Distributed Attack Graph Generation

Kerem Kaynar and Fikret Sivrikaya

Abstract—Attack graphs show possible paths that an attacker can use to intrude into a target network and gain privileges through series of vulnerability exploits. The computation of attack graphs suffers from the state explosion problem occurring most notably when the number of vulnerabilities in the target network grows large. Parallel computation of attack graphs can be utilized to attenuate this problem. When employed in online network security evaluation, the computation of attack graphs can be triggered with the correlated intrusion alerts received from sensors scattered throughout the target network. In such cases, distributed computation of attack graphs becomes valuable. This article introduces a parallel and distributed memory-based algorithm that builds vulnerability-based attack graphs on a distributed multi-agent platform. A virtual shared memory abstraction is proposed to be used over such a platform, whose memory pages are initialized by partitioning the network reachability information. We demonstrate the feasibility of parallel distributed computation of attack graphs and show that even a small degree of parallelism can effectively speed up the generation process as the problem size grows. We also introduce a rich attack template and network model in order to form chains of vulnerability exploits in attack graphs more precisely.

Index Terms—Attack graph, reachability, vulnerability, weakness, exploit, distributed computing

1 Introduction

An attack graph shows possible paths that an attacker can follow to intrude into a network and gain certain target privileges, given a set of initially satisfied privileges as input. It generally represents the relationships among various vulnerability exploits that can be employed by the attacker and the privileges gained by the attacker as a result of exploiting these vulnerabilities. The term vulnerability is treated in different ways in the context of attack graph generation in the literature. In some cases, an inappropriate configuration setting is considered as a vulnerability. In some others, the existence of a specific version of a software product is considered as a vulnerability.

The number of vulnerabilities on the target network, the reachability conditions among vulnerable software instances and the level of detail in vulnerability modelling are the main factors determining the size of the state space during the attack graph generation process. States in this context correspond to nodes in the resulting attack graph. When the state space becomes too large, the popular serial attack graph generation algorithms, running on a single computer and generally employing hashes or other in-memory data structures to hold the already traversed parts of the state space, may become ineffective.

A distributed solution to the attack graph generation problem is proposed in this paper to overcome the state explosion issue. The problem is treated as a graph traversal or search problem, where the graph to be traversed is also built on the fly by applying the proposed distributed algorithm. We use a specific virtual shared memory abstraction over a distributed multi-agent platform to eliminate duplicate traversal of the graph parts. Network reachability information is used to initialize shared memory pages in order to minimize the number of messages transferred among the agents.

Building vulnerability-based attack graphs requires a proper *attack template model*, which represents vulnerabilities with their pre and postconditions. Possible pre and postconditions for vulnerabilities considered in the current literature include vulnerable product specification, the system access level (root, admin, user, etc.) on the attacking and target hosts, and the existence of network connection among the attacking and target host. We provide an attack template model enriching the possible values for pre and postconditions. We introduce indirect conditions in the attack template model, allowing a condition to be defined in terms of product type (technology class of the product, like web server, database client, etc.) instead of a full product specification. We use relative locations for conditions, allowing for instance an attacker to gain privileges on the backend application of the attacked (target) application. The weaknesses defined in CWE [1] and vulnerability-weakness mappings provided by NVD [2] are utilized to derive pre and postconditions for vulnerabilities.

Another aspect that plays an important role in the attack graph generation is *network modelling*, which is concerned with representing the topology and reachability conditions among network hosts. Reachability conditions are affected by filtering rules and security policies employed across the network, among others. They may also be considered as possible preconditions of vulnerabilities. We introduce a new network model based on the software applications installed on network hosts and their information sources. An attacker can gain certain privileges on a host by benefiting from the information sources of a software application on another host. As an example, by gaining access to the cookies file, an attacker

- *K. Kaynar is with GT-ARC, TU Berlin, Germany*
- *F. Sivrikaya is with the DAI-Labor Department, Technical University, Berlin, Germany.*

Manuscript received 11 June 2014; revised 24 Mar. 2015; accepted 4 Apr. 2015. Date of publication 16 Apr. 2015; date of current version 2 Sept. 2016. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TDSC.2015.2423682

can obtain authorization privilege on a different web application, possibly a server-side application. Our network model takes such interactions into account in specifying the reachability conditions.

In the remainder of this section, we present the related work in the domain of attack graph generation, which is followed by an overview of the main contributions of this work. The distributed attack graph generation mechanism proposed in this paper is then detailed in two comprehensive parts. The first part, which is related with the modelling of network security objects in the context of attack graphs, is explained in Section 2. The second part comprises the main distributed attack graph building algorithm, which is explained in Section 3. Section 4 provides the experimental results for the proposed attack graph building mechanism. The paper concludes with a summary and recommendations for future work in Section 5.

1.1 Related Work

We can consider the attack graph generation process to be composed of three main phases: *Attack graph modelling*, *Information collection* and *Attack graph core building*. *Attack graph modelling* includes attack template modelling, attack graph structure determination and network modelling. An attack template model represents a vulnerability with its pre and postconditions. The attack graph structure determines the node and edge types in the generated attack graphs. The network model defines the software running on the network hosts and their relationships. *Information collection* includes collecting vulnerability and target network configuration information. Target network configuration information generally contains the network topology, the software applications installed (or running) on the target network hosts and the filtering rules. Network topology information can be collected by using network mappers (topology builders), while software information can be collected using network and host-based vulnerability scanners, asset managers, etc. The methods and tools that can be used in the information collection part are outside the scope of this paper. Finally, *Attack graph core building* is concerned with the actual graph generation stage, which we will cover in greater detail in the rest of this article.

The selected related literature here focuses mainly on the attack graph modelling and core building phases, which are at the heart of the contributions of the current work.

In terms of attack graph modelling, a generic model for describing attack scenarios based on a specific attack specification language, called JIGSAW, is presented in [3]. The authors propose to use mutation, resequencing, substitution, distribution and looping methods to create variants of sample attack scenarios. In [4], the intrusion correlator, developed previously by the authors, is used to correlate intrusion alerts and produce hyper alert correlation graphsas a result. Hyper alert correlation graphs are used to create attack strategy graphs, whose nodes represent high level attack stages and edges represent causal relations between attack stages. In [5], attack scenarios are modelled in a hierarchical structure. The lowest level represents (atomic) vulnerability exploits used by an attacker, the highest-level tries to summarize attack occurrences into several stages (reconnaissance, penetration, privilege escalation, trace hiding, etc.). All those works in the literature use essentially a similar directed graph structure, relating cause and consequence privileges for atomic vulnerability exploits.

When the attack graph core building process is considered, there is no work in the literature proposing a parallel, distributed algorithm. A parallel, distributed algorithm for the attack graph core building process can surely attenuate the effects of the state explosion problem occurring when the number of vulnerabilities in the target network grows large. It can spread the attack graph building process into more than one computer to eliminate the scalability restriction for the total number of vulnerabilities on the software running on the target network.

In [6], model checking is used to find a single counter example to a given safety property. In [7], the authors develop a model checking tool called NuSMV to compute all the counter examples to a given safety property. One of the approaches resembling to the model checking approach is the logic deduction approach. In [8], an attack graph computation approach based on logic programming is proposed. Logic deduction is applied to reach from the initial facts to the goal facts, giving the resulting attack graph at the end. Both logic deduction and model checking approaches suffer from the state explosion problem stated above. As a result, more importance is given to the methods concentrating on graph search (traversal) approaches.

A breadth-first search-based attack graph building algorithm is proposed in [9]. The authors use a layered attack graph structure, where the bottom layer contains the privileges initially satisfied by an attacker. The upper layer privileges are computed and updated after each iteration of the given algorithm. Already traversed privileges are stored in layers which are checked to eliminate duplicate expansion of the same node. This layered approach on a single machine can be a bottleneck for the execution time of the serial search algorithm, when the number of privileges and vulnerability exploits on the layers grows large.

In [10], an attack graph is constructed so that it only represents the worst-case attack scenario that can be utilized by an attacker. A node on the attack graph corresponds to a host in the target network and an edge represents the vulnerability exploit used by an attacker to obtain the *highest access level* (e.g., administrator) on the target node when attacking from the source node. This worst-case scenario approach is designed to cope with the state explosion problem, however it can not give all the attack paths that need to be patched as stated by the authors.

A new method based on reachability groups is introduced in [11] to decrease the space complexity of the reachability matrix (representing reachability conditions among network hosts). [11] also proposes a new attack graph structure, called multiple-prerequisite graphs, expressing the reachability conditions among network hosts as nodes in the attack graph. Using this new structure and a breadth-first search method, the authors claim to achieve a time complexity *linear in the number of hosts in the target network* for the attack graph generation process, when the reachability groups have a coarse-grained distribution. The main contribution of this work is a space-efficient grouping of reachability conditions giving rise to a decreased number of reachability-based privilege nodes in the resulting attack

graph. The serial search-based core attack graph generation algorithm uses hashes stored in a single computer to eliminate the duplicate attack graph node traversals (expansions). This approach may not be scalable, when faced with the state explosion problem. There are experiments made with around 50,000 network hosts and over 1.5 million ports (each has ten vulnerabilities) discussed in this paper and ran in under four minutes. However, the only privilege types they use are the system access level, reachability and credentials. When the number of these types increases, the results may be affected quadratically. In their network setup, there are 400 filtering rules in each border firewall. Since these rules block a lot of traffic (decreasing the number of attack graph nodes) and the number of nodes in the resulting attack graph is not given in the paper, a solid decision of the scalability of the proposed method may not be performed easily. In [12], multiple-prerequisite graphs are improved to reflect the effects of client-side attacks, personal firewalls, intrusion prevention systems and proxy firewalls by introducing the reverse reachability concept.

A method for determining and removing useless edges from an input attack graph is studied in [13]. In [14], most probably exploitable attack paths are included during the computation of the attack graphs. They introduce three types of probability values that can be used to determine the likelihood of successful utilization of attack paths and prune the attack paths during the attack graph building process. In [15], the authors propose to apply a bidirectional search method to the attack graph generation problem. Forward search starts from the initial attacker states and backward search starts from the goal states. The authors also propose to apply a depth limitation policy during each search to prune the less likely attack paths. Our approach can also be used with a specific pruning criteria to decrease the effects of the state explosion problem much more. In [16], a specially designed search algorithm from artificial intelligence domain, called Planner, is used to generate attack graphs in order to alleviate the effects of the scalability problem. Customized algorithms are proposed for automatic generation of attack paths in polynomial time using Planner as the base module.

An abstraction over vulnerabilities, called attack patterns, is introduced in [17]. An attack pattern represents a set of vulnerabilities that an attacker can utilize by common methods of exploits. The authors use the information in CAPEC [18] database to define their own attack patterns. They group vulnerabilities in attack patterns to simplify the task of the derivation of pre and postconditions for vulnerabilities. We introduce pre and postconditions using the weaknesses in CWE [1] similar to their approach.

1.2 Contributions

Our main contribution in this article is related with the *Attack Graph Core Building* phase of attack graph generation and we try to improve the time and space complexity of the problem by applying a parallel and distributed algorithm to cope with the potential state explosion problem in the resulting attack graphs.

A hyper-graph is used to represent the reachability conditions among networked software applications, in which a hyper-vertex represents a networked software application and a hyper-edge indicates a collection of networked software applications which can access each other using a specific network protocol.

One of the main contributions of this paper is to perform reachability hyper-graph partitioning to guide the distributed search performed to generate an attack graph. Distributed search is performed on a multi-agent distributed platform and the tasks assigned to each agent is determined by applying partitioning to the reachability hyper-graph. Therefore, the tasks of the determination of the attacker privileges that can be obtained on networked software applications are distributed over agents in such a way that an agent can perform attacker privilege determination on the networked software applications that are closely connected. In summary, each distributed agent has to perform attacker privilege determination tasks for a number of closely connected networked software applications, and the list of networked software applications handled by an agent can be changed dynamically at run-time.

Another contribution of this paper is to provide a virtual shared memory abstraction across distributed agents to avoid multiple expansion of nodes (multiple expansion of the attacker privileges) in the resulting attack graph. The algorithm described in [19] is used to provide a virtual shared memory abstraction over a distributed multi-agent system. One of the important parameters of a virtual shared memory abstraction is the initial allocation of memory pages (or, memory objects) to the distributed agents. This is performed by partitioning the reachability hyper-graph, so that the more closely connected any two networked software applications, the more likely that they can be in the same memory page and thus handled by the same agent without having to transfer memory pages from other agents.

The next two sections describe the attack graph modelling and attack graph core building parts of our attack graph generation system in detail.

2 Attack Graph Modelling

As introduced earlier, attack graph modelling is concerned with attack template modelling, attack graph structure determination, and network modelling. Attack template modelling comprises the representation of pre and postconditions for vulnerabilities. It also includes a mechanism for derivation of these conditions for specific vulnerabilities using the information in NVD [2] vulnerability and CWE [1] weakness databases. The determination of attack graph structure includes deciding which types of nodes and edges can be found in an attack graph. Network modelling aims to determine an appropriate representation for the network assets (e.g., software applications running on the network hosts). In the following sections, each of these sub-parts is described in detail.

2.1 Attack Template Modelling

Attack template modelling determines a model for vulnerabilities and their pre and postconditions and also suggests a method for creating instances of them. The attack template model used in our system is shown in Fig. 1, with the formal definitions of its components following next.

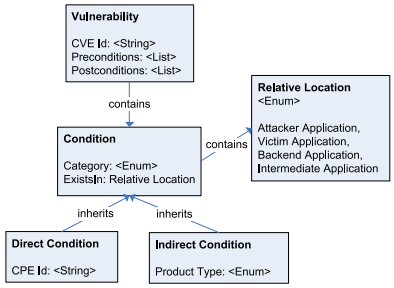

Fig. 1. Attack template model.

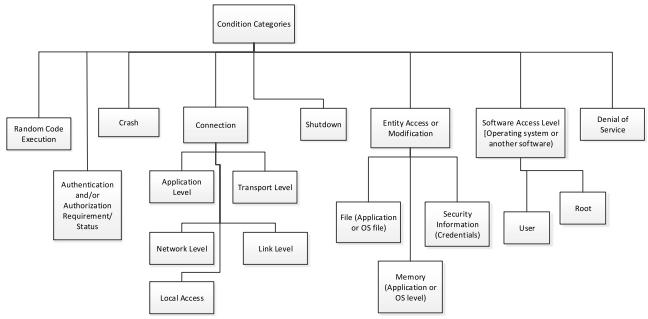

Fig. 2. Pre and postcondition categories.

Definition 2.1. *A condition represents a right that can be gained on a software application. It is a two element tuple* $\langle Category, ExistsIn \rangle$*, where* $Category$ *represents the right gained on a software application and* $ExistsIn$ *represents the location of the software application on which the right is gained. The location is determined relative to the attacker and victim software application. It can be attacker software application, victim software application, a backend application of the victim software application or an intermediate software application that is located between the attacker and victim application and can intercept the traffic between them.*

A condition is independent from any IP network and does not specify an IP address in its definition. For conditions, in the first place, possible values for the category (right) of a condition shall be determined. The level of detail for the condition categories shall be adjusted carefully to adequately express the pre and postconditions of the vulnerabilities and also to eliminate the chaining of unrelated vulnerabilities in the resulting attack graphs. The proposed scheme for the determination of condition categories is illustrated in Fig. 2. Condition categories are hierarchically organized.

Definition 2.2. *A direct condition specifies an additional element to a condition tuple. This element is named* $CPEId$ *and represents the CPE product identifier [20] of the software application on which the condition right (category) is gained.*

Definition 2.3. *An indirect condition specifies an additional element to a condition tuple. This element is named* $ProductType$ *and specifies the product type of the software application on which the condition right (category) is gained.*

The product types are defined in the system. A product type can be mail server, mail client, web server, web client, ftp client, database server application, etc. indicating the technology class of the software application. We have manually derived product types for around 30,000 CPE identifiers related with the mostly used products in the information technology sector. The derived product type-CPE identifier matchings are used in the attack graph building process.

Definition 2.4. *A vulnerability is considered as a single CVE entry defined in CVE ([21], [2]) database. It is represented by a three element tuple* $\langle CVEId, Preconditions, Postconditions \rangle$*. A vulnerability is identified by its unique CVE [21] identifier stored in* $CVEId$ *element. It has a list of preconditions that denote the list of required attacker privileges to exploit the vulnerability. The preconditions are stored in the list* $Preconditions$*. There is a conjunction relation among the preconditions. A vulnerability also has a list of postconditions that denote the privileges obtained by the attacker after successfully exploiting the vulnerability. The postconditions are stored in the list* $Postconditions$*. There is a disjunctive relation among the postconditions. A precondition and a postcondition can be a direct or indirect condition.*

There are over 60,000 vulnerabilities described in NVD [2]. Their descriptions do not allow automatic generation of pre and postconditions easily, and mostly are not detailed enough to derive pre and postconditions in an adequate granularity. As an example, only access vector and authentication are considered as possible preconditions for most of the vulnerabilities. Our method for generating pre and postconditions for vulnerabilities is based on the weaknesses defined in CWE [1] and the vulnerability-weakness mappings provided by NVD. We manually generate pre and postconditions for the weaknesses existing on a specific weakness hierarchy formed by CWE. *Research Concepts (CWE-1000)* hierarchy is used. The relatively small number of weaknesses (on the order of 100) allows manual pre and postcondition generation according to our attack template model. In order to derive preconditions for weaknesses, *Description*, *Applicable Platforms*, *Modes of Introduction* and *Enabling Factors for Exploitation* fields of CWE weakness records are examined manually. In order to derive postconditions for weaknesses, *Description*, *Applicable Platforms* and *Common Consequences* fields of CWE weakness records are examined manually.

The vulnerability descriptions in NVD are processed by simple XML parsing methods to derive the related weakness identifier and CPE identifiers for each vulnerability. CPE identifiers denote the software products containing the vulnerability. After that, pre and postconditions for a vulnerability are derived using:

1) fetching manually defined pre and postconditions for the weakness related with the vulnerability,
2) filling empty CPE Id fields in the fetched pre and postconditions with the CPE identifiers for the vulnerability, if such empty fields exist. If there are more than one CPE identifiers containing the vulnerability, a distinct copy of the pre and postconditions is created for each CPE identifier.

An example can be given using the SQL injection weakness with CWE identifier CWE-89 and the vulnerability with CVE identifier CVE-2013-7375. The manually derived preconditions for the SQL injection weakness are:

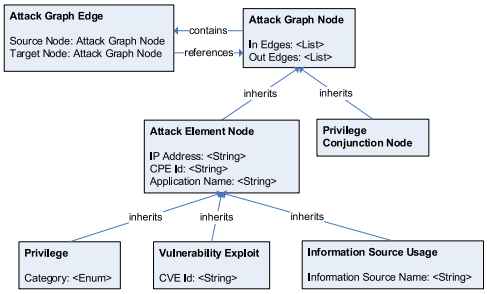

Fig. 3. Attack graph structure.

1) *Transport Level Connection to an Application with an Empty CPE Id on Victim Side.*

The manually derived postconditions for the SQL injection weakness are:

1) *Authorization on an Application with an Empty CPE Id on Victim Side,*
2) *File Modification on an Application with Product Type Database Server on Backend Side.*

After simply (XML) parsing the definition of the vulnerability CVE-2013-7375 in NVD, the related weakness (SQL injection) of it and the CPE identifiers of the software products containing it are determined. There are five CPE identifiers related with the vulnerability. For each of them, the pre and postconditions of the SQL injection weakness are copied and empty CPE identifiers are replaced with them. As can be derived easily, it is meaningful to include empty CPE identifiers on weakness pre and postconditions, which are defined on applications only on the *victim side*. Because, after parsing a vulnerability, we only have concrete CPE identifiers related with the vulnerability on the victim host. However, it is allowed to include non-empty CPE identifiers on weakness pre and postconditions for attacker, victim, intermediate and backend sides.

2.2 Attack Graph Structure Determination

Attack graph structure determines the nodes and the edges of the generated attack graphs. The proposed attack graph structure is shown in Fig. 3. An attack graph can contain four types of nodes. The formal definitions of the different types of nodes in an attack graph and attack graph are given below.

Definition 2.5. *A privilege node represents an attacker privilege on a software application on a network host and is a six element tuple* $\langle IPAddress, CPEId, ApplicationName, Category, InEdges, OutEdges\rangle$. *$IPAddress$ denotes the IP address on which the software application is running. $CPEId$ is the CPE [20] identifier of the product related with the software application. $ApplicationName$ is the name of the software application. $Category$ represents the category of the condition possessed by the attacker on the software application. Possible condition categories are shown in Fig. 2. $InEdges$ and $OutEdges$ are lists holding references to the in and out attack graph edges connected to the privilege node.*

Definition 2.6. *A privilege conjunction node denotes a conjunction connector for a number of privilege nodes in an attack graph. It is a two element tuple* $\langle InEdges, OutEdges\rangle$. *$InEdges$ and $OutEdges$ are lists holding references to the in and out attack graph edges connected to the privilege conjunction node.*

Definition 2.7. *A vulnerability exploit node represents an exploit of a vulnerability on a software application on a network host by an attacker. It is a six element tuple* $\langle IPAddress, CPEId, ApplicationName, CVEId, InEdges, OutEdges\rangle$. $IPAddress$, $CPEId$, $ApplicationName$ *elements are defined same as in Definition 2.5. $CVEId$ is the unique identifier of the exploited vulnerability and is defined in CVE [21] database. $InEdges$ and $OutEdges$ are lists holding references to the in and out attack graph edges connected to the vulnerability exploit node.*

Definition 2.8. *An information source usage node represents an access and usage of an information source (cookie file, DNS table, database table, etc.) on a software application on a network host by an attacker. It is a six element tuple* $\langle IPAddress, CPEId, ApplicationName, ISName, InEdges, OutEdges\rangle$. $IPAddress$, $CPEId$, $ApplicationName$ *elements are defined same as in Definition 2.5. $ISName$ is the name of the used information source inside the software application. $InEdges$ and $OutEdges$ are lists holding references to the in and out attack graph edges connected to the information source usage node.*

Definition 2.9. *An attack graph is a graph* $G = (N, E)$, *where N denotes the set of nodes and E denotes the set of edges of the graph G. $n \in N$ can be a privilege, privilege conjunction, vulnerability exploit or information source usage node. $e \in E$ is a two element tuple* $\langle SourceNode, TargetNode\rangle$ *where $SourceNode$ denotes the source and $TargetNode$ denotes the target node for the edge e. Possible node types for $SourceNode$ and $TargetNode$ are* $\langle Pr, Ve\rangle$, $\langle Pr, Pc\rangle$, $\langle Pr, Isu\rangle$, $\langle Pc, Ve\rangle$, $\langle Pc, Isu\rangle$, $\langle Ve, Pr\rangle$ *and* $\langle Isu, Pr\rangle$, *where Pr denotes a privilege, Ve denotes a vulnerability exploit, Pc denotes a privilege conjunction and Isu denotes an information source usage node.*

The edges of an attack graph just denote relationships among different types of nodes. There can be an edge:

1) from a privilege to a vulnerability exploit or an information source usage indicating that the existence of only the privilege is sufficient and necessary for an attacker to exploit the vulnerability or the information source usage,
2) from a privilege to a privilege conjunction indicating that the privilege is one of the necessary preconditions for an attacker to exploit the vulnerability or information source usage targeted by the privilege conjunction,
3) from a privilege conjunction to a vulnerability exploit or an information source usage indicating that the acquisition of the privileges connected to the privilege conjunction by an attacker gives rise to the vulnerability exploit or the usage of the information source,
4) from a vulnerability exploit or an information source usage to a privilege indicating that the exploit of the vulnerability or usage of the information source gives rise to the acquisition of the privilege by an attacker.

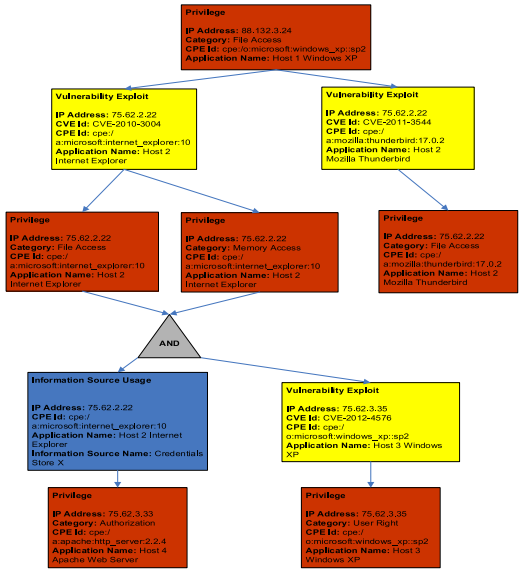

Fig. 4. An example attack graph.

An example attack graph is shown in Fig. 4.

2.3 Network Model

Network model is used to model the target network topology and installed software configuration. It includes network hosts as main elements. Network hosts are connected to each other via their contained network interfaces. Each network interface can have an IP address and a reference to the communication link, which connects it to another network interface, possibly contained in another network host. Our network model is illustrated in Fig. 5, and formally defined next.

Definition 2.10. *A network host is a two element tuple* $\langle NetworkInterfaces, SoftwareApplications\rangle$, *where* $NetworkInterfaces$ *is a list of network interfaces contained by the network host and* $SoftwareApplications$ *is a list of installed software applications on the network host.*

Definition 2.11. *A network interface denotes a OSI Layer 3 interface on a network host and is a three element tuple* $\langle IPAddress, Link, Host\rangle$. $IPAddress$ *is the IP address associated with the network interface.* $Link$ *denotes the communication link connected to the network interface.* $Host$ *denotes the network host containing the network interface.*

Definition 2.12. *A communication link connects two network interfaces and is a two element tuple* $\langle SourceNI, TargetNI\rangle$. $SourceNI$ *refers to the source network interface,* $TargetNI$ *refers to the target network interface connected to the communication link. The designation of source or target network interface is performed randomly and does not impose any restriction on the data transfer direction between the network hosts containing the network interfaces.*

Definition 2.13. *A software application is a five element tuple* $\langle CPEId, HostIPAddress, Port, BackendApplications, InformationSources\rangle$. $CPEId$ *denotes the software product identifier (CPE identifier [20]),* $HostIPAddress$ *denotes the IP address on which the software application is serving and* $Port$ *denotes the port on which it is serving.* $BackendApplications$ *refers to the software applications whose services are used by this software application.* $InformationSources$ *is a list of information sources contained by the software application such as credentials store, cookies, DNS table, routing table, databases.*

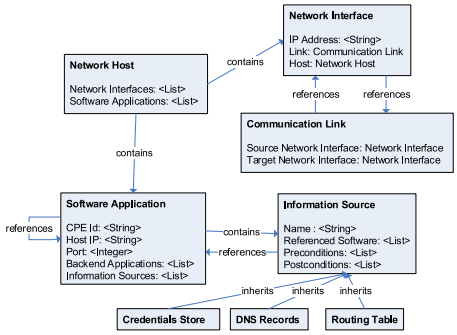

Fig. 5. Network model.

A software application object stores references to other software applications used by the software application in its backend. For instance, a web server application may reference a database server application as its backend application. If an attacker uses a specific vulnerability on this web server application, she can gain privileges on the backend database server application without even using a vulnerability on the database server application. The backend software applications are considered while building attack graphs with the proposed distributed search algorithm.

Definition 2.14. *An information source denotes a sensitive data store that is contained by a software application and can be accessed and used by an attacker. It is represented by a three tuple* $\langle ReferencedSoftware, Preconditions, Postconditions\rangle$. *In order to use an information source, an attacker should satisfy the preconditions that are stored in the list* $Preconditions$ *for the information source. After successfully benefiting from the information source, the attacker gains the postconditions that are stored in the list* $Postconditions$ *for the information source. The postconditions are gained on the software applications referenced by the information source that are stored by the element* $ReferencedSoftware$.

An attacker can gain privileges on a host by benefiting from the information sources on a software application on a different host, e.g., without using any vulnerability defined in NVD. As an example, an attacker, gaining access to the cookies file on a web browser on one host, may gain authorization rights to a web application on another host by using the information stored in the cookies file.

3 Attack Graph Core Building Mechanism

Attack graph core building process is treated as a search problem in this paper, which is solved in a distributed multi-agent environment. For this purpose, a reachability hyper-graph is first formed and then, a modified parallel version of the classical depth-first search algorithm is applied by considering reachability conditions stored in the hyper-graph. One of the main issues in parallel searching in a distributed memory environment is the difficulty of eliminating duplicate attack graph node expansions. To eliminate this problem, a virtual shared memory abstraction

is implemented to provide memory coherency over distributed search agents.

3.1 Reachability Hyper-Graph and Partitioning

Reachability determines the accessibility conditions among the software applications installed on the target network. Filtering rules on firewalls, access control lists on routers, security policies applied among the software applications are among the factors that determine the reachability conditions. All these factors are taken into account to create our reachability hyper-graph, in which a hyper-vertex indicates a software application. Each hyper-vertex has an associated weight value that indicates the density of the workload related with the software application. The computation of this workload accounts for the number of the attack graph nodes that will be created by the search agents related with the software application.

A hyper-edge indicates a collection of source and target software applications such that source applications can directly access target applications, if they satisfy specific conditions. These conditions, allowing direct reachability among the software applications, are stored in the hyper-edge. Such conditions can include network protocols, port numbers, user credentials, etc. A hyper-graph is more efficient in terms of storage space than a reachability matrix or graph.

The resulting reachability hyper-graph is partitioned to determine the initial tasks of each distributed search agent responsible for generating the attack graph. Each search agent shall be responsible for determining the attacker privileges, vulnerability exploits and information source usages for a number of software applications. The software applications are allocated to the search agents with hyper-graph partitioning in such a way that the number of messages transferred among the search agents, when building the resulting attack graph, is minimized and the workload of the search agents are balanced. Any hyper-graph partitioning algorithm can be used to partition the reachability hyper-graph, e.g. [22].

As stated in [22], hyper-graph partitioning is a significant problem with many application areas, including VLSI design, efficient storage of large databases on disks and data mining. The problem is to partition the vertices of a hyper-graph into k roughly equal parts, such that a certain objective function defined over the hyper-edges is optimized. A commonly used objective function is to minimize the number of hyper-edges that span different partitions. These hyper-edges are cluttered across different partitions.

In [22], the authors propose a hyper-graph partitioning method based on greedy k-way partition refinement algorithm. They successively coarsen the given hyper-graph into smaller representative hyper-graphs, partition the smallest representative hyper-graph into k roughly equal parts and then refine the partitions by traversing the representative hyper-graphs back to the original hyper-graph. The partition refinement algorithm is performed by following a greedy approach that selects the vertices to be moved across the partitions by considering the greatest positive reduction (gain) in the objective function caused by the vertex movements. The greedy approach also accounts for load balancing constraints among the partitions in order to create

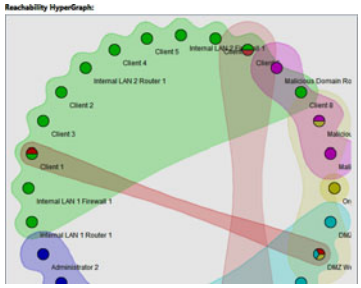

Fig. 6. An example reachability hyper-graph.

roughly equal partitions. Hyper-vertex weights are considered in load balancing.

The main aim of reachability hyper-graph partitioning is to achieve load balancing in terms of hyper-vertex weights and minimize the number of hyper-edges spanning across the search agents. We assign large weight values to hyper-vertices (software applications) related with the initial attacker privileges, since main work density in attack graph computation will be related with (i.e., will be in the neighbourhood of) these hyper-vertices. The minimization of the cluttered edges across search agents provides for the minimization of the transferred messages among the search agents during attack graph computation. An example reachability hyper-graph is depicted in Fig. 6. Software applications that have a common color in Fig. 6 can directly access each other. After partitioning, the reachability hyper-graph contains minimum number of cluttered hyper-edges and the total weight of the hyper-vertices contained in each partition is comparable (load balancing). The hyper-vertices (software applications), which are related with the initially satisfied attacker privileges, are assigned higher weight values than other hyper-vertices, so that the hyper-vertices (the workload) can be distributed to the search agents almost equally.

3.2 Virtual Shared Memory Abstraction

The main aim of the virtual shared memory abstraction is to provide memory coherency across all distributed search agents. *Dynamic Distributed Memory Manager Algorithm* described in [19] is employed in this paper. In this algorithm, there is no centralized memory manager. Each distributed search agent stores information about attacker privileges for the software applications that are assigned to it in its local memory. This information at least comprises the expansion status of the corresponding privileges, namely whether they are used to exploit some vulnerabilities or benefit from information sources to generate additional privileges.

As a distributed search agent is performing, it may need to expand an attacker privilege and request information (expansion status) about the attacker privilege, which is stored in another agent's local memory. In this case, it has to transfer the corresponding information into its local memory. Some messages are transferred among the search agents according to the algorithm described in [19] to transfer the corresponding memory page to the local memory of the requesting search agent. If the corresponding information indicates that the associated privilege has already been expanded, then the requesting agent does not expand it.

```
1: procedure PERFORMDFS(RHG, IPRGS)  ▷ Reachability hyper-graph (RHG) and initial attacker privileges (IPRGS) are inputs
2:     MainStack ←CreateMainStack()  ▷ Create the search main stack
3:     for all ip ∈ IPRGS do
4:         ips ←new PrivilegeStatus()
5:         ips.setExpanded(true)
6:         WriteToSharedMemory(ip, ips)
7:         MainStack.push(ip)
8:         foundPrivileges.add(ip)
9:     end for
10:    while true do
11:        if MainStack.size() > 0 then  ▷ Continue to the search while there are privileges on the main stack
12:            cp ← MainStack.pop()
13:        else
14:            eps ←GetWorkFromOtherAgents()  ▷ Requests privileges from other agents to expand
15:            if eps.size() == 0 then
16:                break
17:            else
18:                MainStack.push(eps)
19:                foundPrivileges.addAll(eps)
20:                continue
21:            end if
22:        end if
23:        hv ←FindVertexForPriv(cp, RHG)
24:        ches ←FindContainingEdges(hv, RHG)
25:        gprgs ←new List()
26:        for all he ∈ ches do
27:            tsas ←FindTargetSoftwareApps(he)
28:            for all tsa ∈ tsas do
29:                for all v ∈ tsa.vulnerabilities() do
30:                    reqprgs ←CheckExploitability(v, cp, tsa)
31:                    if reqprgs! = null then  ▷ Vulnerability can be exploited by an attacker
32:                        vgps ←FindGainedPrivileges(v, cp, tsa)
33:                        gprgs.addAll(vgps)
34:                        UpdateAttackGraph(v,
35:                            reqprgs, vgps, tsa)
36:                    end if
37:                end for
38:                for all is ∈ tsa.infoSources() do
39:                    reqprgs ←CheckExploitability(is, cp, tsa)
40:                    if reqprgs! = null then  ▷ Information source can be used by an attacker
41:                        isgps ←FindGainedPrivileges(is, cp, tsa)
42:                        gprgs.addAll(isgps)
43:                        UpdateAttackGraph(is, reqprgs, isgps, tsa)
44:                    end if
45:                end for
46:            end for
47:        end for
48:        for all gp ∈ gprgs do
49:            newgps ←new PrivilegeStatus()
50:            newgps.setExpanded(true)
51:            oldgps ←ReadAndUpdateSharedMemory(gp, newgps)  ▷ ReadAndUpdateSharedMemory is an atomic operation that updates the status of its input privilege and returns its old status.
52:            if oldgps.expanded == false then
53:                MainStack.push(gp)
54:            end if
55:            foundPrivileges.add(gp)
56:        end for
57:    end while
58: end procedure
```

Fig. 7. Parallel, distributed shared memory-based depth-first search algorithm.

Otherwise, the requesting agent pushes the privilege to its search stack to expand it later, sets its expansion status to true and invalidates the copies of the corresponding memory page in other agents by sending messages to them.

In this proposed method, it is important to decrease the memory read/write faults, thereby memory page transfers. For this, it is crucial to determine a reasonable initial configuration for the virtual shared memory. At the initialization of the virtual shared memory, one must carefully select which memory pages contain which memory objects (privileges) and which search agents store which memory pages locally. In this paper, this is performed by reachability hyper-graph partitioning. Hyper-vertices in the reachability hyper-graph, representing software applications on which the privileges are defined, are distributed as equally as possible to the search agents. The number of hyper-edges cluttered across the search agents is minimized by the partitioning process to minimize the number of transferred messages among the search agents.

3.3 Parallel, Shared Memory-Based Depth-First Search

In this paper, a parallel, shared memory-based depth-first search method is proposed as the core attack graph building algorithm. Each distributed search agent performs this algorithm. Actually, each distributed search agent performs a stack-based depth-first search starting by expanding initially satisfied attacker privileges assigned to it. The decision of expanding a privilege during the search is determined by examining the Boolean expansion status of it stored in the virtual shared memory. If an agent generates a privilege, it first queries the shared memory to learn whether the privilege has already been expanded or not. If the privilege has not already been expanded, the agent sets its expansion status to true and pushes the privilege to its search stack to expand it later.

When a search agent has no privilege to expand on its search stack at a certain time, it starts to request one or more privilege from other search agents. When a search agent requests privilege from the search agent x, the search agent x sends a random number of privileges that are stored on the bottom half of its search stack to the requesting search agent, if it has sufficient privileges in its stack. If the request has resulted in no privilege transfer, then the requesting search agent tries other search agents. If no privilege transfer is occurred after a specified number of trials for each other search agent in a cyclic manner, then the requesting search agent gives up and enters into passive state, but does not terminate. When all the search agents enter into passive state, the distributed search algorithm is assumed to be terminated.

3.3.1 Parallel, Distributed Search Performed By Search Agents

The proposed parallel, distributed shared memory based depth-first search algorithm is given in Fig. 7. It is executed by each distributed search agent on the multi-agent platform. Before performing the search algorithm, reachability hyper-graph has already been generated and virtual shared memory pages have already been initialized with the hyper-graph partitioning results.

During the distributed search algorithm, each agent uses the previously computed reachability hyper-graph to determine which target software applications are reachable from the software application related with the currently processed privilege (*FindContainingEdges()* and *FindTargetSoftwareApps()*). Then, the vulnerabilities and information sources on each of these target software are fetched and their exploitability/usability are checked by calling the function *CheckExploitability()*. For each exploitable vulnerability and usable information source, the new privileges gained by the attacker are computed by calling the function

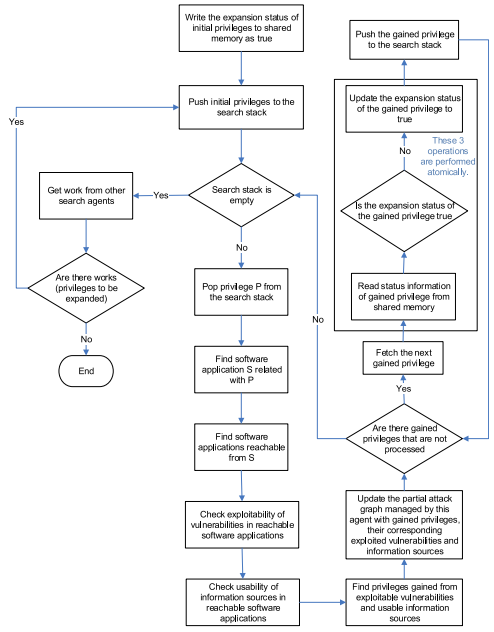

Fig. 8. Flow chart for the parallel, distributed shared memory-based depth-first search algorithm.

FindGainedPrivileges(). The expansion status of each gained privilege stored in the shared memory is checked. If the gained privilege has not already been expanded, its expansion status is updated to true in the shared memory and the gained privilege is pushed into the search stack. The two shared memory operations, namely reading the expansion status from the shared memory and updating it with new status object, are performed atomically by the function *ReadAndUpdateSharedMemory()*. More than one agent can not enter into this function simultaneously.

The flow chart for the parallel, distributed shared memory-based depth-first search algorithm performed by each search agent is also given in Fig. 8.

3.3.2 Checking Exploitability of Vulnerabilities and Information Sources

During the parallel, distributed shared memory-based depth-first search algorithm, the exploitability of each vulnerability/information source in software applications currently reachable by an attacker is checked via matching already expanded (searched) privileges in shared memory to the preconditions of the vulnerability/information source. This is performed by the function *CheckExploitability()*. The pseudocode for the *CheckExploitability()* function is shown in Fig. 9. In the function *CheckExploitability()*, first, the required privileges for the exploitation of the input vulnerability or information source are found by calling the function *FormPrivileges()*. The *FormPrivileges()* function makes the given conditions of the input vulnerability or information source specific to the given source and target software application by adding the information (IP address, CPE [20] name, etc.) of the source and target software application to them. The second parameter of the *FormPrivileges()* function is the source (attacking) software application and the third parameter is the target (attacked) software application. For instance, if the condition is an indirect condition and specifies a product type instead of a CPE name on the

```
1: foundPrivileges ← new Set()        ▷ Global variable across the search agent
2: function CheckExploitability(SP, CP, TSA)        ▷ SP can be a vulnerability or information source, CP is the current privilege, TSA is the target software application
3:     reqprgs ← FormPrivileges(SP.preConditions(),
4:         CP.softwareApp, TSA)
5:     if not (reqprgs contains CP) then
6:         return null;
7:     end if
8:     for all rqp ∈ reqprgs do
9:         if not (foundPrivileges contains rqp) then
10:            rqps ←ReadFromSharedMemory(rqp)
11:            if rqps.expanded == false then        ▷ If the privilege is not expanded, then it means that it is not generated by any search agent until now.
12:                return null
13:            end if
14:        end if
15:    end for
16:    return reqprgs
17: end function
```

Fig. 9. Checking exploitability of a vulnerability or an information source by an attacker.

attacker relative location, then the *FormPrivileges()* function makes this condition more specific by attaching the CPE name of the source software application to it. However, this is possible, only if the source application has the same product type specified in the condition.

If the currently traversed privilege, given as parameter, does not exist in the required privileges, then the *CheckExploitability()* function returns null, indicating that the satisfaction conditions of the input vulnerability or information source are unrelated to the currently traversed privilege. A required privilege is satisfied, if and only if it is found in the *global list foundPrivileges* or it has already been written into the virtual shared memory (its expanded status is true). (*foundPrivileges* is a global set used in both algorithms shown in Figs. 7 and 9.) If at least one of the required privileges is not satisfied, then the function *CheckExploitability()* returns null indicating that the input vulnerability or information source can not be used by the attacker now.

If the required privileges returned by the function *CheckExploitability()* is not null, the vulnerability or the information source can be exploited by an attacker. The newly gained privileges after the exploitation are computed from the postconditions of the exploitable vulnerabilities and usable information sources by calling the function *FindGainedPrivileges()* in Fig. 7. The details of the process of computing the newly gained privileges are given in Appendix A.

3.3.3 Updating Partial Attack Graphs

The partial attack graph computed by the search agent is updated by the newly gained privileges, exploited vulnerabilities and used information sources in *UpdateAttackGraph()* function during the distributed search algorithm as shown in Fig. 7. The partial attack graph is updated according to the attack graph structure described in Section 2.2 as follows: If a vulnerability is exploited or an information source is used by an attacker, a vulnerability exploit node or an information source usage node is created using meta-information (IP address, CPE [20] name, etc.) of the target software application and added to the partial attack graph. If there is one required privilege, given as parameter, then an edge from the required privilege to the vulnerability exploit node or information source usage node is added in

```
1: partialAttackGraph ← new AttackGraph() ▷ Global variable across the search agent code
2: procedure UPDATEATTACKGRAPH(SP, REQPS, GPS, TSA) ▷ SP can be a vulnerability or information source, REQPS is the required privileges, GPS is the gained privileges, TSA is the target software application
3:     if SP instanceof Vulnerability then
4:         exp ← CreateVulnerabilityExploitNode(SP, TSA)
5:     else
6:         exp ← CreateInformationSourceUsageNode(SP, TSA)
7:     end if
8:     if REQPS.size() > 1 then
9:         prjc ← new PrivilegeConjunction()
10:        partialAttackGraph.addNode(prjc)
11:        for all reqp ∈ REQPS do
12:            partialAttackGraph.addEdge(reqp, prjc)
13:        end for
14:        partialAttackGraph.addEdge(prjc, exp)
15:    else
16:        if REQPS.size() == 1 then
17:            partialAttackGraph.addEdge(REQPS.get(0), exp)
18:        end if
19:    end if
20:    for all gp ∈ GPS do
21:        partialAttackGraph.addEdge(exp, gp)
22:    end for
23: end procedure
```

Fig. 10. Updating the partial attack graph for each search agent after exploitation of a vulnerability or usage of an information source.

the partial attack graph for the search agent. Otherwise, a privilege conjunction is created and added to the partial attack graph. An edge from each of the required privileges to the privilege conjunction node is added in the partial attack graph. Also, an edge from the privilege conjunction node to the vulnerability exploit node or information source usage node is added in the partial attack graph. The pseudo-code for the function *UpdateAttackGraph()* is shown in Fig. 10.

After all search agents finish computing their partial attack graphs, they send the partial attack graphs to the designated leader agent and the leader agent computes the final attack graph by merging the partial attack graphs. The details of the process of merging the partial attack graphs by the leader agent are provided in Appendix B.

3.4 Complexity Analysis

The complexity of the distributed search algorithm, shown in Fig. 7, is considered in terms of the message transfer and execution time complexity. The maximum number of messages transferred among the distributed search agents is dominated by the number of the memory page faults encountered by them, since we try to provide for comparable workload for each search agent during the execution of the search algorithm.

The number of possible privileges that can be obtained on a software application is constant. If virtual shared memory abstraction is used without any specific memory initialization scheme (e.g. reachability hyper-graph partitioning), then the maximum number of memory page faults encountered by the search agents is $O(E)$, where E denotes the number of edges among directly reachable software applications. This is because, each edge among directly reachable software applications is traversed maximum C times totally by all agents during attack graph building, where C denotes the maximum total number of privileges related with a software application and is constant. Actually in the computation of the attack graph, there can be at most $O(E * C)$ privileges directly obtainable from a specific privilege. Since each privilege has its expansion status stored in the virtual shared memory, each of the edges between a privilege and its directly obtainable privileges is traversed exactly once. Memory page transfers can occur, when going from one privilege to each of its directly obtainable privileges during the attack graph building. C is constant, so the maximum number of memory page faults encountered by the search agents is $O(E)$. In the worst-case, this complexity is $O(N^2)$, where N denotes the number of network interfaces/hosts containing software applications.

If virtual shared memory abstraction is used with the reachability hyper-graph partitioning-based memory initialization, then the maximum number of memory page faults encountered by the search agents is $O(N * H)$, where H is the number of cluttered hyper-edges (whose contents are distributed into more than one search agent) after reachability hyper-graph partitioning, N denotes the number of network interfaces/hosts. Each privilege is processed only once and can have at most H containing hyper-edges, which can be stored in the memory of other search agents. Memory page transfers can occur, when going from one privilege to each of its directly obtainable privileges which are related with the software applications stored in the cluttered hyper-edges. It should be noted that this complexity reasoning is made under the assumption of comparable workload for each search agent during the execution of the search algorithm.

According to [19], the worst-case number of messages for locating the owner of a single page K times is $O(P + K * log(P))$, where P is the number of processors (the search agents in our case). Therefore, the worst-case complexity of the number of messages transferred among the search agents in our algorithm is $O(P + N * H * log(P))$, where H denotes the number of cluttered hyper-edges obtained after reachability hyper-graph partitioning as described above.

When considering the execution time complexity of the distributed search algorithm, since a virtual shared memory abstraction is applied over distributed search agents, each edge in the resulting attack graph is traversed only once as in popular serial depth-first or breadth-first search-based algorithms running on one processor, by controlling the memory coherent expansion status for privileges. The worst-case time complexity of popular serial depth-first or breadth-first search-based attack graph building algorithms is $O(N^2)$, where N denotes the number of network interfaces/hosts (when considering constant maximum number of privileges per network interface/host). If we assume the number of search agents is P, then the worst-case time complexity of our distributed, parallel depth-first search based algorithm is $O(N^2/P)$, since it also traverses each edge in the resulting attack graph once.

The execution time complexity of the merging algorithm used to merge partial attack graphs (generated by the search agents) by the leader search agent and shown in Fig. 14 is $O(P * N * log(N))$, if we assume that getting a value from a hash map takes $O(log(N))$ time.

As a result, the time complexity of the overall distributed attack graph generation algorithm is:

$$O((N^2/P) + P + N * H * logP + P * N * log(N)). \quad (1)$$

If we can use an adequate number of search agents in order to make P get values comparable to $log(N)$, then the time

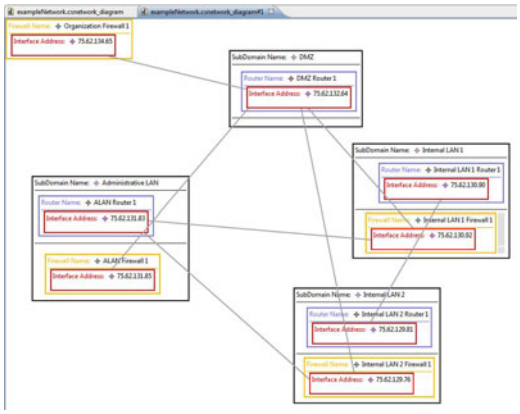

Fig. 11. An example target network for attack graph generation.

complexity of the overall algorithm becomes $O(N^2/log(N))$. For example, if the number of network interfaces/hosts containing software applications (N) is on the order of 5,000, we may use 13 search agents to achieve this complexity.

4 Experiments

The experiments used to evaluate the performance of the proposed attack graph building mechanism are performed using the open-source multi-agent platform JIAC V [23], developed at DAI-Labor in TU Berlin. Each search agent in JIAC V multi-agent environment can be considered to execute in its own thread. JIAC V agents communicate via TCP sockets in the employed experimental configuration. The original network (network domain) to be tested (for which the attack graph is to be generated) is given in Fig. 11. It consists of four sub-domains: DMZ, Administrative LAN, Internal LAN 1 and Internal LAN 2. There is a malicious external domain, consisting of a single sub-domain, connected to the domain of the example network. Each host found in the Internal LANs and Administrative LAN in the target network domain contains the following applications: MS Windows 7 gold, MS Outlook 2007, MS Office 2010, MS Internet Explorer 10. Two of the web servers found in DMZ in the target network domain contain Apache HTTP Server 2.4.3, the other two contain MS IIS Server 6.0. One of the database servers in DMZ contains Oracle9i Database Server 9.0.1.2, the other contains MySQL Database Server 5.1. Mail server in DMZ contains MS Exchange Server 2010. The vulnerabilities of each of these applications (with their pre and postconditions) are stored on a specific vulnerability database managed in our system.

The aim of these experiments is to compare the performance of the distributed search algorithm, proposed as the core attack graph building mechanism in this paper, with the performance of the serial search-based attack graph building algorithms, which has worst-case time complexity $O(N^2)$ where N denotes the number of network interfaces/hosts in the target network. To the best of our knowledge, no attack graph building algorithm proposed so far in the literature attains a worst-case time complexity better than $O(N^2)$. Also, there has been no distributed attack graph building algorithm proposed so far. By increasing the count of the network hosts (except routers and firewalls) in the Internal LANs in the target network domain and in the external malicious network domain in each experiment iteration, the performance of the serial (depth-first) search-based and our distributed attack graph building algorithm are compared. Proposed distributed attack graph building algorithm is executed with two to four search agents running on a computer with Intel X7550 quad-core processor. The configurations of the firewalls are held constant across the experiment iterations. The running times of both algorithms with respect to the total number of hosts in the resulting network (target network domain combined with external malicious network domain) are shown in Table 1.

In Table 1, the privilege count in the resulting attack graph for each target network is also shown. A privilege is a type of attack graph node and represents a specific right that is gained by an attacker on a software application on a network host as described in Section 2.2. Therefore, the number of privilege nodes in the resulting attack graphs represent their size roughly. Since attack graph building algorithms are mostly based on the traversal of privileges and aim to eliminate duplicate expansion of privileges as much as possible, the number of privilege nodes in the

TABLE 1
Running Times of Serial DFS and the Proposed Distributed, Parallel Algorithm

Network Size (Host Count)	Attack Graph Size (Privilege Count)	Running Time (sec.)			
		Serial DFS	Dist. (2 Agents)	Dist. (3 Agents)	Dist. (4 Agents)
18	352	1.85	5.95	5.93	5.99
27	680	2.68	6.67	6.83	6.85
36	1,008	3.82	7.71	7.30	7.99
54	1,664	5.19	9.41	9.25	10.47
90	2,976	10.72	13.09	12.24	12.79
126	4,288	19.59	21.11	18.25	16.79
162	5,600	32.35	29.73	27.36	25.63
198	6,912	50.49	38.69	36.48	34.65
243	8,552	78.29	62.65	48.85	46.66
288	10,192	124.23	86.32	79.45	68.55
333	11,832	171.35	122.63	105.41	88.68
387	13,438	231.86	167.87	122.28	100.22
441	16,754	295.57	200.52	151.82	121.71
495	19,936	385.36	261.82	198.21	145.25

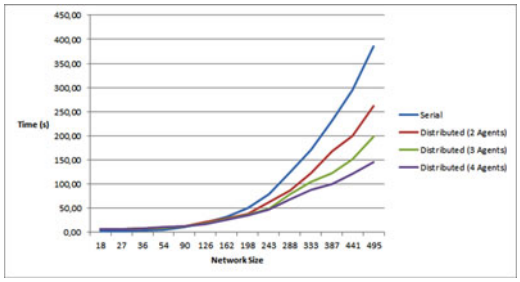

Fig. 12. Performance comparison of the serial and proposed distributed attack graph building algorithms in terms of execution time.

resulting attack graphs also serves as an indication of the workload that must be performed by the attack graph building algorithms. By indicating the workload for each experiment, we compare the performance of the serial algorithm with our distributed algorithm in the face of increasing workload. In each of these experiments, there is only one initial attacker privilege that is supposed to be gained on one of the malicious nodes. This initial privilege triggers the attack graph building process. Page size for the virtual shared memory is set so that each page can contain all the privileges associated with 16 network hosts. Additionally, for each target network, the average number of vulnerabilities per network host is almost 10.

When the benefit (time decrease) gained from parallelism (increasing the number of search agents executing the attack graph building process) exceeds the time increase caused by the message transfers among the agents, proposed distributed attack graph building algorithm becomes desirable. For instance, starting from the experiment where the target network has 162 hosts (resulting attack graph has 5,600 privileges), the proposed algorithm running with two-agent configuration gives better performance in terms of time with respect to the serial graph search-based attack graph building algorithms. Similar condition is true for the execution of the proposed algorithm with three and four agent-configurations starting from the experiment where the target network has 126 hosts. The performance gain by the distributed computation of attack graphs goes up to 40 percent or more using only four agents for networks of at least 250 hosts. Fig. 12 further depicts the performance gains obtained by the distributed algorithm in terms of the execution time.

We can also conclude from the experiment results that while the number of network hosts increases, the execution time for the four agent configuration becomes much better than three agent configuration and the execution time for the three agent configuration becomes much better than two agent configuration. Namely, while the number of network hosts increases, the performance gain obtained with a specific number of agents compared to the performance gain obtained with less number of agents increases. The benefit of the distributed algorithm becomes more obvious.

5 Conclusion and Future Work

In this paper, a distributed search-based algorithm is introduced for full attack graph generation on a multi-agent platform, over which a virtual shared memory abstraction is proposed. The experiment results demonstrate that distributed computation of attack graphs can be utilized to overcome the state space explosion problem occurring in the attack graph building process when the number of hosts and vulnerabilities on the target network grows. It can also be utilized in real-time attack scenario detection and prediction using intrusion alerts obtained from various sensors scattered across a network. In this case, a distributed search agent builds its corresponding partial attack graph according to the intrusion alerts it receives from a specific part of the network. It can also communicate with other search agents to form the links among the partial attack graphs of each other.

One of the possible future works may include the assessment of the advantages that can be gained by allowing duplicate privilege expansion at some points in the attack graph building process to refrain from additional memory page transfers among search agents. Therefore, at certain points, the principle of preserving coherence across search agents' memories can be relaxed. One other future work can be to develop methods using map-reduce-based approaches in order to solve the attack graph core building problem. Additionally, as a future work, some values may be assigned to the vulnerability exploits and information source usage vertices in an attack graph at run-time to eliminate the computation of attack paths that are less likely to be utilized by a potential attacker because of difficulty of utilization, less damage level introduced to the target network, etc. In this case, instead of all possible attack paths to the given target privileges, a number of advantageous paths may be computed. This process of advantageous attack paths computation can benefit from heuristic search algorithms aiming to find multiple shortest paths among attack graph vertices. For this purpose, network security domain-specific heuristics to direct such search algorithms can be developed.

Appendix A
Computing Gained Privileges

When computing the newly gained privileges, *relative locations for the postconditions* of the exploitable vulnerabilities and usable information sources are accounted. If the relative location for a postcondition refers to the backend application of the parameter target software application, then the backend software applications used by the target software application are found and corresponding gained privileges on the backend software applications are computed using the postcondition. By this way, the attacker can gain privileges on a (backend) software application by using another application's vulnerability instead of using a vulnerability of the (backend) software application. Newly gained privileges are pushed to the main search stack used by the search agent, if they have not been already expanded. The pseudo-code for the function *FindGainedPrivileges()* is shown in Fig. 13.

Appendix B
Merging Partial Attack Graphs

After all search agents finish computing their partial attack graphs, the partial attack graphs are merged by one of the search agents, designated as the leader search agent. The other agents send their computed partial attack graphs to the leader agent and the leader agent performs the

```
1: function FindGainedPrivileges(SP, CP, TSA)  ▷ SP can be a vulnerability or information source, CP is the current privilege, TSA is the target software application
2:     BA ← RelativeLocation.BackendApplication
           ▷ Account for relative location value of backend application for postconditions
3:     for all psc ∈ SP.postConditions() do
4:         if psc.ExistsIn == BA then
5:             for all bsa ∈ TSA.backendSoftwareApps() do
6:                 gprgs.addAll(FormPrivileges(psc,
7:                     CP.softwareApp, bsa))
8:             end for
9:         else
10:            gprgs ← FormPrivileges(psc,
11:                CP.softwareApp, TSA)
12:        end if
13:    end for
14:    return gprgs
15: end function
```

Fig. 13. Finding the privileges gained by an attacker after exploiting a vulnerability or an information source.

procedure, shown in Fig. 14, to merge them and form a single attack graph.

Attack graph merge algorithm starts by merging the same privileges existing in different partial attack graphs. If a privilege exists in more than one partial attack graph, only one instance of it can contain descendant nodes. Because, we expand each privilege only once during the distributed search algorithm via holding its expansion status in virtual shared memory. Therefore, if there is an instance of a privilege in a partial attack graph that contains descendant nodes, this instance should exist in the resulting attack graph computed by the leader search agent. Other instances are removed from the attack graph. In edges of other instances (existing privileges) are updated to target the instance containing descendant nodes by calling the *UpdateInEdgesOfExistingNode()* function. This function updates the target nodes of the in edges of its first parameter with the value of the second parameter.

After eliminating duplicate privileges in the resulting attack graph, merging algorithm eliminates duplicate vulnerability exploit and information source usage nodes (exploit nodes) contained by the resulting attack graph. If there are same instances of an exploit node in the resulting attack graph, their out edges are merged (combined). If we find an exploit node named *exploitNode* during the traversal of the *exploit node list of the resulting attack graph*, for which the same node has already existed, we first find the out neighbour nodes of the existing exploit node. We add an edge from the *exploitNode* to each out neighbour node. Then, we update the target nodes of the in edges of the already existing exploit node with the value of *exploitNode* by calling the *UpdateInEdgesOfExistingNode()* function. We remove the already existing exploit node from the resulting attack graph. As a summary, we collect the in/out neighbours of the same exploit nodes, make one of these exploit nodes refer to all of these in/out neighbours and remove the other exploit node from the attack graph.

```
1: function MergePartialAttackGraphs(PGS)  ▷ PGS is the partial attack graphs generated by the search agents
2:     if PGS.size() == 0 then
3:         return new AttackGraph()
4:     end if
5:     ag ← PGS.removeFirst()
   ▷ Update privileges for the attack graph
6:     phm ← FormHashMap(ag.privileges())  ▷ Hash of privileges mapped by their identifiers
7:     for all prag ∈ PGS do
8:         for all p ∈ prag.privileges() do
9:             esp ← phm.get(p.id())
10:            if esp! = null then
11:                if (p.outEdges.size() > 0 then
12:                    if esp.outEdges.size() == 0) then
           ▷ Remove the existing privilege from the attack graph and add the privilege and its subtree in place of existing privilege
13:                        ag.addNodeWithItsSubTree(p)
14:                        UpdateInEdgesOfExistingNode(esp, p)
15:                        ag.removeNode(esp)
16:                        phm.put(p.id(), p)
17:                    end if
18:                end if
19:            else
20:                ag.addNodeWithItsSubTree(p)
21:                phm.put(p.id(), p)
22:            end if
23:        end for
24:    end for
   ▷ Remove duplicate vulnerability exploit and information source usage nodes for the attack graph
25:    exhm ← new HashMap()
26:    expl ← ag.vulnerabilityExploits
27:    expl.addAll(ag.informationSourceUsages)
28:    for all expn ∈ expl do
29:        esxpn ← exhm.get(expn.id())
30:        if esxpn! = null then
31:            for all onn ∈ esxpn.outNeighbourNodes() do
32:                ag.addEdge(expn, onn)
33:            end for
34:            UpdateInEdgesOfExistingNode(esxpn, expn)
35:            ag.removeNode(esxpn)
36:        end if
37:        ehm.put(expn.id(), expn)
38:    end for
39:    return ag
40: end function
```

Fig. 14. Merging the partial attack graphs by the leader search agent.

ACKNOWLEDGMENTS

Parts of this work were supported by funding from the German Federal Minister of Education and Research under grant 01IS13003. The responsibility of this publication lies solely with the authors. K. Kaynar is the corresponding author.

REFERENCES

[1] (2014, Nov.) Common weakness enumeration. [Online]. Available: http://cwe.mitre.org/
[2] (2014, Nov.) National vulnerability database. [Online]. Available: http://nvd.nist.gov/
[3] S. J. Templeton and K. Levitt, "A requires/provides model for computer attacks," in *Proc. Workshop New Secur. Paradigms*, Jan. 2000, vol. 50, no. 1, pp. 31–38.
[4] P. Ning and D. Xu, "Learning attack strategies from intrusion alerts," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, Jan. 2003, vol. 50, no. 1, pp. 200–209.
[5] I. Kotenko and M. Stepashkin, "Attack graph based evaluation of network security," in *Proc. 10th IFIP TC-6 TC-11 Int. Conf. Commun. Multimedia Secur.*, 2006, pp. 216–227.
[6] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, Jan. 2000, vol. 50, no. 1, pp. 36–44.
[7] O. Sheyner, J. Haines, S. Ja, R. Lippmann, and J. M. Wing, "Automated generation and analysis of attack graphs," in *Proc. IEEE Symp. Secur. Privacy*, Jan. 2002, vol. 50, no. 1, pp. 36–44.
[8] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proc. 13th ACM Conf. Comput. Commun. Secur.*, Jan. 2006, vol. 50, no. 1, pp. 336–345.
[9] P. Ammann, D. Wijesekara, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proc. 9th ACM Conf. Comput. Commun. Secur.*, Jan. 2002, vol. 50, no. 1, pp. 217–224.

[10] P. Ammann, J. Pamula, J. Street, and R. Ritchey, "A host-based approach to network attack chaining analysis," in *Proc. 21st Annu. Comput. Secur. Appl. Conf.*, Jan. 2005, vol. 50, no. 1, pp. 72–84.
[11] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *Proc. 22nd Annu. Comput. Secur. Appl. Conf.*, Jan. 2006, vol. 50, no. 1, pp. 121–130.
[12] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer, "Modeling modern network attacks and countermeasures using attack graphs," in *Proc. Annu. Comput. Secur. Appl. Conf.*, Jan. 2009, vol. 50, no. 1, pp. 117–126.
[13] S. Bhattacharya, S. Malhotra, and S. K. Ghsoh, "A scalable representation towards attack graph generation," in *Proc. 1st Int. Conf. Inform. Technol.*, May 2008, vol. 50, no. 1, pp. 1–4.
[14] L. Zhang, J.-B. Hu, and Z. Chen, "A probability-based approach to attack graphs generation," in *Proc. 2nd Int. Symp. Electron. Commerce Secur.*, May 2009, vol. 50, no. 1, pp. 343–347.
[15] J. Ma, Y. Wang, J. Sun, and X. Hu, "A scalable, bidirectional-based search strategy to generate attack graphs," in *Proc. Int. Conf. Comput. Inf. Technol.*, vol. 0, pp. 2976–2981, 2010.
[16] N. Ghosh and S. K. Ghosh, "A planner-based approach to generate and analyze minimal attack graph," *Appl. Intell.*, vol. 36, no. 2, pp. 369–390, Mar. 2012.
[17] Y. Zhang, F. Chen, and J. Su, "A scalable approach to full attack graphs generation," *Engineering Secure Software and Systems*. Berlin, Germany: Springer, 2009, pp. 150–163.
[18] (2014, Nov.) Common attack pattern enumeration and classification. [Online]. Available: http://capec.mitre.org/
[19] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," *ACM Trans. Comput. Syst.*, vol. 7, no. 4, pp. 321–359, Nov. 1989.
[20] (2014, Nov.) Common platform enumeration. [Online]. Available: http://cpe.mitre.org/
[21] (2014, Nov.) Common vulnerabilities and exposures. [Online]. Available: http://cve.mitre.org/
[22] G. Karypis and V. Kumar, "Multilevel k-way hypergraph partitioning," in *Proc. 36th Annu. ACM/IEEE Des. Autom. Conf.*, 1999, pp. 343–348.
[23] B. Hirsch, T. Konnerth, and A. Heler, " Merging agents and services the jiac agent platform," in *Multi-Agent Programming*, A. El Fallah Seghrouchni, J. Dix, M. Dastani, and R. H. Bordini, Eds. New York, NY, USA: Springer, 2009, pp. 159–185.

Kerem Kaynar received the Bsc and Msc degrees in the Computer Engineering Department of Middle East Technical University, Ankara, Turkey in 2004 and 2009, respectively. He has seven years of working experience in the field of cybersecurity in Turkey. He has been working as a research assistant for more than two years in TU Berlin, Germany, where he has developed software in network security projects. He is currently working at German Turkish Advanced Research Center. His current research interests include network security testing, malware detection, and vulnerability discovery.

Fikret Sivrikaya received the PhD degree in computer science from the Rensselaer Polytechnic Institute, NY in 2007. He is currently the director of Network and Mobility group at DAI-Labor, TU Berlin, where he is responsible for coordinating project and research activities in the area of telecommunications, and has participated in and coordinated many national and international ICT projects. Before joining TU Berlin, he was a research assistant at RPI and earlier worked in the ICT industry in Istanbul, Turkey between 1999 and 2002. His current research interests mainly cover wireless communication protocols, medium access control and routing issues in multi-hop ad-hoc networks, distributed algorithms and optimization.