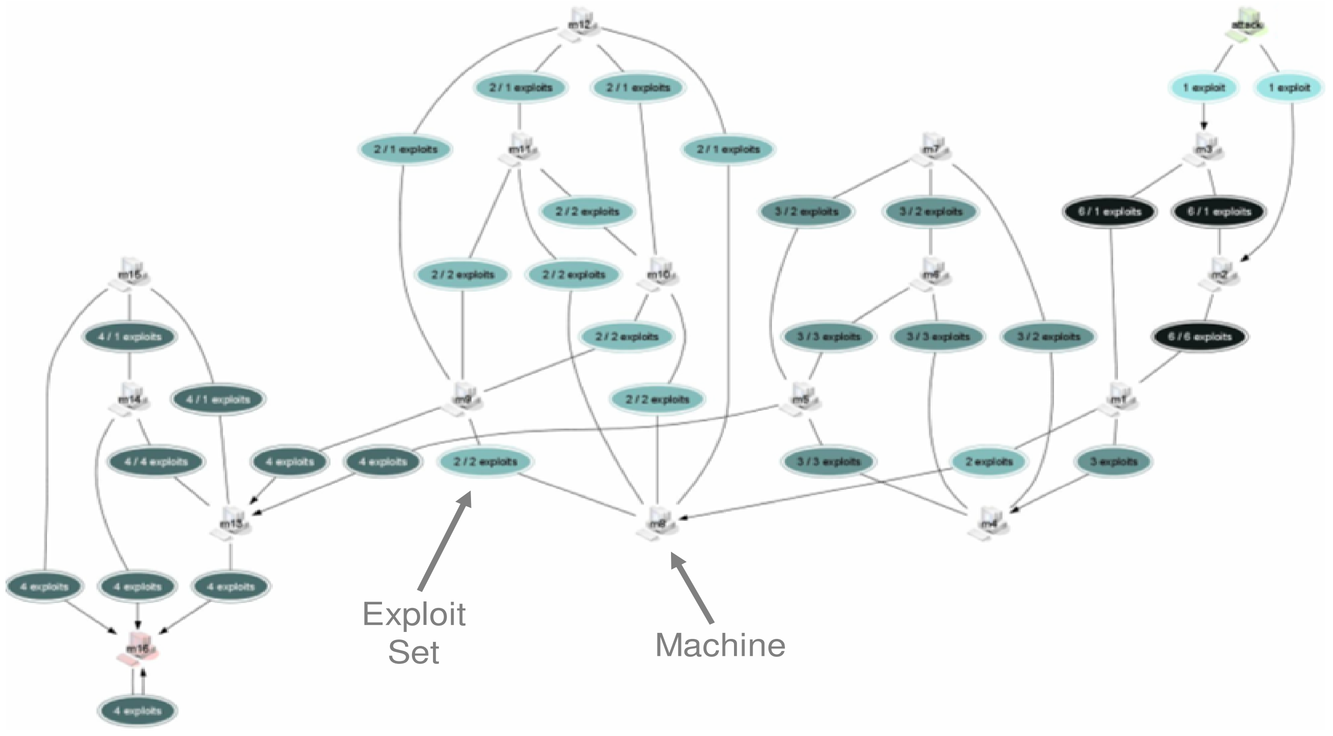

Figure 7. Aggregation to machines and exploit sets.

In Figure 8, the example attack graph has been aggregated to the level of protection domains, which was done automatically from the network model via subnet mask and IP addresses. Protection domains are shown explicitly, and with adjacent machines indicating protection domain membership. The initial attack machine, being in a separate subnet, is shown as a single machine.

In Figure 8, edges between protection domains and machines are without arrowheads, indicating that exploits are launched in both directions among machines in a protection domain. Protection domain glyphs include the number of machines in the protection domain, as well as counts of exploits launched within the subnet. Exploit sets (in plain text) adjacent to machines represent exploits that could be launched against a machine from within the protection domain. Actually, in this visual representation, these exploits are independent of attacker machine, and as such they represent machine vulnerability rather than aggregation of individual exploits. The actual exploits are instead aggregated in the protection domain vertex. Exploit sets across protection domains are adjacent to their respective attacker/victim machines, explicitly showing the interaction across protection domains.

In Figure 8, the overall attack flow is quite clear, in which the attack starts by compromising Subnet1, branches to Subnet2 and Subnet3 (with no interaction between Subnet2 and Subnet3), and ends with compromise of Subnet4. The protection domains (subnets) are represented explicitly, and interaction across subnets is clear. Moreover, in comparison to Figure 6 and Figure 7, attack graph complexity is greatly reduced.

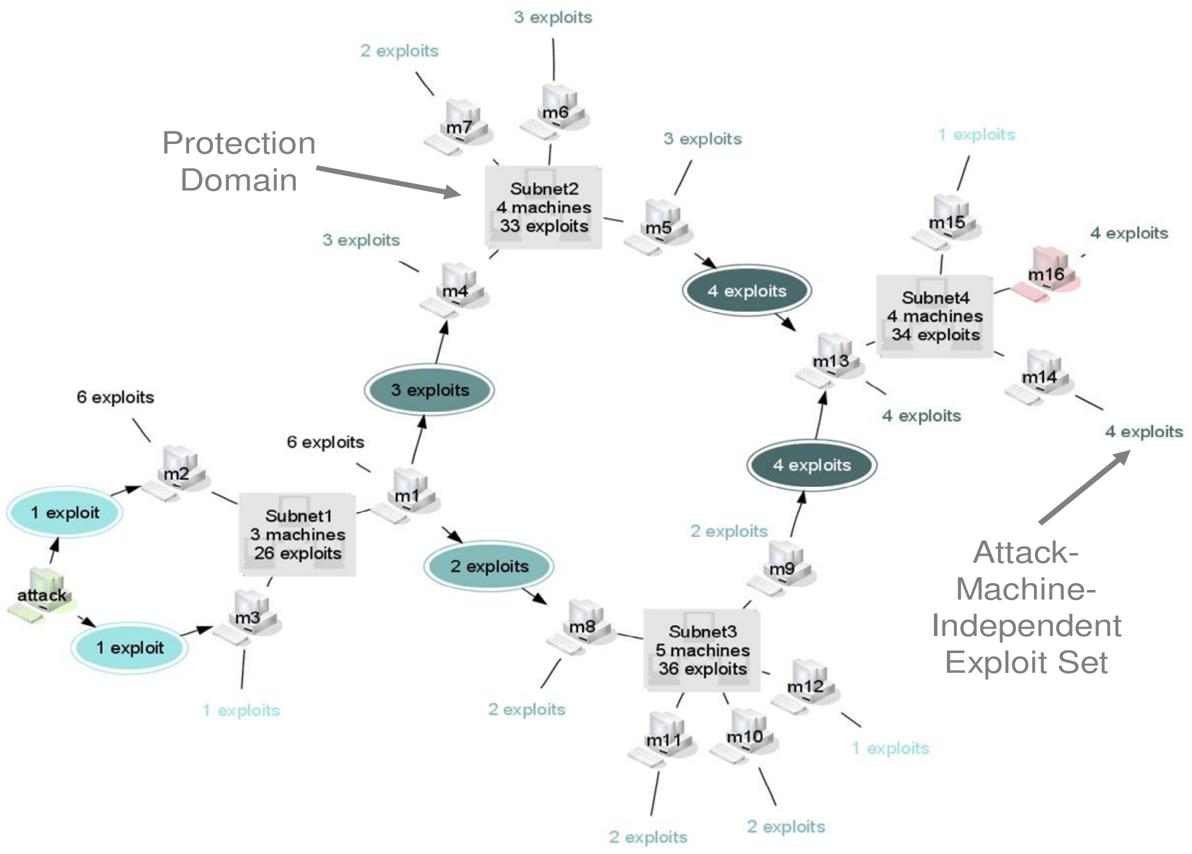

Figure 8. Attack graph aggregated to protection domains.

Figure 9 shows a higher-level visual representation of the example attack graph aggregated to protection domains. In this representation, individual machines in each subnet and their associated attack-machine-independent exploits are not shown. Here, the subnet structure, interaction across subnets, and overall attack flow are perfectly clear. In practice, this is the kind of high-level view that would initially be presented to the user. The attack graph would initially be aggregated (automatically) at each level of the hierarchy, and interactively de-aggregated and the user explores the attack graph top-down. This interactive de-aggregation is localized, e.g., only individual objects would be de-aggregated, as opposed to global de-aggregation in which all objects at a given level are de-aggregated.

In previous views for this example attack graph, objects have been aggregated homogeneously to a particular level of the hierarchy. But our framework is not restricted to such single-level views. In fact, views are possible with arbitrarily mixed levels. That is, objects at various levels of aggregation are consistent with one another. Thus in our framework, one can focus attention and retain context in arbitrary ways.

Figure 10 shows such a mixed-level view for our example attack graph. The view includes objects at all levels of the aggregation hierarchy. The level of detail is low near the start of the attack, and increases toward the attack goal. This might be the kind view one would use to focus on details near the goal and keep the early part of the attack graph as a high-level overview.

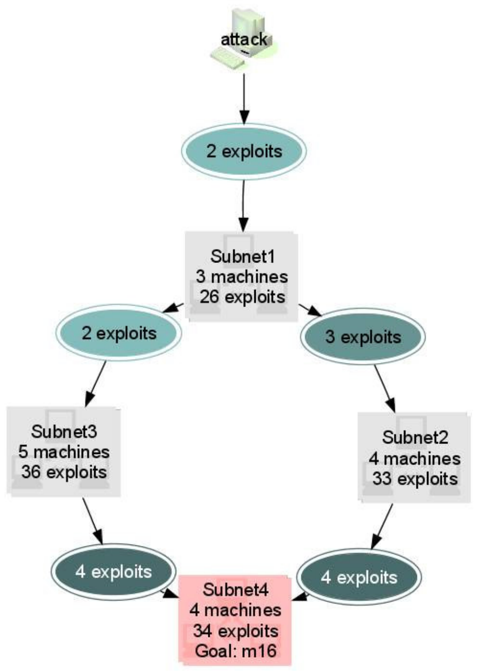

Figure 9. High-level view of protection-domain aggregation.

In Figure 10, Subnet1 and Subnet2 are aggregated to a protection domain set, whose glyph contains a count of the combined machines and exploits among them. For Subnet3, 4 of the 5 machines have been aggregated into a machine set, and the remaining machine is un-aggregated. Also, 3 of the 4 exploits from machine m9 to machine m13 (from Subnet3 to Subnet4, respectively) are aggregated. Subnet4 does not appear explicitly, i.e., there is no aggregation up to the level of protection domain for Subnet4. Rather, 2 of the 4 machines in Subnet4 are in a machine set, and the remaining 2 machines appear separately. Between machines m13 and m16, 3 of the 4 exploits are aggregated to a set. For the remaining exploit, all preconditions and postconditions are un-aggregated, as opposed to the exploit between machines m9 to m13, in which the preconditions and postconditions are aggregated to condition sets.

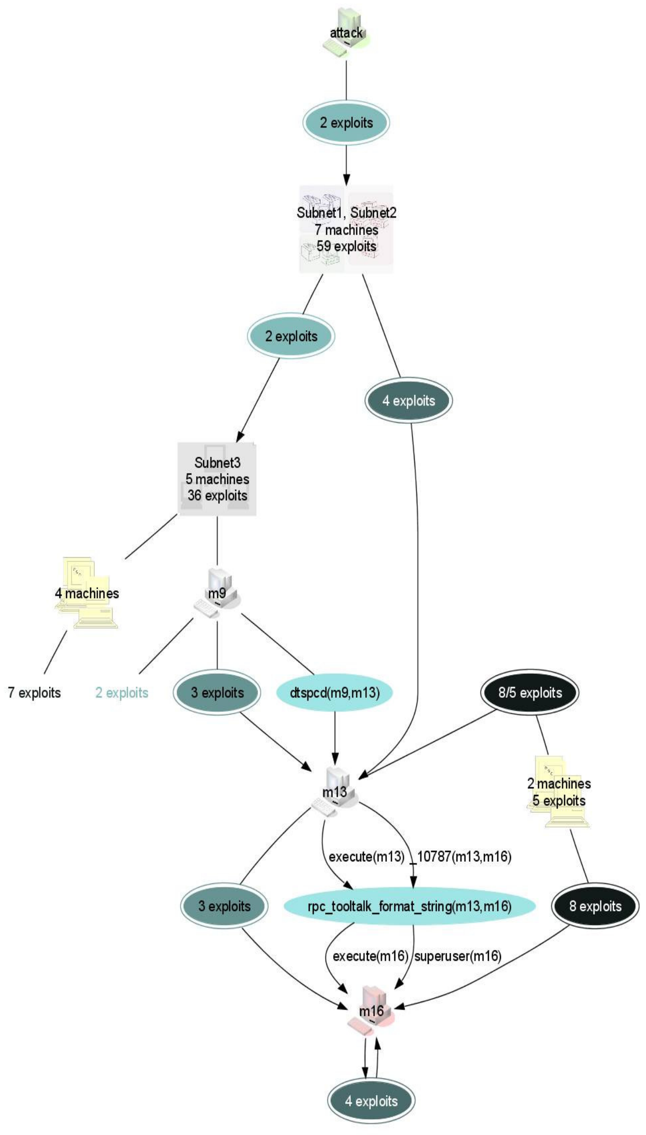

Figure 10. Representation with mixed levels of aggregation.

6. SUMMARY AND CONCLUSIONS

In this paper, we have addressed what is perhaps the greatest challenge in making network attack graphs practical for real networks – managing their visual complexity in user interaction. Interactive visualization provides a powerful way to explore attack graphs, but for realistic networks, attack graph complexity often greatly exceeds the human capacity for understanding. While significant work has been done on generating attack graphs automatically, the management of attack graph visual complexity has until now remained a largely unsolved problem.

We describe a framework for managing attack graph complexity through hierarchical graph aggregation, in which non-overlapping attack subgraphs are recursively collapsed to single vertices. This allows an arbitrary degree of complexity compression, with higher levels of aggregation corresponding to higher levels of abstraction, and aggregation or de-aggregation occurring either top-down or bottom-up. We define a hierarchy of rules that controls aggregation at each level of the hierarchy, which is particularly useful for automatic aggregation at each level. The rules are based on either common attribute values for attack graph elements or attack subgraph connectedness, and can be efficiently implemented. The overall worst-case computational complexity in our framework is quadratic, but users often interact with small portions of the attack graph at a given time, making computation negligible.

We introduce the novel protection domain abstraction, which is particularly effective at reducing attack graph complexity while preserving problem semantics. The protection domain represents a fully connected subgraph (clique) among machines and exploits. Protection domains are known from the network configuration (e.g., subnets and firewall rules), so that we avoid the expense of finding attack graph cliques.

We apply our framework to a real network to demonstrate its effectiveness at managing attack graph complexity. This demonstration also shows our novel visual representations for the abstractions defined by our aggregation hierarchy. For the demonstration, we apply a software system we have developed for generating and visualizing network attack graphs. Our system builds a network vulnerability/connectivity model automatically via the Nessus vulnerability scanner, and includes several hundred modeled potential exploits. Given a starting machine and network attack goal, the system then builds an attack graph of exploits and their dependencies, and allows one to explore the resulting attack graph through interactive visualization.

Attack graphs can show how an attacker might combine individual vulnerabilities to seriously compromise a network. Automatic attack graph generation can exceed human analytical capacity, and encourages inexpensive "what-if" analysis in which proposed network configurations are tested for overall impact on network security. Previous work has demonstrated that scalable approaches are possible for attack graph generation. We describe a framework in which attack graphs are not only *computationally* scalable, but also *cognitively* scalable. Our rich visual representations capture relationships among semantically-relevant attack graph abstractions. As a result, the application of attack graph analysis to real networks becomes significantly closer to reality.

7. ACKNOWLEDGMENTS

The work of Sushil Jajodia was partially supported by the Air Force Research Laboratory, Rome under the contract F30602-00-2-0512, Army Research Office under the grant DAAD19-03-1-0257, and Army Research Laboratory under the contract W911QX-04-C-0101.

8. REFERENCES

[1] C. Ramakrishnan, R. Sekar, "Model-Based Analysis of Configuration Vulnerabilities," in *Proceedings of the 7th ACM Conference on Computer and Communication Security*, November 2000.

[2] R. Ritchey, P. Ammann, "Using Model Checking to Analyze Network Vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2000.

[3] O. Sheyner, J. Haines, S. Jha, R. Lippmann, J. Wing, "Automated Generation and Analysis of Attack Graphs," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2002.

[4] S. Jha, O. Sheyner, J. Wing, "Two Formal Analyses of Attack Graphs," in *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, Nova Scotia, Canada, June 2002.

[5] R. Baldwin, *Kuang: Rule Based Security Checking*, Technical Report, MIT Lab for Computer Science, May 1994.

[6] D. Zerkle, K. Levitt, "Netkuang – A Multi-Host Configuration Vulnerability Checker," in *Proceedings of the 6th USENIX Unix Security Symposium*, San Jose, CA, 1996.

[7] C. Phillips, L. Swiler, "A Graph-Based System for Network-Vulnerability Analysis," in *Proceedings of the New Security Paradigms Workshop*, Charlottesville, VA, 1998.

[8] L. Swiler, C. Phillips, D. Ellis, S. Chakerian, "Computer-Attack Graph Generation Tool," in *Proceedings of the DARPA Information Survivability Conference & Exposition II*, June 2001.

[9] J. Dawkins, C. Campbell, J. Hale, "Modeling Network Attacks: Extending the Attack Tree Paradigm," in *Proceedings of the Workshop on Statistical and Machine Learning Techniques in Computer Intrusion Detection*, Johns Hopkins University, June 2002.

[10] P. Ammann, D. Wijesekera, S. Kaushik, "Scalable, Graph-Based Network Vulnerability Analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, Washington, DC, November 2002.

[11] S. Jajodia, S. Noel, B. O'Berry, "Topological Analysis of Network Attack Vulnerability," in *Managing Cyber Threats: Issues, Approaches and Challenges*, V. Kumar, J. Srivastava, A. Lazarevic (eds.), Kluwer Academic Publisher, 2003.

[12] F. Cuppens, R. Ortalo, "LAMBDA: A Language to Model a Database for Detection of Attacks," in *Proceedings of the 3rd International Workshop on Recent Advances in Intrusion Detection*, Toulouse, France, October 2000.

[13] S. Templeton, K. Levitt, "A Requires/Provides Model for Computer Attacks," in *Proceedings of the New Security Paradigms Workshop*, Cork Ireland, 2000.

[14] R. Ritchey, B. O'Berry, S. Noel, "Representing TCP/IP Connectivity for Topological Analysis of Network Security," in *Proceedings of the 18th Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 2002.

[15] F. Cuppens, A. Miege, "Alert Correlation in a Cooperative Intrusion Detection Framework," in Proceedings of the 2002 IEEE Symposium on Security and Privacy, May 2002.

[16] P. Ning, Y. Cui, D. Reeves, "Constructing Attack Scenarios through Correlation of Intrusion Alerts," in *Proceedings of the 9th ACM Conference on Computer & Communications Security*, Washington D.C., November 2002.

[17] P. Ning, D. Xu, C. Healey, R. St. Amant, "Building Attack Scenarios through Integration of Complementary Alert Correlation Methods," in *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, February, 2004.

[18] S. Noel, S. Jajodia, "Correlating Intrusion Events and Building Attack Scenarios through Attack Graph Distances," submitted.

[19] P. Eades, Q.-W. Feng, "Multilevel Visualization of Clustered Graphs," in *Proceedings of the Symposium on Graph Drawing*, September, 1996.

[20] A. Buchsbaum, J. Westbrook, "Maintaining Hierarchical Graph Views," in *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, 2000.

[21] M. Raitner, "HGV: A Library for Hierarchies, Graphs, and Views," in *Proceedings of the Symposium on Graph Drawing*, 2002.

[22] M. Raitner, *Maintaining Hierarchical Graph Views for Dynamic Graphs*, Technical Report, MIP-0403, University of Passau, January, 2004.

[23] S. Noel, S. Jajodia, B. O'Berry, M. Jacobs, "Efficient Minimum-Cost Network Hardening via Exploit Dependency Graphs," in *Proceedings of the 19th Annual Computer Security Applications Conference*, Las Vegas, Nevada, December 2003.

[24] Nessus vulnerability scanner, http://www.nessus.org/.