QUANTITATIVE RISK ASSESSMENT UNDER MULTI-CONTEXT ENVIRONMENTS

by

SU ZHANG

B.S., Northeast Normal University, China, 2008

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences

College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2014

Abstract

If you cannot measure it, you cannot improve it. Quantifying security with metrics is important not only because we want to have a scoring system to track our efforts in hardening cyber environments, but also because current labor resources cannot administrate the exponentially enlarged network without a feasible risk prioritization methodology. Unlike height, weight or temperature, risk from vulnerabilities is sophisticated to assess and the assessment is heavily context-dependent.

Existing vulnerability assessment methodologies (e.g. CVSS scoring system, etc) mainly focus on the evaluation over intrinsic risk of individual vulnerabilities without taking their contexts into consideration. Vulnerability assessment over network usually output one aggregated metric indicating the security level of each host. However, none of these work captures the severity change of each individual vulnerabilities under different contexts.

I have captured a number of such contexts for vulnerability assessment. For example, the correlation of vulnerabilities belonging to the same application should be considered while aggregating their risk scores. At system level, a vulnerability detected on a highly depended library code should be assigned with a higher risk metric than a vulnerability on a rarely used client side application, even when the two have the same intrinsic risk. Similarly at cloud environment, vulnerabilities with higher prevalences deserve more attention. Besides, zero-day vulnerabilities are largely utilized by attackers therefore should not be ignored while assessing the risks. Historical vulnerability information at application level can be used to predict underground risks. To assess vulnerability with a higher accuracy, feasibility, scalability and efficiency, I developed a systematic vulnerability assessment approach under each of these contexts.

QUANTITATIVE RISK ASSESSMENT UNDER MULTI-CONTEXT ENVIRONMENTS

by

Su Zhang

B.S., Northeast Normal University, China, 2008

A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences

College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2014

Approved by:

Major Professor

Xinming Ou

Copyright

Abstract

If you cannot measure it, you cannot improve it. Quantifying security with metrics is important not only because we want to have a scoring system to track our efforts in hardening cyber environments, but also because current labor resources cannot administrate the exponentially enlarged network without a feasible risk prioritization methodology. Unlike height, weight or temperature, risk from vulnerabilities is sophisticated to assess and the assessment is heavily context-dependent.

Existing vulnerability assessment methodologies (e.g. CVSS scoring system, etc) mainly focus on the evaluation over intrinsic risk of individual vulnerabilities without taking their contexts into consideration. Vulnerability assessment over network usually output one aggregated metric indicating the security level of each host. However, none of these work captures the severity change of each individual vulnerabilities under different contexts.

I have captured a number of such contexts for vulnerability assessment. For example, the correlation of vulnerabilities belonging to the same application should be considered while aggregating their risk scores. At system level, a vulnerability detected on a highly depended library code should be assigned with a higher risk metric than a vulnerability on a rarely used client side application, even when the two have the same intrinsic risk. Similarly at cloud environment, vulnerabilities with higher prevalences deserve more attention. Besides, zero-day vulnerabilities are largely utilized by attackers therefore should not be ignored while assessing the risks. Historical vulnerability information at application level can be used to predict underground risks. To assess vulnerability with a higher accuracy, feasibility, scalability and efficiency, I developed a systematic vulnerability assessment approach under each of these contexts.

Table of Contents

List of Figures

List of Algorithms

List of Tables

Acknowledgments

I would like to express my deepest appreciation to my advisor, Dr.Xinming (Simon) Ou. I can benefit from his insightful guidance all of the time. His passion on scientific research always motivates me to take the toughest research challenge. He can pinpoint the root of cause whenever I encountered a bottleneck and show me the right approach to get it through. He is always fully responsible to everyone around him. I feel lucky and honorable of having such an unforgettable experience with him. I would not have reached this point without his continuous encourage and support.

Like I said the past six years is a fruitful and unforgettable pleasant journey. I believe it is just a starting point of our collaboration and I am looking forward to having more work done in the future together with Simon.

I would like to thank everyone in my thesis committee for their participation in this whole process and for their valuable comments and suggestions.

I highly appreciate each faculty member who has instructed and encouraged me over the past six years.

Lastly, for my fellow student friends, I enjoyed the time we spent together and we will for sure keep each other in touch.

This work was supported by the Air Force Office of Scientific Research award FA9550-12-1-0106 and U.S. National Science Foundation awards 0716665, 0954138, and 1018703. Any opinions, findings and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the above agencies.

Chapter 1

Introduction

1.1 Vulnerability Assessment

Vulnerability assessment is a critical task for various roles in IT industry. System administrators need network security metrics in order to make hardening plans accordingly. Cloud providers need to evaluate how much more risk has been brought by the new paradigm compared to the traditional network environment. Image publishers need to use system-wide attack surface metrics to track the risk level of their images. Image user also want to choose reliable images by using risk metrics in order to minimize potential risks. Each of these contexts needs a vulnerability assessment approach to assist stake holders to make appropriate decisions in an automatic manner. However, measuring security is a sophisticated work. Unlike height, weight or temperature, security can not be measured directly through any measuring instrument. Vulnerability severity can be determined by various factors. CVSS scoring system introduces a number of such factors like access complexity: how much effort an attacker needs to make in order to access and exploit the vulnerability; authentication: if the attacker needs to be authenticated and number of times of being authenticated before he can exploit the vulnerability; Access vector: if the vulnerability could be exploited locally, within a LAN or remotely; and impact factors on confidentiality, integrity and availability, etc. Other factors like the availability of exploit have been proposed as well. However, it is

difficult to assign numeric values to those factors and the normalization of these values is another challenging task. Moreover, these intrinsic metrics are not enough for security measurement under different contexts. While evaluating security level of a complicated network environment, similarity among vulnerabilities should be taken into consideration. Package dependency also needs to be taken into account while accessing the system wide attack surface of a given vulnerability. Cloud environment should also be evaluated differently because of its more homogeneous and stable settings compared with traditional network environment. Last but not least, unknown security holes need to be evaluated in complementary with known vulnerabilities in order to provide a comprehensive risk assessment result under various cyber contexts.

As the topology and configuration of networks are getting complicated, understanding the overall security situation of the whole network is becoming a challenging task. Reading security reports for each host is infeasible for system administrator to understand the overall security situation of the network. Existing work like attack graph and risk assessment algorithm on top of it has been developed in order to tackle this issue. However, the visualization and dependencies among attack paths need to be well addressed. Attack graph for a mid-sized network can be difficult to comprehend. System administrator may not be able to capture the key part among a large set of edges and nodes. Also the dependency among attack paths need to be captured in order to provide a more accurate risk assessment result.

While evaluating risks at a microscope (e.g.system level), a vulnerability scanning tool can only output known vulnerabilities existed on the system. However, only the intrinsic security metrics (e.g.CVSS scores) can be automatically retrieved. These metrics do not necessarily reflect the risk level of the vulnerability under this specific context. The exploitability of each vulnerability is highly correlated to its exposability, which is mostly based on its dependency relations over the whole system. Therefore, it is necessary to construct a set of metrics with the consideration of the package dependency in order to assist

the image publishers to reduce the system based attack surface accordingly. Customers could also utilize these metrics to choose reliable images.

Compared to traditional network, cloud computing especially infrastructure as a service cloud offers more flexible service by providing a number of configurable VMs. With the infrastructure empowered by virtualization, the attack surface has also been largely remodeled from the traditional network environment. As identified by [10, 13, 34, 54], there is a number of cloud specific security issues. Other than that, cloud platform usually has more stable settings than traditional networks. However, the system hardening usually depends on the user but not the provider. Given the fact that most of the cloud customers are individuals or small web service providers, a considerable number of hosts on cloud are exposed to attackers and therefore form a large attack surface over the platform. Meanwhile, a plenty of users are using the images provided by third party providers, therefore attackers could first do a penetration test over the image to identify a number of easily exploitable vulnerabilities and then compromise a large number of hosts by launching similar attacks repeatedly in the cloud. Therefore, while evaluating risk under this context, the lowered cost effective ratio from attacker's side needs to be captured by cloud stakeholders.

Most existing work on risk assessment is about known vulnerabilities, but the risk from underground security holes should not be ignored because a considerable amount of cyber attacks originated from these unknown security holes. Different applications suggest different risk levels even when there is no known vulnerabilities detected on all of them. A creditability like system is needed in order to let administrators or customer to estimate the potential risk at application level. Along with the risk assessment approaches on known vulnerabilities, a more completed security evaluation approach with the consideration of unknown security holes is urgently needed.

1.2 Related Work

1.2.1 Network Security Evaluation

Attack graph technique has become a common tool for network security evaluation and it has been developed for the purpose of automatically identifying multi-stage attack paths in an enterprise network [8, 21, 22, 24, 38, 42–44, 49, 51, 52, 72–74, 80, 87, 89, 94, 96]. It has been observed that attack graphs are often too large to be easily understood by human observers, such as system administrators. In order to reduce the complexity of attack graphs to make them more accessible to use by system administrators, various approaches have been proposed to improve the visualization through abstraction, data reduction, and user interaction [37, 40, 51, 52, 69, 103]. However, not much work has been done to study the effect of attack graph complexity on quantitative security assessment approaches based on attack graphs. My study found that complexity caused by repetitive information commonly found in attack graphs not only increases the difficulty for the system administrator in digesting the information provided by the graph, but also distorts the risk picture by unrealistically casting the attack success likelihood for some privileges under probability-based security assessment. My approach [110, 111] show that such distortion can be avoided by abstracting the input to the attack-graph generator, *i.e.*, the network model, so that such redundancy is removed a priori. By performing abstraction directly on the network model, the attack graph result can also be rendered on a higher level of system description which is easier to grasp by a human user.

Quantitative security assessment methods based on attack graphs have been proposed to indicate the severity levels of various vulnerabilities [30, 35, 70, 88, 92, 99–101]. Such methods typically utilize the dependency relations represented in an attack graph to aggregate individual vulnerability metrics to reflect their cumulative effects on an enterprise network. However, not all dependency relations are explicitly presented in an attack graph, particularly the similarities among large numbers of attack paths leading to the same privilege.

Not accounting for the existence of this dependency on a large scale will significantly skew the analysis results. One method of dealing with such hidden dependency is to introduce additional nodes and arcs in the graph to model them, but this will make the visualization problem even more severe. I provide a method based on model abstraction to remove the redundancy, and thus the hidden dependency resulted from it, so that it is no longer a problem for realistic risk assessment.

The size of enterprise networks could make vulnerability scanning prohibitively expensive [105]. My abstraction technique provides a possible angle to address this problem. Prioritization can be applied based on the abstract model for identifying scanning which host can potentially provide critical information on the system's security. For example, if a host in the same abstract group has already been scanned, scanning one more host in the group may not provide the most useful information about the system's security vulnerabilities.

1.2.2 Software Dependency Security Evaluation

Risks from package dependency have been well researched [6, 18, 26, 46, 66, 77, 81, 106, 117]. Neuhaus et al. [66] evaluate risk per Red Hat package based on historical security vulnerabilities and package dependencies. But they do not evaluate attack surface exposed by individual vulnerabilities. Besides, they only measure outgoing risk but not incoming risk for each package. Raemaekers et al. [81] explore the risk from third party applications. Instead of measuring attack surface from individual known vulnerabilities, they focus on if a referenced package is well scrutinized and the prevalence of usage per package. A set of work [6, 46, 77, 106, 117] study the importance of component level dependency when assessing software quality but no concrete security metric has been proposed. Chowdhury et al. [18] evaluate risk from source code (class) level of dependency (e.g. complexity, coupling, and cohesion). However, their work is about inferring unknown vulnerabilities rather than evaluate attack surface for known vulnerabilities.

A number of work study risks from Java applications [25, 32, 33, 58, 65, 78, 79]. Nasiri et al. [65] evaluate the attack surface from J2EE and .Net platform by quantitatively comparing their CVSS scores directly, but no package dependency is considered during the evaluation. Drake et al. [25] evaluate JRE memory corruption attack surface from engineering point of view, but they do not provide quantitative measurement of the attack surface. Gong et al. [33] retrospect the evolution of security mechanism on Java in the past ten years at high level. Both Pérez et al. [79] and Goichon et al. [32] propose vulnerability detection approaches after scanning Java source code. Marouf [58] classifies vulnerabilities specific to Java and proposes possible countermeasures against these threats. Similarly, Parrend et al.[78] classify Java vulnerability at component level rather than source code level.

Work regarding attack surface evaluation have been conducted by researchers [17, 36, 39, 56, 57, 67, 67, 98]. Neuhaus et al. [67] rank vulnerable components in Firefox based on historical detected vulnerabilities. Similar to my approach, they evaluate risk at component level. However, they consider these components as independent units rather than inter-depended nodes. My approach also captures incoming and outgoing risks at different granularities, which has not been done by other related work.

The definition of attack surface is also adapted in industry. Similar to [56], which evaluates attack surface over Linux systems, Microsoft attack surface[1] focuses on Windows by enlisting a number of threats based on the configuration of a given system. However, none of these takes package dependency into consideration while measuring system attack surface.

1.2.3 Cloud Computing Security Evaluation

Security issues regarding the public images in Amazon EC2 have been studied [10, 13, 108]. Sensitive information (e.g., SSH/SSL key and source code) leak has been detected by Bugiel et al. [13]. They also suggested several solutions for various cloud specific threats. More

[1] http://www.microsoft.com/en-us/download/details.aspx?id=24487

comprehensive experiments over EC2 have been conducted by Balduzzi et al. [10]. They scanned a larger number of public AMIs and found more security issues like software vulnerabilities and malwares. However, neither of them evaluate potential threat from prevalent known software vulnerabilities, which I believe is the most straightforward and efficient way for attackers to intrude an IaaS cloud. Patching frameworks have been proposed for cloud [53, 114], but they also do not assess the threat of exploiting prevalent vulnerabilities.

Game theory has been used for modeling attacks and defenses [15, 16, 45, 47, 82, 83, 86, 107]. Activities (exploiting and hardening) between attackers and defenders perfectly conform to a 2-player game. Yan et al. [107] model a game between DDoS attackers and defenders. Cavusoglu et al. [15, 16] model the cost of patching and intrusion for both application vendors and software users. They claim that the optimal patch plan can be made once the vendors and enterprise software users can agree on the date of patch release and deployment. However, this assumption is hard to achieve under cloud situation since prevalent applications usually have a large number of users, and software vendors usually do not coordinate with every customer regarding the patch release and deployment. Jormakka et al. [45] model information warfare with gaming theory. Their model analyzes what decision should be made for both parties in order to obtain optimal expected payoff under four different information warfare scenarios. Rajbhandari et al. [82] propose a game theory based approach in helping security professionals to prioritize countermeasures based on security incident evaluations. Khirwadkar [47] constructs a game theoretic model between attackers and defenders by using Fictitious-Play approach in order to make sure the two parties are not under complete information environment. Game theory based analysis regarding network security is surveyed in [86].

Another line of work is to visualize economic incentive from defender's perspective [9, 11, 19, 27–29, 84, 85, 93], which helps the defender appropriately allocate resources to security-related tasks. Studer et al. [93] evaluate DDoS attacks from both technical and economic points of view, and provide evaluation on monetary loss due to these attacks

in addition to the economic appraisal by [19]. Bilge et al. [11] evaluate threats from 0-day vulnerabilities. They find that by average a 0-day vulnerability has already been in existence for ten months. Moreover, around 42% of vulnerabilities can be exploited by a large volume of attacks within one month from its original release, which suggests that exploits can be obtained by attackers easily. Exploit tools like Blackhole [2] can be utilized by attackers in a straightforward manner. Consequently, launching 1-day exploit [71] to unpatched systems is much easier than 0-day exploits as attackers do not have to write and test exploits by themselves. Dacey [20] point out that about 95% of exploits are rooted from unpatched systems. Forbath et al. [27] evaluate patch management cost for both Windows and open source software. Frei et al. [29] visualize time lengths between vulnerability disclosure date, patch date, and exploit date, and believe these time periods represent the current status of security industry. However, they do not consider the date of patch deployment, which is captured in my model to evaluate the dynamic threat of individual platforms or systems. Richardson et al. [85] conduct a survey indicating that 62.3% respondents apply patch after a security incident happens. A report by Mellberg [64] indicates that only 59% of small companies ($\leq$ \$50M revenue) have patch management which conforms to the number investigated by Richardson et al. [85]. This motivates my modeling study [112] since a large number of IaaS users are individuals and small companies [3].

1.2.4 Zero-day Risk Evaluation

Alhazmi and Malaiya [7] have addressed the problem of building models for predicting the number of vulnerabilities that will appear in the future. They targeted operating systems instead of applications. The Alhazmi-Malaiya Logistic model works well for fitting existing data, when evaluated in terms of average error (AE) and average bias (AB) of number of vulnerabilities over time. However, fitting existing data is a prerequisite of testing models: predictive power is the most important criteria [75] . They did test the predictive accuracy

[2] http://nakedsecurity.sophos.com/exploring-the-blackhole-exploit-kit/

of their models and got satisfactory results [75].

Ozment [76] examined the vulnerability discovery models (proposed by Alhazmi Malaiya [7]) and pointed some limitations that make these models inapplicable. One of them is that there is not enough information included in a government supported vulnerability database (*e.g.* National Vulnerability Database). This is confirmed by my empirical study.

McQueen *et al.* [61] designed algorithms for estimating the number of zero-day vulnerabilities on each given day. This number can indicate the overall risk level from zero-day vulnerabilities. However, for different applications the risks could be different. My work aimed to construct software-specific prediction models.

Massacci *et al.* [59, 68] compared several existing vulnerability databases based on the type of vulnerability features available in each of them. They mentioned that many important features are not included in most databases. *e.g.* discovery date is hard to find. Even though certain databases (such as OSVDB that as I also studied) claim they include the features, most of the entries are blank. For their Firefox vulnerability database, they employed textual retrieval techniques and took keywords from CVS developer's commit log to get several other features by cross-referencing through CVE ids. They showed that by using two different data sources for doing the same experiment, the results could be quite different due to the high degree of inconsistency in the data available for the research community at the current time. They further tried to confirm the correctness of their database by comparing data from different sources. They used data-mining techniques (based on the database they built) to prioritize the security level of software components for Firefox.

Ingols *et al.* [41] tried to model network attacks and countermeasures using attack graphs. They pointed out the dangers from zero-day attacks and also mentioned the importance of modeling them. There has been a long line of attack-graph works [8, 21, 22, 24, 42, 44, 51, 73, 80, 89] which can potentially benefit from the estimation of the likelihood of zero-day vulnerabilities in specific applications. I created a model [109] by using historical data to predict risk from unknown vulnerabilities. The result from my prediction model can

possibly be plugged into the attack graph based risk assessment algorithms and provide a more comprehensive risk assessment result.

1.3 Original Contributions

This dissertation proposes several quantitative risk assessment approaches under various contexts. Simply put, all of these approaches bridge the gap between standard industrial risk assessment approach (e.g. CVSS scoring system for each vulnerability) and security evaluation under specific contexts. These context aware risk assessment approaches are illustrated as follow:

1. Network risk assessment. My work improved the accuracy and scalability of vulnerability assessment on enterprise networks. By capturing the hidden correlations among similar vulnerabilities (e.g. grouping vulnerabilities on same applications), the attack graphs representing network security status have been downsized with even higher accuracy. By first grouping hosts by topology and then break down them based on configurations, each host in attack graph will then represent a broken down subnet (hosts within it have same reachability and configuration). The cluster-based process provides stakeholder a clearer and preciser attack graph.

2. Package dependency risk assessment. The motivation of this work originates from my 2011 summer internship at VMware. Besides enriched functions, library dependency brings risks to product at the same time. Prioritizing patching plan at vulnerability level cannot simply rely upon metrics such like CVSS base score. Accurate measurement on the package-dependency risk at vulnerability level is of vita importance. I developed a methodology systematically assessing such risk at vulnerability, component, application and system level respectively. Providing stakeholders with a full stack of component dependency based risk metrics.

3. IaaS cloud risk assessment. Known vulnerabilities pose exponential larger attack surfaces on public clouds than on individual machines. Amazon EC2 offers public images which could be run by any user. However, a considerable amount of these images exist known vulnerabilities due to out of date applications. A cloud is usually composed by a number of centralized clusters of servers. As a result, prevalent vulnerabilities on a public cloud open a large attack surface to attackers. Compared with traditional network environment, attackers have a lower cost-effectiveness ratio by repeating same attacks over adjacent hosts. My game theoretic model indicates current public cloud platforms are under greater risky situation than traditional network environment. I construct risk density model to illustrate the enlarged attack surface. Countermeasures against such risk are provided to reduce the attack surface by changing the parameters value in the risk density models.

4. Zero-day risk assessment. This work aims at estimating zero-day risk at application level through historical data. I extensively mined vulnerability data at NVD by using data-mining algorithms provided by a popular data-mining/machine learning toolkit WEKA. I use a number of attributes as predictive features, for instance, CVSS metrics, length between vulnerabilities discovery dates, product name, version and vendor, etc. I want to predict time to next vulnerability (TTNV) for each given application. My experimental results indicate that for most of the applications, the target attribute can hardly be validated. However for certain application, the validation results are promising. This may be because of inaccuracy data created by either vendors (on the security bulletin/advisories) or the vulnerability database maintainers.

The overall contribution of this dissertation is the construction of quantitative risk assessment frameworks under various contexts. It bridges the gap between risk assessment of intrinsic risk of individual vulnerabilities and the risk assessment needed by various roles in IT industry under specific contexts. By providing automatic calculation algorithms under these heterogeneous environments, it improved the feasibility of risk assessment task for

these IT stakeholders. The risk assessment toolkit ranges from micro to macro, not only can cloud provider, network administrator but also software developer and service user can benefit from this work. In short, it accelerates the security assessment and task prioritization through automatic manner. These risk assessment approaches can be linked and output more accurate metrics if enough data enough data can be obtained in the future. The overall contribution could also be depicted in Figure 1.1. A blue solid arrow represents a completed construction of a risk assessment approach based on the existing standard risk assessment approach (i.e. CVSS). A red dotted line represents a possible link between two approaches.

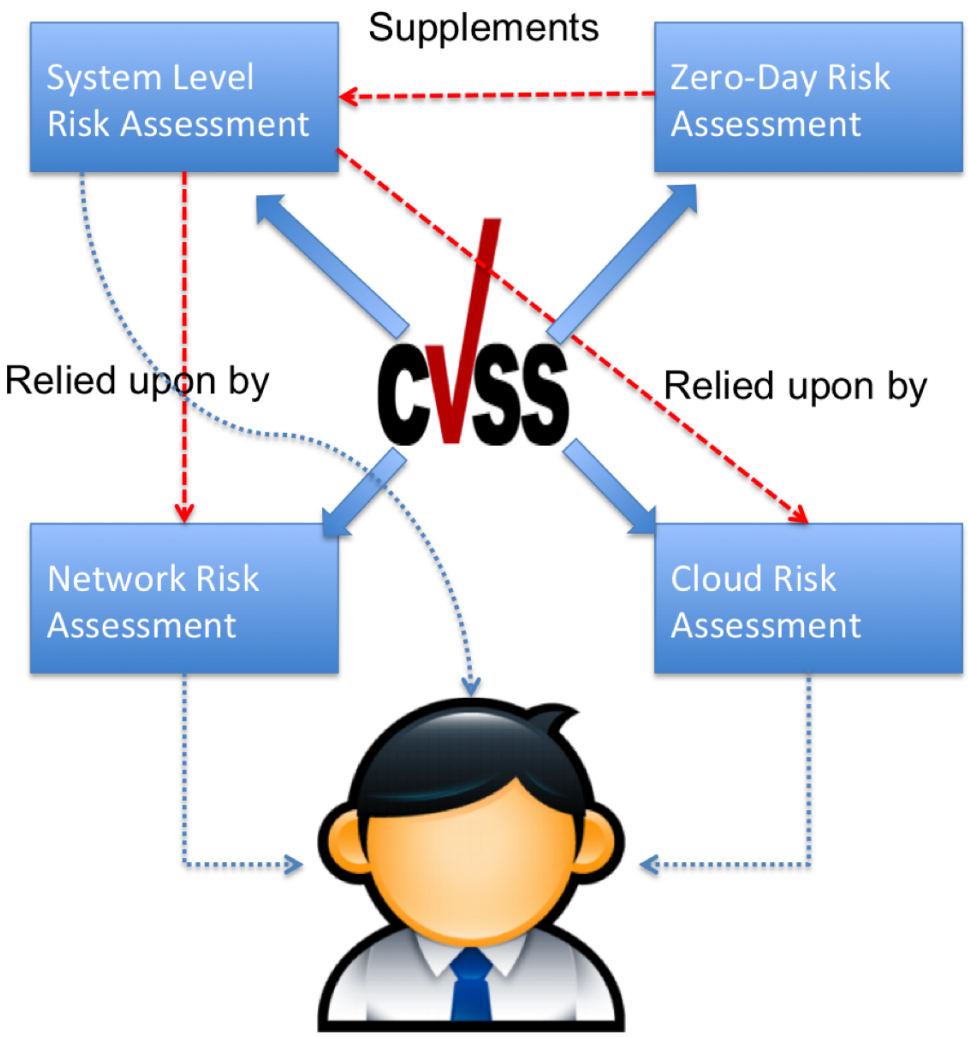


Figure 1.1: *A comprehensive risk assessment approach under multi-context cyber environment*

Chapter 2

Network Risk Assessment through Model Abstraction

Attack graphs are a common approach to security evaluation. It is often used in conjunction with risk assessment tools to provide recommendations to system administrators on how to mitigate the discovered problems.There are two main utilities of attack graphs: visualization and risk assessment. A major obstacle in these utilities is the size and complexity of attack graphs from even moderate-size networks. The large number of attack paths towards the same target not only makes the graph too dense to read, but also distorts risk assessment results by ignoring the fact that many of the attack steps are similar and not independent.

No solution proposed addressing the distortion problem in risk assessment caused by the redundancy in attack graphs, especially in the context of quantitative security assessment. Traditional approaches [35, 99, 100] would assess all the attack paths to the attacker's target without taking the similarities of these paths into consideration. Consequently, the explosion in the attack-graph's size could yield high risk metrics, often misleading the system administrator's judgment. While one could post-process the graph and remove such redundancy, like in previous works [69, 103], I believe a better approach is to pre-process *the input* to attack-graph generation so that such redundancy is removed by abstracting the network model, instead of the attack graph. There are a number of benefits of abstracting

the network model:

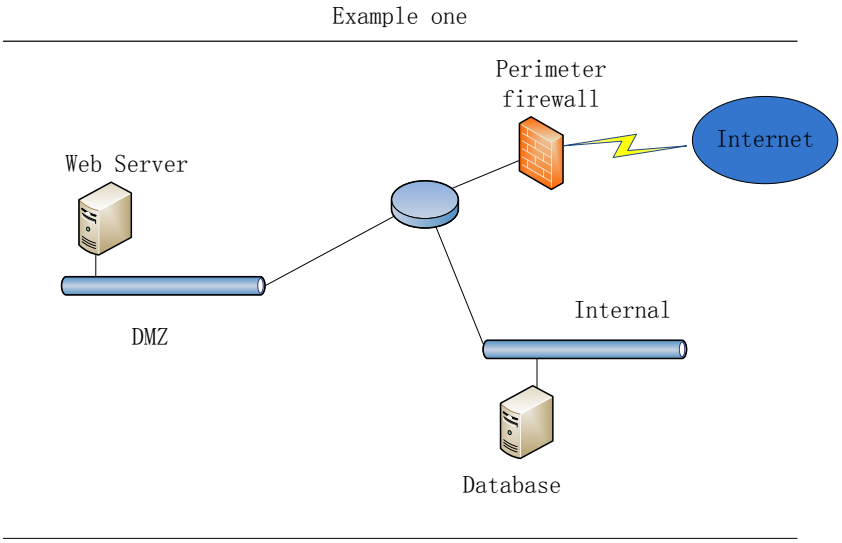
- From a human user's perspective, removing redundancy in the network description provides a better high-level view of both the system and the security vulnerabilities identified therein. The semantics of the abstract network model matches better with how a human would manage a large network system, and as a result the output of the attack-graph analysis is natural to communicate to human users.

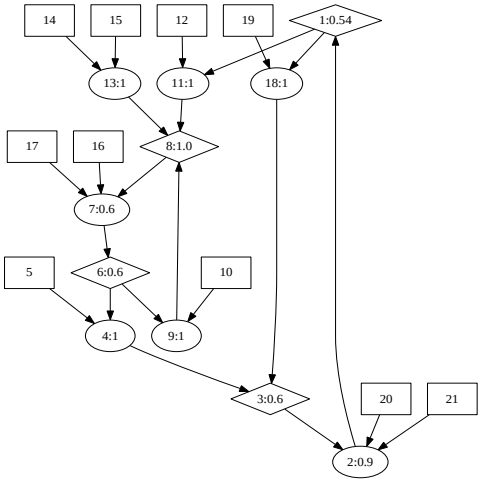
- After abstracting the network model, the distortion in quantitative security assessment results due to repetitive similar attack paths will be rectified.

In this chapter, I will introduce a network abstraction model, through which the size of attack graph can be significantly reduced. This abstraction model could increase both visualization of attack graph and accuracy of risk assessment approaches. The abstraction consists three steps: reachability-based grouping, vulnerability grouping and configuration based break down. Each of the abstracted node will represent a number of hosts with similar configuration and reachability. Under real network scenarios, the ratio between number of hosts and abstracted node can be large because clustering management is common. My approach is also easy to be extended with different grouping policies, which can help user to find a balance between scalability and accuracy.

2.1 An Example

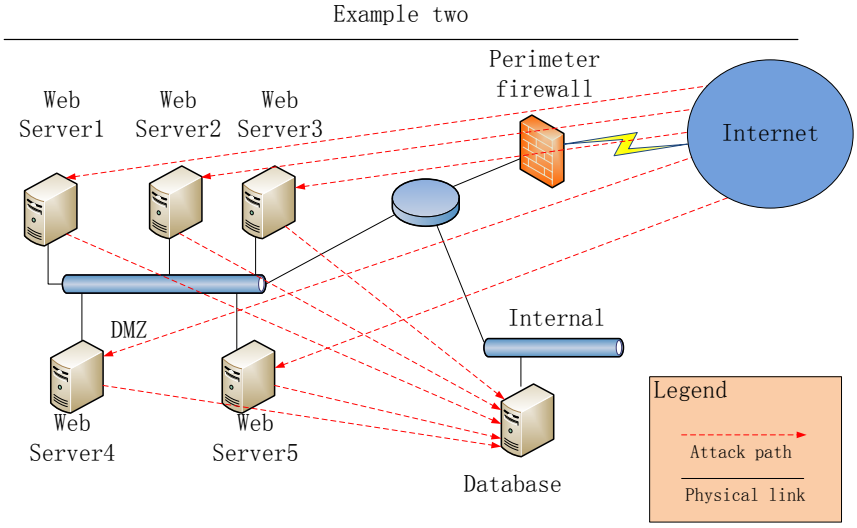
Figure 2.1 shows a simple network. An attacker could launch attacks from the Internet against the web server, which then provides him a stepping stone to exploit the database server in the internal network. The lower part of the figure shows a MulVAL attack graph [73, 74] generated from this network model. The labels of the graph nodes are shown at the right-hand side. Diamond-shaped nodes represent privileges an attacker could gain in the system; circle nodes represent attack steps that achieve the privileges; rectangular nodes represent network configuration settings.





1: execCode(database,user)
2: remote exploit of CVE-2009-2446
3: netAccess(database,tcp, 3306)
4: multi-hop access
6: execCode(webServer,user)
7: remote exploit of CVE-2009-1918
8: netAccess(webServer,tcp,80)
9: multi-hop access
11: multi-hop access
13: direct network access
15: attackerLocated(Internet)
18: multi-hop access

Figure 2.1: *Scenario of example one and its attack graph*



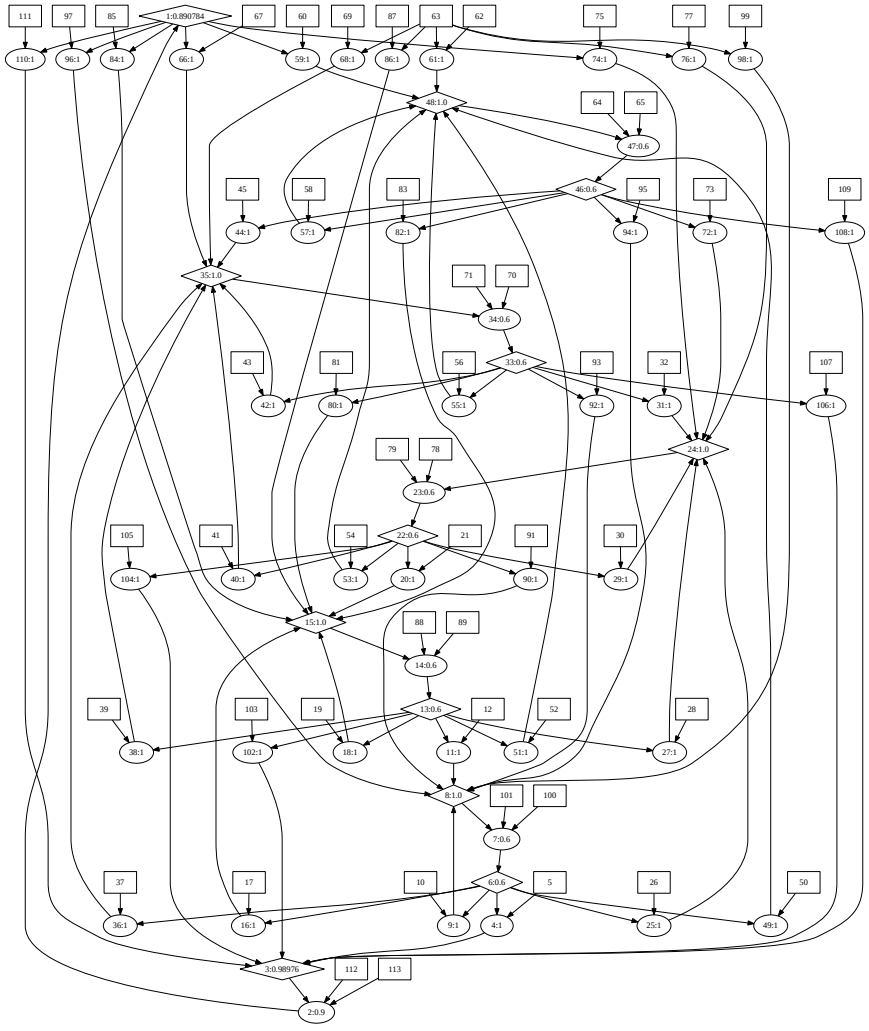


Figure 2.2: *Scenario of example two and its attack graph*

Figure 2.2 shows the topology and attack graph of a similar scenario, but with five identical servers in the DMZ zone. We can see that the attack graph gets very complicated. Human users, like a system administrator, may have difficulty tracing through the many identified attack paths. An abstracted view of the attack graph can highlight the real underlying issues in the network. We must also consider whether the multitude of attack paths shown in this attack graph reflects a realistic risk picture. The dotted lines in the network topology illustrate a subset of the attack paths identified in the graph. There are five ways to attack the database server, utilizing five different sources in the DMZ. However, the five servers in DMZ are identically configured. Thus if an attacker can exploit any one of them, he can exploit the others as well. In this case, having four more servers will not significantly increase the attacker's chance of success.

2.2 Network model abstraction

2.2.1 Abstraction criteria

Similarity among hosts

For large enterprise networks, it is not unusual to have thousands of machines in a subnet with same or similar reachability and configuration. If an attacker could compromise one of the machines, he is likely able to do the same for the others. This would result in a large number of similar attack paths in the attack graph. These attack paths should not be considered independent when assessing the system's security risk: if the attacker failed in compromising one of the hosts, he would probably fail on the others with the same properties (reachability and configuration) as well. Network reachability and host configuration determine to a large extent the exploitability of a host machine. For this reason, the machines with the same reachability and similar configurations can be grouped and treated as a single host.

Similarity among vulnerabilities

A single host may contain dozens or even hundreds of vulnerabilities, each of which may appear in a distinct attack path to further compromise the system. However, not all these paths provide unique valuable information since many vulnerabilities are similar in nature. They may belong to the same application, require the same pre-requisites to be exploited, and provide the same privilege to the attacker. From a human user's perspective, it is more important to know, at a higher level, that *some* vulnerability in the application could result in a security breach, rather than enumerating all the distinct but similar attack paths. Since vulnerabilities in the same application are often exploited by the same or similar mechanisms, if the attacker fails in exploiting one of them, it is reasonable to assume a low chance of successful attack by similar exploits. For this reason, these vulnerabilities can be grouped together as a single vulnerability and an aggregate metric can be assigned as the indicator on the success likelihood of exploiting any one of them, instead of combining them as if each exploit can be carried out with an independent probability. For example, when a host has 10 vulnerabilities in Firefox, we can say with X likelihood an attacker can successfully exploit any one of them, where X is computed based on each vulnerability's CVSS score [62], taking into consideration the similarity among the 10 vulnerabilities. One simple approach would be to use the highest risk probability value as representative of the whole set.

2.2.2 Abstraction steps

My network model abstraction process is carried out in three steps.

1. *Reachability-based grouping.* Hosts with the same network reachability (both to and from) are grouped together.

2. *Vulnerability grouping.* Vulnerabilities on each host are grouped based on their similarities.

3. *Configuration-based breakdown.* Hosts within each reachability group are further divided based on their configuration information, specifically the types of vulnerabilities they possess.

Reachability-based grouping

I group all the hosts based on their reachability information. I first give two definitions.

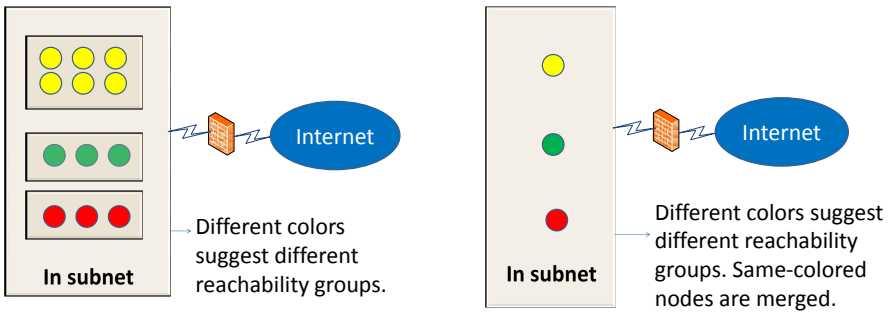
Definition 1. *reachTo(H) is a set of triples (host, protocol, port) where H can reach host through protocol at port. Similarly, reachFrom(H) is a set of triples (host, protocol, port) where host can reach H through protocol and port.*

Definition 2. *Let H_1 and H_2 be two hosts. I say $H_1 \equiv_r H_2$ if reachTo(H_1) = reachTo(H_2) $\wedge$ reachFrom(H_1) = reachFrom(H_2)*

I put hosts into the same reachability group if they belong to the same equivalence class $\equiv_r$. Then all the hosts in the same reachability group can be abstracted as a single node. Figures 2.3(a) and 2.3(b) illustrate this idea, and Algorithm 2.1 explains the grouping process. The grouping is applied to all the machines in a subnet. The interpretation of a subnet is a collection of machines communication among which is unfiltered. I incrementally add reachability information into a set. If host H's reachability has been recorded, I find the existing group through a hash map and put H into the corresponding group. Otherwise store the reachability information, create a new group label and map it to a singleton set with H in it. I do this for all the hosts in each subnet. The time complexity for this algorithm is $O(n^2)$ where n is the number of hosts in the network. I need to go over all the hosts within the subnet and for each host linear time is needed to identify its reachability information.

Vulnerability grouping

I group vulnerabilities on each machine based on the application they belong to. Typically vulnerabilities in one application will be of the same type (local, remote client or remote service). For example, vulnerabilities of Adobe Reader are remote client since they are



(a) Before reachability-based grouping (b) After reachability-based grouping

Figure 2.3: *Before and after reachability-based grouping*

Algorithm 2.1 Pseudocode for reachability-based grouping

Require: A set of (reachTo(h), reachFrom(h)) for each host h in a subnet.
Ensure: A hash map L, which maps a group label α to a list of hosts having the same reachability (reachTo and reachFrom).

```
Lr ← {} {Lr is a set of triples (α, reachToSet, reachFromSet).}
Queue Q ← all the hosts of the given subnet
L ← empty map {initialize the return value}
while Q is not empty do
  n ← dequeue(Q)
  if Lr contains (α, reachTo(n), reachFrom(n)) then
    L[α] ← L[α]∪{n} {if the reachability of n is the same as some other host that has
    been processed, add n to its equivalent class.}
  else
    create a fresh α
    Lr ← Lr ∪(α, reachTo(n), reachFrom(n)) {Otherwise put its reachability information
    into Lr}
    L[α]← {n}
  end if
end while
return L
```

always triggered when a user opens the application on a malicious input, possibly sent by a remote attacker. Security holes in IIS, on the other hand, most likely belong to remote service vulnerabilities. After grouping based on applications, I can provide the system administrator a clearer view of the system's vulnerabilities — instead of showing a long list of CVE ID's, I show the vulnerable applications that affect the system's security. One issue that needs to be addressed is how to assign an aggregate vulnerability metric to the virtual vulnerability after grouping. Such vulnerability metrics, like CVSS scores, are important in quantitative assessment of a system's security. Intuitively, the more vulnerabilities in an application, the more exploitable the application is. But the degree of exploitability does not simply grow linearly since many of the vulnerabilities will be similar. My current grouping algorithm (Algorithm 2.2) simply takes the highest value, but it will be straightforward to plug in a different aggregation method.

Algorithm 2.2 Pseudocode for vulnerability grouping

Require: A set of ungrouped vulnerabilities on a machine (S_u)
Ensure: A hash map L that maps an application to its vulnerability score

```
Lr ←{} {Lr is a set of applications that have appeared so far}
L ← empty hash map
while S_u ≠ {} do
  take v from S_u
  if Lr contains (v.application) then
    if L[v.application] < v.score then
      L[v.application] = v.score
    end if
  else
    L[v.application] = v.score
    Lr.add(v.application)
  end if
end while
return L
```

Configuration-based breakdown

For hosts in the same reachability group, their configurations could be different from one another. Thus, if an attacker is able to exploit one host within the group, it does not mean he could compromise the others as well. This means grouping based on reachability alone is too coarse. In order to reflect differences in attackability, we need to "break down" the merged node based on configuration settings. In my current implementation, I have only included software vulnerability as the configuration information. When deployed on production systems, one can rely upon package management systems to decide whether two hosts have the same or similar software set up. Algorithm 2.3 shows the process of configuration-based grouping. The algorithm iterates over all the hosts in a reachability group and records its configuration information. If a host's configuration matches one previously recorded, meaning some other hosts have the same types of vulnerabilities, this host will not be recorded in the set. At the end of the algorithm, the returned set only contains one representative host for each group of hosts with the same reachability and configuration. The complexity of the algorithm is linear in the number of hosts.

Algorithm 2.3 Pseudocode for configuration-based break down

Require: A list L, each element of which is a set of machines belonging to the same reachability group, and with the vulnerabilities grouped.

Ensure: Further-refined group S_c based on vulnerability information. Each element in S_c is a representative for a group of hosts with the same reachability and configuration.

```
while L≠{} do
    remove h from L
    Lr ← empty map; {Lr is a set of pairs (hostname, configuration). It is used to store
    the distinct configurations that have appeared so far.}
    if Lr contains (_, h.configuration) then
        continue {if its configuration has appeared before, skip}
    else
        Lr.add((h, h.configuration)) {if its configuration has not appeared before, record it}
    end if
end while
S_c = ⋃_{(h,_)∈Lr} h {collect all representative hosts in Lr and put them into S_c}
return S_c
```

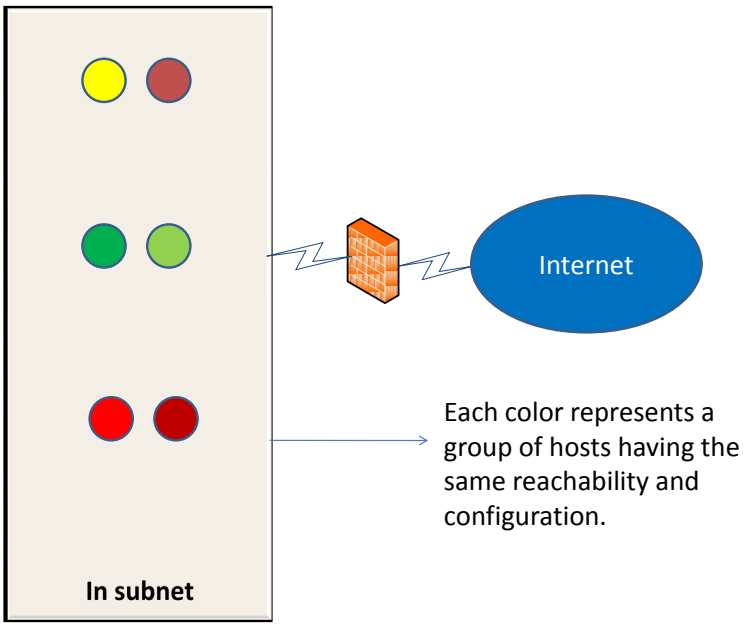


Figure 2.4: *After configuration-based breakdown.*

2.3 Experimentation Result

To evaluate the effect of model abstraction on quantitative security assessment of computer networks, I apply probabilistic metric models [35, 99] on the generated attack graphs. In such metric models, each attack step is associated with a (conditional) probability indicating the success likelihood of the exploit when its pre-conditions (predecessor nodes) are all satisfied. The model then computes the absolute probability that a privilege can be obtained by an attacker based on the graph structure. I use MulVAL [73, 74] attack-graph generator in the evaluation. My security metric implementation follows Homer's algorithm [35].

I created one scenario to illustrate the visualization effect and rectification on the distortion in metric calculation generated by the large number of similar attack paths. The topology information of the example is shown in Fig. 2.5. There are three subnets: Internal Servers, DMZ, and Normal Users. Each subnet has ten machines, evenly divided into two different types of configuration (one is Linux and the other Windows). Machines with

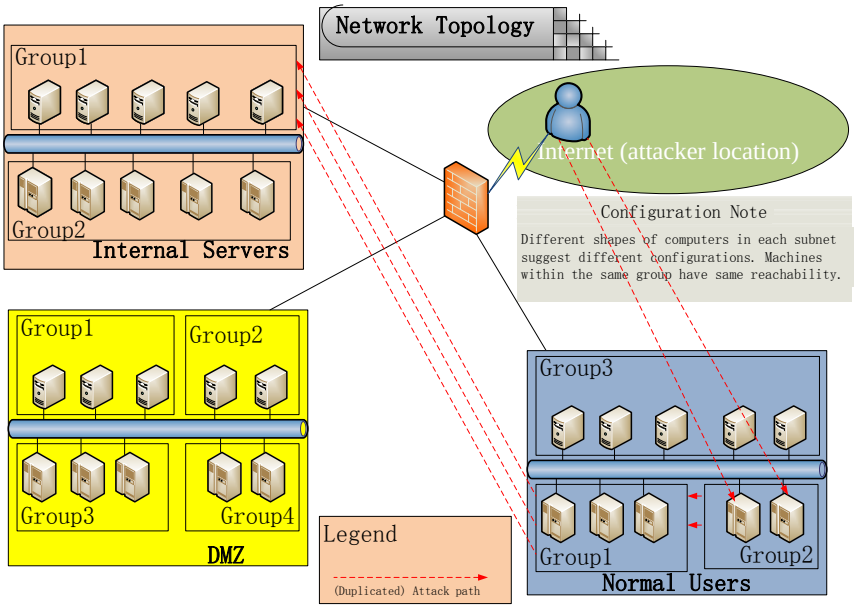


Figure 2.5: *Network topology.*

Table 2.1: *Reachability Table*

source		**destination**		**protocol**	**port**
subnet	group	subnet	group		
Internet		DMZ	1	tcp	80
DMZ	1	Internet		tcp	25
Internet		DMZ	4	tcp	80
DMZ	4	Internal	2	tcp	1433
User	2	Internet		tcp	80
User	3	Internet		*	*
Internet		User	2	tcp	80
User	1	Internet		*	*
User	1	Internal	1	nfs	
User	1	Internal	1	Tcp	3306

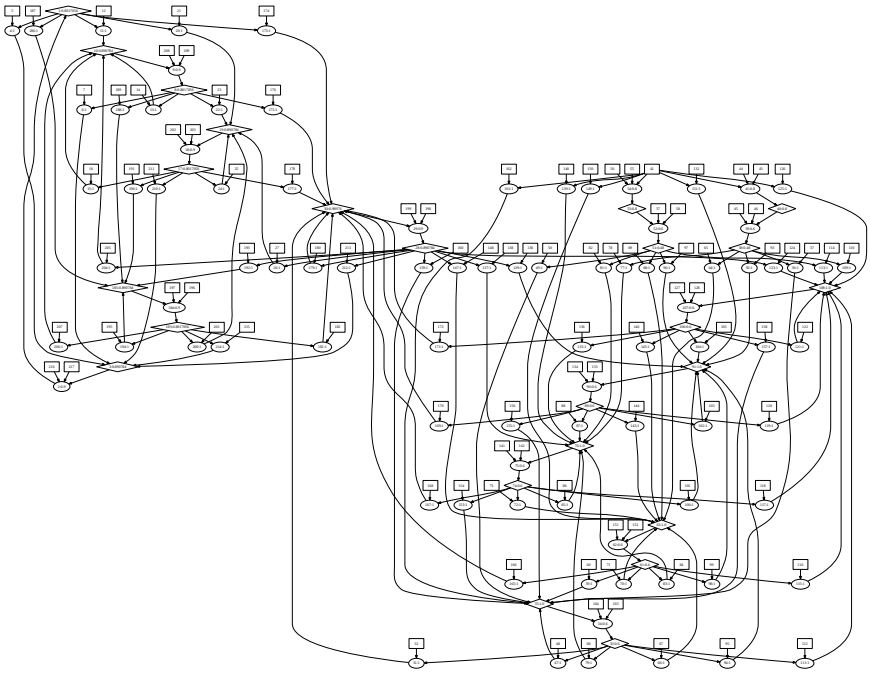
different shapes represent different configurations. Machines in the same group have the same configuration and reachability. There are two types of vulnerabilities on each host, and the types of vulnerabilities could be either local, remote server or remote client. The reachability relations among those host groups can be found in Table 2.1. The table does not include reachability within a subnet, which is unfiltered. If a group does not have any inter-subnet reachability, it will not show up in the table.

2.3.1 Attack graph generation

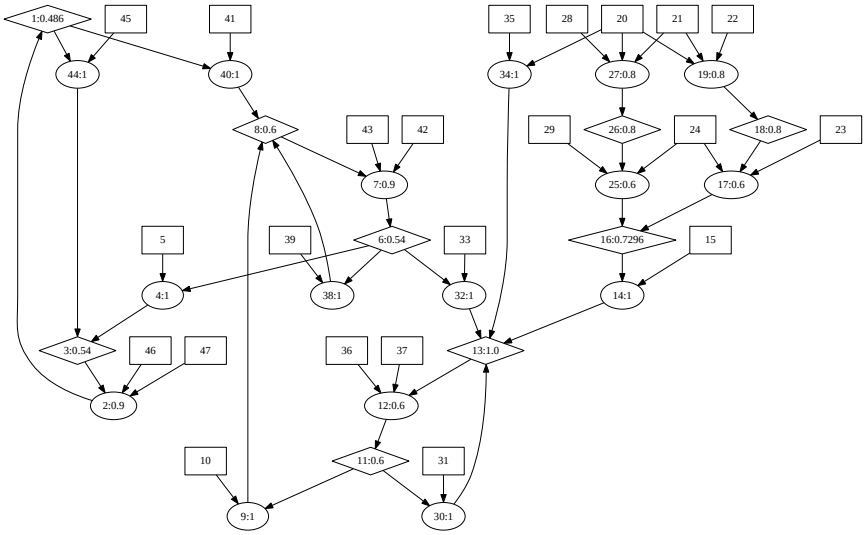
I created the input for MulVAL based on the configuration of the network, and I ran the abstraction model generator to generate an abstracted input. I ran MulVAL with both original and abstracted input and obtained two different attack graphs, shown in Figures 2.6(a) and 2.6(b). The size of the attack graph was reduced significantly after abstraction (281 arcs and 217 vertices, to 55 arcs and 47 vertices). I verified that all the "representative" attack paths leading to the attacker goal are retained in the abstracted model.

2.3.2 Quantitative security metrics

I compared the quantitative metrics results obtained from the original input and the abstracted input. There is a significant difference between the risk metrics on the original network (0.802) and the abstracted one (0.486) for a three-hop attack which is the deepest chain in this experiment (illustrated in the red dotted lines in Fig. 2.5). This attack chain includes three sets of attack steps: 1) from Internet to Group2 in the "Normal Users" subnet, via client-side vulnerabilities; 2) from Group2 to Group 1 in the "Normal Users" subnet, via service vulnerabilities; 3) from Group1 in the "Normal Users" subnet to Group1 in the "Internal Servers" subnet, via service vulnerabilities. Each group here refers to a set of hosts with the same reachability and configuration (vulnerabilities). Usually there are multiple attack paths between two groups since there are multiple hosts within each group and they have similar configurations; thus the multiple attack paths have similar natures. From a pure probabilistic semantics, the more paths between two groups, the higher success likelihood the attacker will gain in moving on these paths. However, these paths are not independent and failure on one of them would likely indicate failures on the other; therefore the higher risk metrics are not justified. Moreover, the hosts in the two groups are equivalent in terms of the network access they provide the attackers. Due to the above reasons, the attack paths should be merged into one, before quantitative risk assessment.



(a) Attack graph of the original model (281 arcs and 217 vertices).



(b) Attack graph of the abstracted model (55 arcs and 47 vertices).

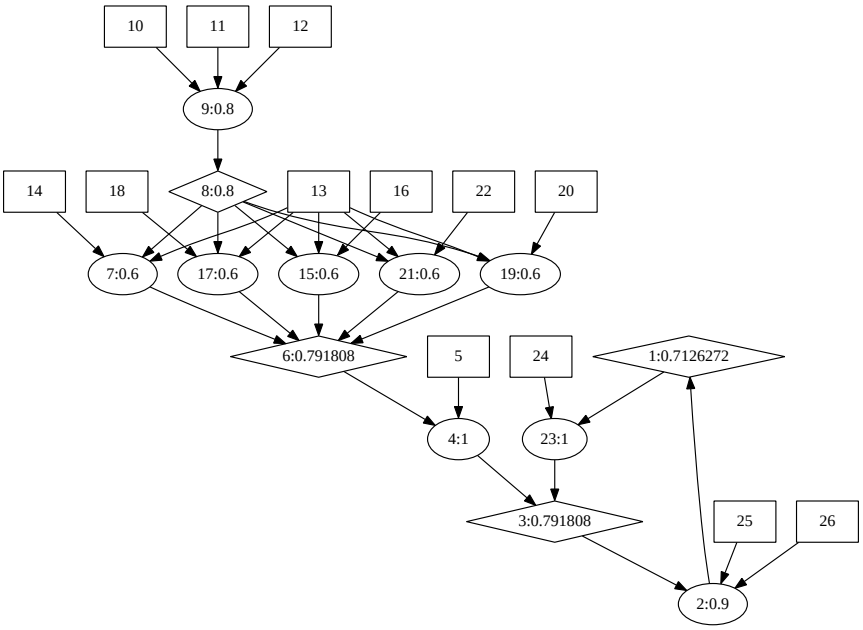
Figure 2.6: *Comparison of attack graphs from original and abstracted models*

By removing redundancy in the attack graphs through model abstraction, the distortion in the risk assessment result has been effectively avoided.

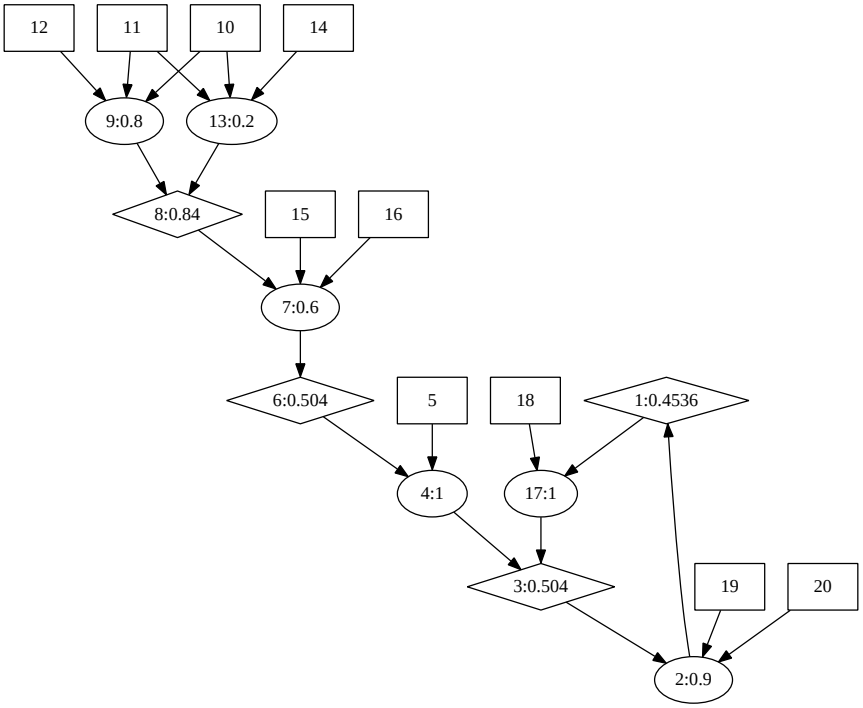
To demonstrate the effect of vulnerability grouping on the quantitative security assessment result, I used the network topology shown in Fig. 2.1, assuming there are five client-side vulnerabilities (from the same application) on the web server and the remote service vulnerability has been patched. I then computed the likelihood that the web server could be compromised through any of the client-side vulnerabilities, assuming the client program may occasionally be used on the server. The nature of client-side vulnerabilities from the same application are similar from both attacker and the victim's perspective, because the victim would open the same application to trigger the exploits, and due to the similar functionalities (and therefore program components) of the same application, the security holes are also similar. If an attacker knows the structure of the application very well, he should be able to utilize the vulnerability easily; if he does not understand the mechanism of the software, he probably will not be able to utilize any of the security holes with ease. Therefore viewing the same type (client-side or service) of security holes on an application as one is more realistic than treating them independently. I compared the results before and after grouping vulnerabilities. It is obvious that the complexity of the attack graph is reduced significantly from Figure 2.7(a) to Figure 2.7(b). More importantly, the quantitative metrics indicating the likelihood that the server can be compromised through one of the client-side vulnerabilities drops from 0.71 to 0.45. This is a more realistic assessment, since the five client-side vulnerabilities are similar and should not significantly increase the attacker's success likelihood.

2.4 Discussion

I have presented an abstraction technique to aid in network security assessment based on attack graphs. I show that the large amount of repetitive information commonly found in attack graphs not only makes it hard to digest the security problems, but also distorts the



(a) Attack graph of a single machine before vulnerability grouping.



(b) Attack graph of a single machine after vulnerability grouping.

Figure 2.7: *Effect of vulnerability grouping on a single host*

risk picture by disproportionately amplifying the attack likelihood against privileges that have a large number of similar attack paths leading to them. I proposed an approach to abstract the network model so that such repetitive information is removed before an attack graph is generated. The abstraction happens at both the network and the host level, so that machines that have the same reachability relation and similar configurations with respect to vulnerability types are grouped together and represented as a single node in the abstracted model. The experiments show that such abstraction not only effectively reduces the size and complexity of the attack graphs, but also makes the quantitative security assessment results more conforming to reality. This shows that appropriate abstraction on the input is a useful technique for attack graph-based analysis.

The abstraction techniques I have proposed are mostly suitable for risk assessment on the macroscopic level of an enterprise network. Abstraction unavoidably loses information and in reality no two hosts are completely identical. The abstracted network model can help in identifying security risks caused by the overall design and structure of the network, but may lose subtle security breaches that may occur due to, *e.g.* misconfiguration of a single host that is mistakenly deemed identical to a group of other hosts since the details of the differences may have been abstracted away. In general the more homogeneous the system is, the more pronounced the effect of abstraction will be. However, since no two hosts are really completely identical, the process is a balancing act. Being overly detailed about a host's configuration may lead to no possibility of abstraction and result in a huge attack graph where important security problems are buried. On the other hand, overly abstract models may lose the important information for subsequent analysis. System administrators need to define the granularity based on their expected available time and effort to run such process.

Chapter 3

Package Dependency Risk Assessment

The previous chapter improved accuracy and visualization of risk assessment over large scale networks, whose accuracy highly depends attack surfaces of individual systems within the network. Attack surface usually refers to exploitable resource exposed to attackers [56, 57]. The attack surface brought by a vulnerability could be dramatically enlarged when more packages installed depending on the vulnerable application because more resource can be accessed by the attacker to exploit the vulnerability. Therefore the attack surface metric could serve as an effective indicator for vulnerability assessment, which is considered as a critical task for security prioritization. Currently, the well known and de facto standard vulnerability scoring system – common vulnerability scoring system (CVSS) [63] – quantifies the risk for each known vulnerability. Specifically, CVSS measures exploitability metrics (access vector, access complexity, and authentication) and impact metrics (confidentiality, integrity, and availability loss) of a vulnerability, which are then used to calculate a base score ranging from 0 to 10 indicating the severity of the vulnerability. Besides the base score, CVSS also provides temporal and environmental scores for system administrators to fill out in order to assess vulnerabilities appropriately, where a temporal score constructs a correlation between the exploitability of a vulnerability and time, and the environmental score evaluates the coverage of the vulnerability over entire network and potential loss

associated with these machines. Even though the environmental score provides context aware factors for CVSS, it is mostly subjective to system administrators.

Moreover, CVSS does not take into consideration of package dependency, which, based on our analysis in this chapter, dramatically affects the exploitability of a vulnerability, especially when it appears in a prevalent package used by many other packages. Therefore current CVSS does not reveal the fact that vulnerabilities on highly depended packages usually bring larger attack surfaces compared to those detected on a client application, even when they have the same CVSS scores. Because packages depended by a number of applications are usually more exposable than "ground" software (with no dependent), attackers have more incentive to intrude a system through each of these dependents (or their dependents). Therefore, the attack surface brought by package dependency should not be ignored, and accurately measuring the attack surface is non-trivial when evaluating vulnerability severity.

In this chapter, I will introduce a risk evaluation approach on system level attack surface. I captured package dependency at system level by constructing a set of systematic risk assessment approaches. Each of these approach is able to output a risk metric at different granularities (vulnerability level, component level, package level or system level). These approaches bridge the gap between the standard vulnerability assessment approach and system wide attack surface evaluation. With my approach, vulnerability and component level metrics can assist system administrators in prioritizing patching or hardening plans towards the entire system, while the overall package and system level metrics can help developers to choose secure and reliable development images, platforms, and specific systems. My solution also helps other stakeholders to observe the evolution of package dependency based attack surface for a given system.

I do experiments on systems with JRE vulnerabilities, with installed packages and applications relying on JRE. The experimental results confirm that the attack surface of a vulnerability is enlarged as the number of its dependents increases. The same trend is

observed at component and package levels.

3.1 Overview

3.1.1 A Real Motivating Example

To motivate my attack surface analysis with package dependency, I systematically analyze the risk trend of a set of VMware products through VMware Security Advisories (VMSA)[1]. Each VMSA indicates an official notification regarding a set of known security vulnerabilities that affects VMware products, each of which represents a Common Vulnerabilities and Exposures (CVE) record included in the U.S. National Vulnerability Database (NVD[2]). Each VMSA entry includes the origin of the vulnerabilities, vulnerability IDs, affected applications, and proposed solutions to the issue. Based on my analysis of VMSA entries from July 2007 to December 2012, I find out that almost two thirds (56/90) of the VMSAs include vulnerabilities originated from third party applications that affect VMware products, as Table 3.1 shows. For instance, ESX – the last generation hypervisor – may be exploited by vulnerabilities described in 27 VMSAs detected on the Linux management console, which provides management functions for ESX like executing scripts or installing third party agents for hardware monitoring, backup, and system management [4].

For another instance, Java Runtime Environment (JRE) is required by a number of VMware products including ESX, Server, vMA, vCenter, and vCenter Update Manager, therefore a known vulnerability on JRE could possibly make each of these products exploitable. Other major attack surface carriers include OpenSSL (9 out of 90), Kerberos 5 (8 out of 90), Apache Tomcat (6 out of 90), and libxml (6 out of 90). Note that one VMSA usually mentions multiple risks included in different applications (See Table 3.1 for details).

My analysis with VMSA motivates a security metric with the consideration of package dependency, which can help system administrator and software developer to identify vulner-

[1] http://www.vmware.com/security/advisories/
[2] http://nvd.nist.gov/

Table 3.1: *Risks from Third Party Packages to VMware Products*

Third-party Package Name	# of VMSAs	Affected VMware Products
Console Operating System	27	ESX
JRE	11	ESX, Server, vMA, vCenter, vCenter Update Manager
OpenSSL	9	ESX, ESXi, vCenter
kerberos5	8	ESX, ESXi
Apache Tomcat	6	ESX, vCenter
libxml	6	ESX

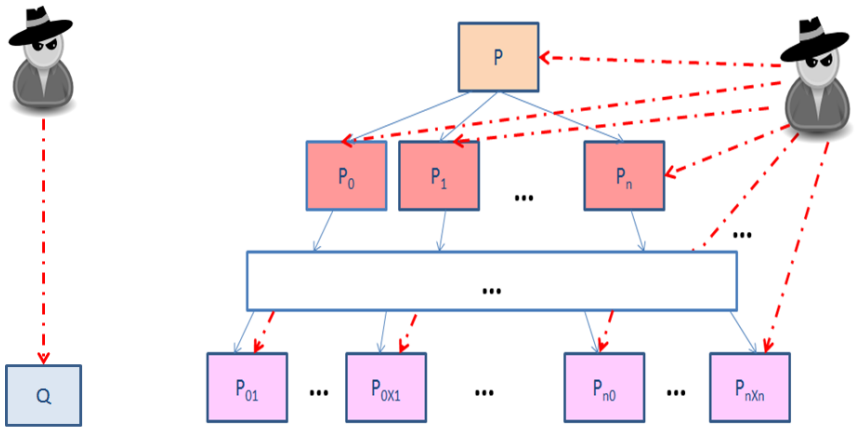


Figure 3.1: *Comparison of attack paths to a vulnerable client side application* `Q` *and a highly depended library* `P`.

abilities on highly depended programs (e.g., JRE and Linux_console) with larger attack surfaces, compared to others such as client side vulnerabilities (see Figure 3.1). Consequently, the system administrator may want to patch a JRE vulnerability affecting a number of products earlier than others even they may have the same CVSS score. A system level metric can also help stakeholders in choosing system images with smaller attack surface and monitor how the dependency based attack surfaces evolve over time.

3.1.2 Why Component Level Dependency Analysis?

From the perspective of software engineering, a system can be decomposed into various of packages. One package can usually be further divided into one or more components, each of which is made up from classes with related functions. From above motivating example with

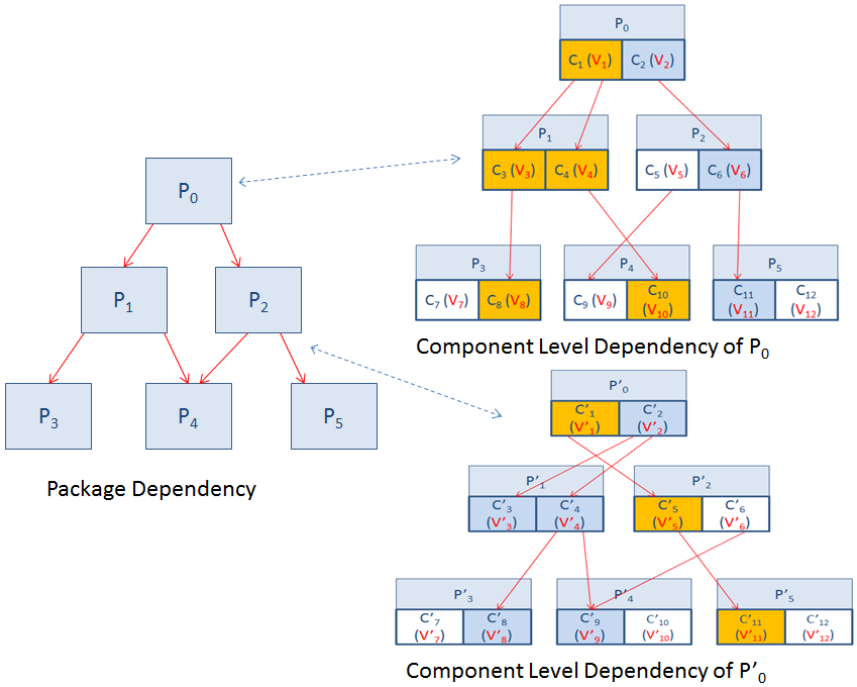


Figure 3.2: *One package level dependency with two different component level dependencies.*

VMSA, we have seen attack surfaces from third party packages should not be ignored for risk analysis, and we need to look into package dependencies to know how the attack surface is injected by external packages to a system. When measuring such *dependency based attack surfaces*, I analyze at component level for the following reasons.

More accurate dependency information than package level: Component level dependency is finer-grained than package level, therefore it could locate attack surfaces with higher accuracy. As Figure 3.2 shows, given two packages with the same dependency map at package level, their attack surfaces could vary significantly if known vulnerabilities on the two packages are on components with different dependency maps. Also, components on the same package should be differentiated as their effects on the attack surface can be significantly different. My experiments (see Section 3.3 and Table 3.4) indicate that component `2d` in a vulnerable JRE has the largest number (17) of vulnerabilities. However, it is only lightly depended by installed applications in my system (see last three columns in Table 3.2). Therefore, its attack surface ranks lower than the heavily depended component

`deserialization` even though the latter only has one detected vulnerability.

Table 3.2: *Historical Vulnerable JRE Components/Packages & Number of its Dependents on Different Applications*

Component Name	JRE Package Name	# of Historical Vulnerabilities	# of dependent classes		
			`FreeHEP`	`Weka`	`LibreOffice`
font	java.awt.font	4	0	1	0
2d	java.awt.geom	20	0	1	5
beans	java.beans	1	0	114	19
deserialization	java.io	3	14	378	1124
networking	java.net/javax.net	2	1	14	88
rmi	java.rmi	1	0	4	0
concurrency	java.util.concurrent	1	0	1	2
imageIO	javax.imageio	7	0	4	5
java web start	javax.jnlp	18	0	0	0
jmx	javax.management	4	0	0	0
ldap	javax.naming.ldap	1	0	0	0
scripting	javax.script	2	0	0	0
kerberos	javax.security.auth.kerberos	1	0	0	0
sound	javax.sound	13	0	0	0
swing	javax.swing	8	2	167	128
xml	javax.xml	3	1	4	26
corba	org.omg.CORBA	3	0	0	0
deployment	N/A	15	0	0	0
deployment toolkit	N/A	4	0	0	0
hotspot	N/A	4	0	0	0
install	N/A	3	0	0	0
jsse	N/A	2	0	0	0
jvm	N/A	5	0	0	0
oracle jrockit	N/A	1	0	0	0
(un)Pack200	N/A	5	0	0	0
proxy mechanism	N/A	2	0	0	0
update	N/A	3	0	0	0

Less complex dependency information than class level: I keep my dependency analysis at component level rather than go further into class or object level because it is usually difficult to distinguish the sources or causes of vulnerabilities at that level. Each component is a unit to realize a set of related functions. Classes within the same component are usually more integrated and interacted compared to those in different components. Therefore for each vulnerability, its exploitability highly depends on its accessibility at the component level. Previous studies also show that a vulnerability becomes significantly more exploitable when attackers know that its component is accessible [77, 78]. Besides, it is usually difficult to construct a map between vulnerabilities and the classes on which they detected. Furthermore, proprietary software vendors usually do not disclose their product information at class level. However security bugs and alerts are usually maintained by database like Bugzilla

at component level[3], which makes the vulnerability-component map retrievable [67]. Moreover, the complexity of a class level dependency map is exponentially higher compared to a component level dependency graph. I believe it is infeasible to achieve efficient analysis with class level graph when dealing with a complex system including a large number of software packages.

3.2 Dependency-based Attack Surface Analysis

This section explains the details of my dependency-based attack surface analysis. Before that I explain the definitions for various attack surface metrics.

3.2.1 Package Dependency at Component Level

In general, a package dependency refers to a code reuse by a component from the library packages that it relies upon. Such code reuse could be at either binary or source code level. For example, third party code could be called as a compiled jar file or be imported as head files in source code. As shown in Figure 3.2, each directed line represents one dependency relationship, where the destination node represents the package or component that reuses some codes from the source node package or component.

In the analysis, I do not differentiate dependency strength at component level. Even though other metrics such as the number of references between the two components can be obtained and used as the weight, the correlation between these metrics and the strength of dependency is difficult to be determined and judged without a comprehensive analysis over the source code of a target package. Therefore, I assign an equal weight 1 to each dependency between two components in my analysis. But I still keep a weight variable in the algorithms just for future customization of the dependency weight based on different preferences.

[3]A vulnerability is usually identified as a security bug in Bugzilla.

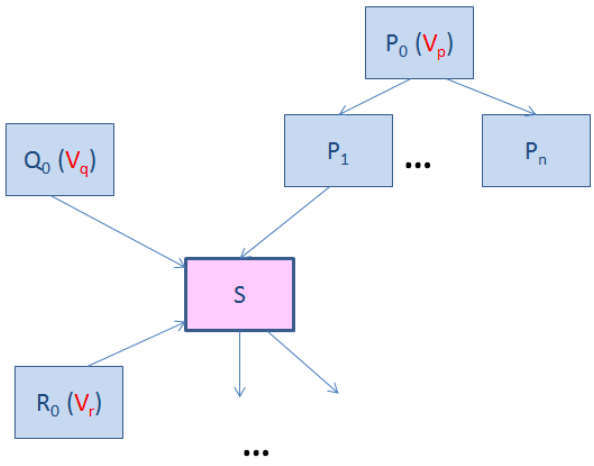


Figure 3.3: *Incoming attack surface injected into Package S.*

3.2.2 Attack Surface Metrics

The attack surface of a vulnerability refers to exposed resources can be utilized by attackers to exploit the vulnerability successfully. The attack surface of a component, a package, or a system is the aggregated attack surfaces of individual vulnerabilities that affect it. Based on the origin and the entrance of the risk, I classify attack surface into two categories: outgoing attack surface and incoming attack surface. Outgoing attack surface indicates how much risk can be brought to the whole system from a given vulnerability, component, or package. The right part of Figure 3.1 illustrates possible attack paths through outgoing attack surface of package `P`, which is depended by many packages $\texttt{P}_0$, $\texttt{P}_1$, ..., $\texttt{P}_n$, each of which is in turn depended by other packages. Incoming attack surface means the injected risk to a given component, package or system from the packages it depends upon, as illustrated in Figure 3.3.

I define several package dependency based attack surface metrics according to variant evaluation granularities, as illustrated in Table 3.3 and explained briefly as follows.

- Vulnerability attack surface (VAS): the outgoing attack surface from a single vulnerability to the whole system;
- Component attack surface (CAS): the outgoing attack surface from a single component

to the whole system;

- Package attack surface (PAS): the outgoing attack surface from a single package to the whole system;
- System attack surface (SAS): package dependency based attack surface of the entire system;
- inCAS: the attack surfaces of a given component injected by the code that it relies upon;
- inPAS: the attack surfaces of a given package injected by the code that it relies upon.

Among these metrics, VAS, CAS, and PAS are outgoing attack surface metrics, and inCAS and inPAS are incoming attack surfaces. SAS can be either incoming or outgoing attack surface as the dependency based attack surface within one system is self looped.

Table 3.3: *Attack Surface Definitions*

Attack Surface Name	Definitions
VAS	Out-going attack surface from a vulnerability
CAS	Out-going attack surface from a component
PAS	Out-going attack surface from a package
SAS	package dependency based attack surface for a system
inCAS	incoming attack surface to a component
inPAS	incoming attack surface to a package

3.2.3 Component-based Attack Surface Analysis

Vulnerability Attack Surface

I define VAS as a system wide package dependency based attack surface originated from a given vulnerability. VAS can be used to compare the exploitabilities of different vulnerabilities *within the same system*. The comparison results can be used to prioritize patching or hardening tasks at vulnerability level.

As Algorithm 3.1 shows, for each vulnerability, I first identify its component. Usually, the vulnerability-component map is provided by software vendors through security advisories, e.g., Oracle Security Advisories[4]. Starting from the component of the target vulnerability, I do a breadth first search until depth d, where d is the level of dependency. For example, if package p_a depends on p_b which depends on p_c, then when evaluating p_a, p_a and p_b are considered but not p_c if d is one. However, all of them are considered when d is larger than one. The depth could be customized based on user preferences. Each component (directly or indirectly) depending on the vulnerable component is considered as part of the attack surface brought by the vulnerability. The impact factor on each component is the attack surface of the target vulnerability exposed through that component. I assign the CVSS score of the vulnerability as the impact factor of the component where it resides (the 'vulnerable component')[5]. For components on multiple depending chains from the vulnerable component, I only consider its closest dependency and ignore the rest. For example, component c_a depends on c_b which depends on c_c, and c_a also depends on c_c directly. Under this circumstance, I ignore the dependency $c_a \Longrightarrow c_b \Longrightarrow c_c$ but only consider $c_a \Longrightarrow c_c$.

I define a damping factor[6] (ranging from 0 to 1) to represent the residual risk after each level of dependency, which is used to estimate attack surface from/to nested depended packages. The impact factor on a given component equals to the multiplication of the dependency impact value from the component it depends on (the dependency impact value is returned by function depImpact (c_1, c_2) when c_1 depends on c_2. I assign "1" to all impact values in the experiments because I treat all dependencies equally as mentioned in Section 3.2.1), the damping factor and the impact factor of the component it depends on. Their impact factor values will be eventually added up to one number, indicating the attack surface of the given vulnerability to the whole system.

[4] http://www.oracle.com/technetwork/topics/security/

[5] The calculation of impact factors of dependent components will be illustrated in the following paragraph.

[6] I assign 0.1 as the damping factor for the experiments

In a nutshell, I process a weighted (component-based) dependency graph through breadth first search, I calculate an impact factor for each component (within the dependency graph from the vulnerable component) from the given vulnerability. I then add up all of these impact factors into one number, indicating the attack surface exposed by the target vulnerability.

Component Attack Surface

A component's outgoing attack surface CAS indicates the attack surface brought by the component to the whole system. The CAS helps a system administrator to identify dependency based attack surface at component level. The calculation of CAS is based on VAS as following:

$$CAS(c_0, d) = \sum_{component(v)==c_0} VAS(v, d) \tag{3.1}$$

where $VAS(v, d)$ can be solved through Algorithm 3.1 and $component(v)$ returns the component containing vulnerability v. Basically, the CAS of a component is an aggregation of the VASes of all vulnerabilities detected in the component.

Package Attack Surface

A package's outgoing attack surface PAS indicates the attack surface exposed by the package through package dependency to the whole system. Similar to the usage and calculation of CAS, PAS can be used to compare the package dependency based attack surfaces among packages installed on the same system. The PAS value of a package is an aggregation of CAS of each component belonging to the package, as the following shows:

[12]I assign "1" to all DIV as mentioned in Section 3.2.1

[8]The damping factor represents the residual risk after each level of dependency. User can assign a value between 0 and 1 based on their own estimation.

Algorithm 3.1 Dependency-based Attack Surface Measurement for Individual Vulnerabilities: **VAS**(v_0, d)

Require: Parameters: v_0 – the Target vulnerability; d – Depth of assessment.
System configurations:
A map between the vulnerability v_0 and its component component (v_0).
A system wide component dependency map (dependents of component c are dependOn(c)).
Ensure: The package dependency based attack surface VAS brought by vulnerability v_0.
$c_0 \leftarrow$ component(v_0) {Retrieve the vulnerable component}
Queue Q $\leftarrow (c_0, 0)$ {Q is a queue of pairs (vulnerableComponent, depth)}
Table $v_0.t \leftarrow$ *empty table*
{$v_0.t$ is a table tracking processed components. The key is the affected component and the value is its impact factor from vulnerability v_0.}
$v_0.t$.put($c_0, v_0.cvss$) {The impact factor of c_0 equals to the CVSS score of v_0}
while Q is not empty **do**
  $(c_n, n) \leftarrow$ dequeue(Q)
  if $n \geq d$ **then**
    continue {if current component has already reached the pre-defined deepest level, then no need to retrieve its dependents}
  end if
  for each c_k in dependOn(c) **do**
    if $v_0.t$.containsKey (c_k) **then**
      continue
      {If the component has been previously processed, then it will be skipped}
    end if
    Q.enqueue($c_k, n+1$) {Update Q in order to process dependents of c_k if within the predefined depth}
    $IF_c = v_0.t$.get(c) {retrieve the impact factor of the current component c}
    $DIV = depImpact(c, c_k)$
    { $depImpact(c, c_k)$ returns dependency impact value[7]between c and c_k.}
    $IF = DIV \times DF \times IF_c$ {DF means Damping Factor[8]. This is the calculation of impact factor (IF) of component c_k}
    $VAS+ = IF$ {Cumulatively update attack surface}
    $v_0.t$.put(c_k, IF) {Update processed element table}
  end for
end while
return VAS {Sum up all impact factors of v_0 into VAS}

$$PAS(p_0, d) = \sum_{package(c)==p_0} CAS(c, d) \tag{3.2}$$

Algorithm 3.2 Incoming attack surface measurement for individual components: **inCAS**(c_0, d)

Require: Parameters: c_0 – the target component; d – Depth of assessment.
  System configurations:
  A map between each vulnerability v and its component component (v).
  A system wide dependency map (dependents of component c are dependOn(c)).
  Detected vulnerabilities V of the whole system.
Ensure: An incoming attack surface inCAS to component c_0.
  for each v in V **do**
    VAS(v, d) {Evaluate impacts of all vulnerabilities}
    if v.t containsKey (c_0) **then**
      $IF = v.t.get(c_0)$ {If c_0 is impacted by current vulnerability v, then retrieve its impact factor and cumulatively update the attack surface to c_0}
      $inCAS+ = IF$
    end if
  end for
  return inCAS

where $CAS(c, d)$ is calculated by Equation 3.1 and $package(c)$ obtains the software package where component c resides.

inCAS

The inCAS of a component is injected attack surface to this component by third party libraries. The inCAS value for each component is consistent with the number of vulnerable components it relies upon and the number of vulnerabilities on each of these components. Its calculation can be found at Algorithm 3.2. After each round of process, I validate the component list against the current component. If the current component is affected by the vulnerability, then its attack surface cumulatively updated, otherwise I skip to the next vulnerability. After process all vulnerabilities, the incoming attack surface to the component can be obtained.

inPAS

The inPAS of a package is the incoming package dependency based attack surface to this package. The calculation of inPAS is similar to that of PAS, where the incoming attack surface to each component of the package is added into its inPAS, as the following shows.

$$inPAS(p_0, d) = \sum_{package(c)==p_0} inCAS(c, d) \tag{3.3}$$

where $inCAS(c, d)$ can be solved by Algorithm 3.2, and as mentioned earlier at PAS calculation, $package(c)$ returns the package containing component c. inPAS can assist a system administrator to know to what extend that a package's attack surface is affected by third party applications.

System Attack Surface

SAS indicates the attack surface exposed to the whole system, by considering the dependency among all packages of the system. SAS can be used to compare package dependency based attack surfaces among different systems. This metric is helpful when a system administrator needs to know how the overall security status evolves or how to choose a relative secure image among a considerable amount of systems. The calculation of SAS can be like the following:

$$SAS(s_0, d) = \sum_{p \in installedPackages(s_0)} PAS(p, d) \tag{3.4}$$

where $PAS(p, d)$ can be obtained by Equation 3.2, and $installedPackages(s_0)$ returns all packages installed on the system s_0.

3.3 Experiments

In order to visualize the analysis of package dependency based attack surface, I do a number of experiments. I use a public image on Amazon EC2[9] as the target system, and adopt Java Runtime Environment (JRE) as the target library code, which is depended by three software packages in the system: `Weka`, `LibreOffice`, and `FreeHEP`. The system comes with JRE 1.6 update 17. I install other applications in order to calculate and compare different attack surfaces appropriately. The experiments are in four folds:

- Calculating outgoing attack surfaces from individual vulnerabilities (VAS) in JRE;
- Ranking vulnerable components (CAS) within a JRE;
- Calculating JRE outgoing attack surface (PAS) for the target system;
- Comparing system-wide package dependency based attack surface (SAS) with systems under two different configurations.

In the experiment, I focus on the study of variant attack surfaces injected by known vulnerabilities from JRE, therefore I do not calculate incoming attack surfaces of JRE and other packages. I note that a complete component level dependency graph is required to study both incoming and outgoing attack surfaces of all packages in a system. A system administrator should have the motivation of drawing such graphs for a particular system under investigation.

In the experiments, I select one public image in EC2 and run on two virtual machines (VM). The image comes with JRE 1.6 update 17, which is not the up-to-date version and has 134 known vulnerabilities according to National Vulnerability Database. I install two different sets of applications in the two virtual machines. There are 45 JRE vulnerabilities in the components that are depended by the installed applications. Table 3.4 shows the component distribution of these vulnerabilities.

[9] http://aws.amazon.com/ec2

Table 3.4: *JRE 1.6 update 17 vulnerable components with dependents*

Component	swing	AWT	networking	deserialization	imageIO	RMI	beans	2D	Concurrency
# of Vulnerabilities	6	4	10	1	2	2	2	17	1

In one VM, I install three JRE depended prevalent applications: `FreeHEP`, `Weka`, and `LibreOffice`. I calculate their vulnerability, component, and package outgoing attack surfaces within this VM. I then install only one JRE depended application (`FreeHEP`) to the other VM in order to compare the attack surfaces with the first VM at system level. Table 3.6, 3.7, and 3.5 show the details of the component dependency of these applications, where the component names are obtained from the the first level sub-folders under the main application folder.

3.3.1 Calculating Attack Surfaces for Individual Vulnerabilities (VAS)

I find out most of the JRE vulnerabilities are at high severity level – the average CVSS score is 7.7 out of 10 in the target JRE. Therefore the VAS value is dominated by the number of dependents of components. I observe that component `deserialization`, `swing`, `beans`, and `networking` are highly depended by the installed applications (See Table 3.2 for historical JRE vulnerabilities at component level and component level dependency between JRE and these installed applications). As a result, most vulnerabilities detected on these components have higher VAS values compared to the those with the lightly depended components, e.g., `imageIO`, `2d`, `rmi`, and `concurrency`. More detail can be found at column 1, 2 and 4 at Table 3.8.

3.3.2 Ranking Vulnerable Components (CAS) within a Package

Both component dependency and the number of vulnerabilities are considered while calculating the CAS of each component. Even though `deserialization` is the most depended component (see Table 3.2) and `2d` has largest number of vulnerabilities (column 6 at Ta-

Table 3.5: *`LibreOffice` Component Level Dependency: Each numeric value represents the imported times of the depended component by the dependent component.*

Dependent \ Depended	2d	beans	(de)serialization	networking	concurrency	imageIO	swing	xml
bean			5	1		2		
chart2			1					
unoxml			1					
toolkit 4		12	3				86	
l10ntools			6	1				
vcl			7					
extensions			1					
filter			14	4				11
linguistic			1					
writerfilter			2					
nlpsolver			3					
android			3		2			
sw			2					
connectivity			7	2				
javaunohelper			14	3				
xmlsecurity			14	1			28	6
package			4	1				
ridljar			4	5				
scripting		17	167	41			123	11
framework			10	1			2	
reportdesign			3	2				
forms			3					
sc			3					
stoc			4	3				
swext			4	14			4	
odk	2	1	42	10		1	86	
rhino				3			8	
wizards	2	11	5				17	8
qadevOOo			823	8		2	2	2
ure				1				
accessibility							5	
hsqldb					1			
sfx2			2					
testtools			5					
jurt			46	17				
dbaccess			6	2				
embeddedobj			5	4			2	
reportbuilder			49	6				7
languagetool			4					
bridges			5	1				
unotest			5					
ucb			2					
xmerge			305	6				46

Table 3.6: *`FreeHEP` Component Level Dependency: Each numeric value represents the imported times of the depended component by the dependent component.*

Dependent \ Depended	beans	(de)serialization	networking	swing	xml
util		5			
io		6			
menu	2	3	2	24	1
wbxml		14			

Table 3.7: *`Weka` Component Level Dependency: Each numeric value represents the imported times of the depended component by the dependent component.*

Dependent \ Depended	2d	beans	(de)serialization	networking	concurrency	imageIO	swing	xml	font
associations			22						
classifiers		5	108				36	2	
core		7	305	9			3	8	
estimators			7						
filters			18						
attributeSelection		5	22						
clusters		3	31				28		
datagenerators			8						
experiment		4	66	3			2		
gui	1	235	250	5	3	14	897		1

ble 3.8), neither of them has the largest CAS value. The experimental results indicate that component `swing` has the highest CAS value (column 5 at Table 3.8). This is because it is heavily depended by applications in the target system and has a considerable amount of vulnerabilities as well. Therefore, when considering attack surface at component level, component `swing` deserves more attention than others as it exposes the largest attack surface to the system.

3.3.3 JRE Outgoing Attack Surface (PAS)

The experiments only consider JRE as a vulnerable package with outgoing attack surface. Therefore I do not rank outgoing attack surfaces of different packages but only show the PAS of JRE. Based on PAS calculation formula at Eq. 3.2. the PAS of JRE is 3002.9. When multiple libraries exist on the system, the PAS can be used to compare exposed attack surfaces by those packages. Higher PAS indicates the need of more attention from system

Table 3.8: *Component Level Risk Rankings: Depended Components with no vulnerabilities are not shown*

Component	CVE Id	CVSS	VAS	CAS (Rank)	# of Vulnerabilities (Rank)
Deserialization	CVE-2012-5084	7.6	1159.76	1159.76 (2)	1 (7)
swing	CVE-2012-1716	10	307	1743.76 (1)	6 (3)
	CVE-2011-3549	10	307		
	CVE-2011-0871	10	307		
	CVE-2010-4465	10	307		
	CVE-2010-3557	6.8	208.76		
	CVE-2010-3553	10	307		
beans	CVE-2012-5086	10	143	286 (4)	2 (4)
	CVE-2012-4681	10	143		
networking	CVE-2011-3552	2.6	29.38	562.74 (3)	10 (2)
	CVE-2011-3547	5.0	56.5		
	CVE-2011-0867	5.0	56.5		
	CVE-2010-4448	2.6	29.38		
	CVE-2010-3574	10	113		
	CVE-2010-3573	5.1	57.63		
	CVE-2010-3560	2.6	29.38		
	CVE-2010-3551	5.0	56.5		
	CVE-2010-3549	6.8	76.84		
	CVE-2010-3541	5.1	57.63		
imageIO	CVE-2010-0846	7.5	75	150 (6)	2 (4)
	CVE-2010-0841	7.5	75		
2d	CVE-2012-5083	10	16	232.88 (5)	17 (1)
	CVE-2012-1531	10	16		
	CVE-2011-3551	9.3	14.88		
	CVE-2011-0873	10	16		
	CVE-2011-0868	5.0	8		
	CVE-2011-0862	10	16		
	CVE-2010-4471	5.0	8		
	CVE-2010-3571	10	16		
	CVE-2010-3567	10	16		
	CVE-2010-3566	10	16		
	CVE-2010-3565	10	16		
	CVE-2010-3562	10	16		
	CVE-2010-3556	10	16		
	CVE-2010-0849	7.5	12		
	CVE-2010-0848	7.5	12		
	CVE-2010-0847	7.5	12		
	CVE-2010-0838	7.5	12		
rmi	CVE-2011-3556	7.5	10.5	19.98 (7)	2 (4)
	CVE-2011-3557	6.8	9.48		
concurrency	CVE-2012-5069	5.8	7.54	7.54 (8)	1 (7)

administrators.

3.3.4 System Attack Surfaces (SAS)

We want to compare SAS values with systems running with different sets of installed applications. I launch two VMs in EC2: VM 1 runs JRE 1.6 update 17, with `FreeHEP`, `Weka`, and `LibreOffice` installed, and VM 2 runs the same JRE 1.6 update 17, but with only `FreeHEP` installed. I calculate the SAS values from JRE vulnerabilities based on Eq. 3.4[10]. As shown in Table 3.9, for VM 1, the SAS value is 3002.9, while for VM 2, the SAS value decreases to 305.76. This obviously confirms the significant difference of attack surfaces brought by a single vulnerable package (JRE) to systems with different configurations. The SAS value increases with more applications depending on vulnerable packages installed, which indicates an escalation on the security level of the depended libraries.

Table 3.9: *SAS for Two Different Systems on EC2*

	VM 1 (Vulnerable JRE with 3 dependents)	VM 2 (Vulnerable JRE with 1 dependent)
SAS	3002.9	305.76

3.3.5 Observations

The experimental results lead to several observations. First, dependency at component level dominates the value of vulnerability level attack surface (VAS). The source component of each vulnerability basically determines its attack surface to the whole system. Secondly, component level attack surface (CAS) is determined by both component dependency and the number of vulnerabilities from the component itself. Either counting the number of vulnerabilities or counting the number of its component dependents is a naive approach of calculating attack surface exposed by a given component. Last but not least, the number of dependents on vulnerable libraries of a system determines the attack surface value at

[10]the *installedPackages*() includes JRE and its dependents for the experiments, and only JRE has PAS value as others are not depended by any packages.

system level (SAS). The SAS value of the system increases as more applications installed which depend on vulnerable third-party libraries.

3.4 Limitations

There are limitations regarding the applicability of my methodology.

Dependency Information Retrieval: Not all package dependencies can be automatically captured. For accurate attack surface analysis, the component dependency graph of a system should be obtained, e.g., by system administrators, as certain packages are close source. Even for open source packages, they are not always written in the same programming language. JRE is a suitable example for us as many of its dependents are open source software written in Java. Besides, JRE has the largest number of known vulnerabilities among all library packages, according to the National Vulnerability Database. Once the component level dependency graph is available, my methodology could be generalized into other library packages.

Nested Dependency: By checking the dependency graph generated by `debtree`, I observe that in the target systems, even though there are packages depending on JRE's dependents, they all depend on JRE directly as well. Therefore I only count on the lowest level dependencies at my experiments, which dominate the attack surface metrics. That is, the experimental results are not affected by nested dependency. But this may not be true for other vulnerable depended packages in a system, especially when the dependency is objected-based instead of class-based.

Finer-grained Dependency Weighting: I treat each component dependency equally in my experiments. However, the dependency measurement may be finer-grained. Static code analysis or large scale data analysis may be helpful in gauging the dependency with a higher accuracy. Types of different dependencies [6, 77] may also affect exposed attack surfaces differently. Similarly, the accuracy of residual risk after each level of dependency

(the 'damping factor') may be further improved by large scale experiments as well.

3.5 Discussion

We define attack surface exposed through package dependency at vulnerability, component, package, and system levels, and propose a set of metrics to construct them. Besides outgoing attack surfaces, I propose algorithms calculating incoming attack surfaces injected through package dependency into individual components and packages. I validate the proposals and algorithms through experiments over JRE vulnerabilities in systems with different component dependency graphs. My approach provides systematic methodology to prioritize security tasks for system administrators, and provides inputs for choosing system images for application developers with multiple dependency options. Once researchers at software engineer domain have a finer-grained dependency weighting approach among software packages, the model could naturally borrow the dependency strength and thereafter provide more accurate risk metrics to users.

Chapter 4

Cloud Platform Risk Assessment

Previous chapters mostly focus on risk assessment under traditional network environment. Network modeling is mainly about capturing scenarios between a number of hosts. However, cloud computing has attracted a significant amount of users since its inception. Cloud specific threats need to be aware of as risk model on cloud environment has changed from traditional network surroundings.

Even though security bulletins have been setup by cloud provider (e.g. Amazon) to notify users about vulnerability information, previous experience has told us that significant effort is needed to bridge the gap between the provided service and current security situation. Specifically, I find that Amazon security bulletin usually releases critical vulnerability information more than two weeks later than original release date, e.g., by software vendors or community (cf. Section 4.1 for our study result). The exploit window could be even longer since there is no guarantee that every cloud user will and will be able to apply the update with the release, even though he has got notified. Also, a cloud provider may not be able to identify all known vulnerabilities on its platform. For known vulnerabilities, this attack window is way longer than it should be [11]. Besides, the prevalence of individual vulnerabilities has not been considered when publishing security bulletins. For example, Amazon only use CVSS score [63] to indicate the severity of vulnerabilities, which is indica-

tive for individual vulnerabilities on traditional in-house servers. However, threat from the prevalence of individual vulnerabilities should be re-evaluated under cloud environment. A prevalent image with known vulnerabilities can be instantiated by a large number of users in cloud, therefore it may generate large number of security holes for attackers. Attackers can do penetration test over public images, from where they can identify prevalent known vulnerabilities of running VMs and launch the same attack repeatedly to different instances. If the prevalent vulnerabilities indeed spread over the cloud, the attacker obtains an ideal cost-effective vehicle by exploiting the vulnerabilities to a large number of VMs. Therefore, with the new computing model of public cloud, it is even easier for attackers to launch attacks through prevalent vulnerabilities.

On the other side, cloud also provides an ideal venue to deploy defense mechanisms in large-scale. For example, with the homogeneous cloud environment, automatic patching becomes more efficient than in traditional in-house environment. A number of patching frameworks have been proposed towards known security holes in cloud [53, 114]. However, there is no empirical study and analysis on the cost and gain effectiveness of defending in cloud environment.

In this chapter, I will present my find about the attack cost effectiveness change over the IaaS cloud compared to traditional network environment. I empirically analyze the cost and effectiveness for exploiting known vulnerabilities under two different environments (traditional in-house and public IaaS cloud). I take AWS in my study since it has more publicly available information than other IaaS providers. I first identify with real data analysis that prevalent known vulnerabilities are very common in AWS AMIs, and demonstrate with real penetrations test that attack with these vulnerabilities is very trivial by malicious cloud users. I then statically analyze that both attack and patch are more cost-effective in cloud than under traditional environment. By statically I mean my analysis is over one time spot. To further investigate the relationship and strategy of attackers and defenders in cloud environment, I map these scenarios into a two-player game theoretic model. The

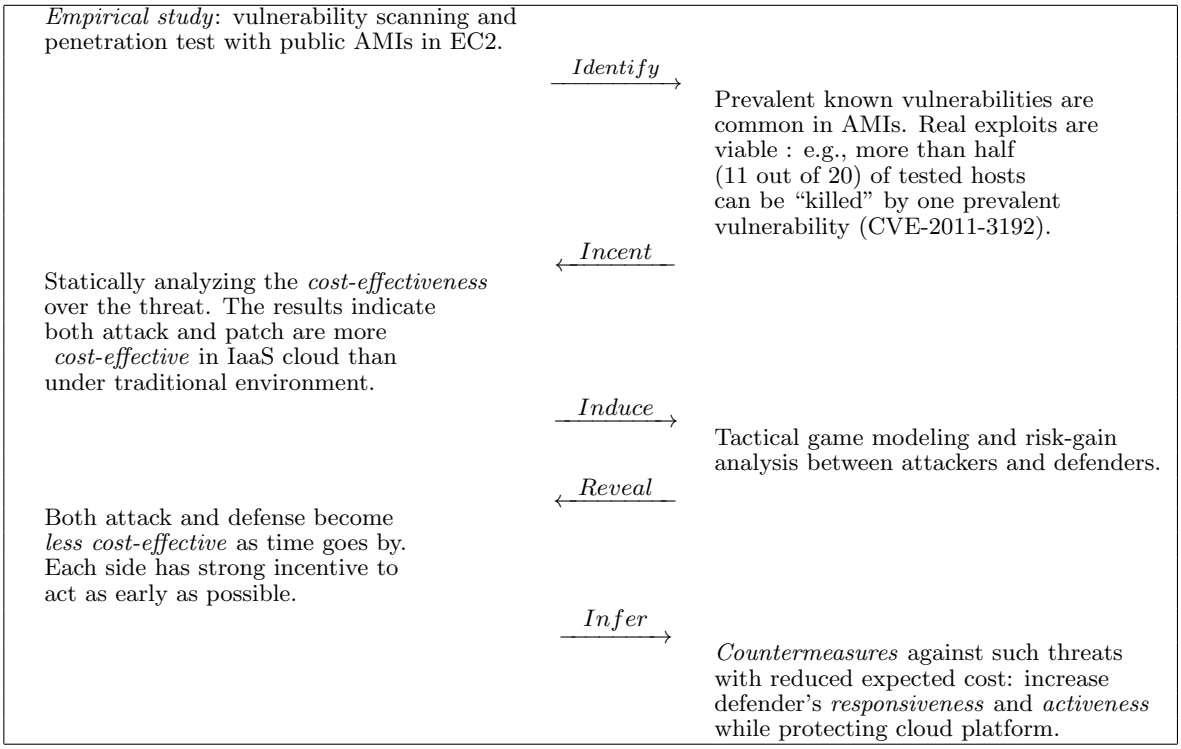


Figure 4.1: *Contribution Map.*

model indicates that the current security of public cloud is at the mercy of attackers. I then construct risk-gain analysis to simulate the evolution of the cost-effectiveness from defenders and attackers under different circumstances. The results show that cloud defender should be more responsive and proactive when hardening cloud platform as the attack surface increases dramatically compared to traditional computing environment. Moreover, our model illustrates that both attack and defense are more time-sensitive in cloud as they become less cost-effective as time goes by. I then propose countermeasures according to the evaluation results. Figure 4.1 is a contribution map of this chapter.

4.1 Empirical Study: Methodology and Finds

4.1.1 Background

Amazon EC2 I do experiments on public images over Amazon EC2. As a leading IaaS cloud provider, Amazon EC2 provides a platform by allowing different principals sharing their images publicly. Open source organizations like BitNami [1] and Ubuntu, IT companies such as Oracle and Amazon itself, and arbitrary number of individual contributors have published over 6,000 public images. Like potential attackers, I do penetration test over these images by launching corresponding VMs in order to analyze the weakness of running instances in the cloud.

Nessus Vulnerability Scanner Nessus [2] is a commercial vulnerability scanner developed from an open source product. It checks against configuration settings of a host and outputs a detailed report including security vulnerabilities, warnings, and system information, which can be from 50 to hundreds of pages. Therefore it is usually difficult for cloud administrators and users to read reports one by one in order to understand all security details in the cloud.

National Vulnerability Database (NVD) NVD [3] is an open database maintained by National Institute of Standards and Technology (NIST), which is regarded as one of the most comprehensive open vulnerability databases. Each entry in NVD is indexed by a Common Vulnerability Exposure(CVE), which is associated severity base score with a set of characteristics for that vulnerability. The base score is called "Common Vulnerability Scoring System (CVSS) base score" ranging from 0 to 10. The score indicates the overall severity of the vulnerability (the higher the worse). Vulnerability characteristics are called CVSS vectors which include access vectors and impact vectors. Access vectors contains Access Complexity (AC) indicating the difficulty level (High (H), Medium (M) or Low (L)) of exploiting the vulnerability, Access Vector (AV) indicating under what (Local (L),

[1] http://bitnami.org/
[2] http://www.tenable.com/products/nessus
[3] http://nvd.nist.gov/

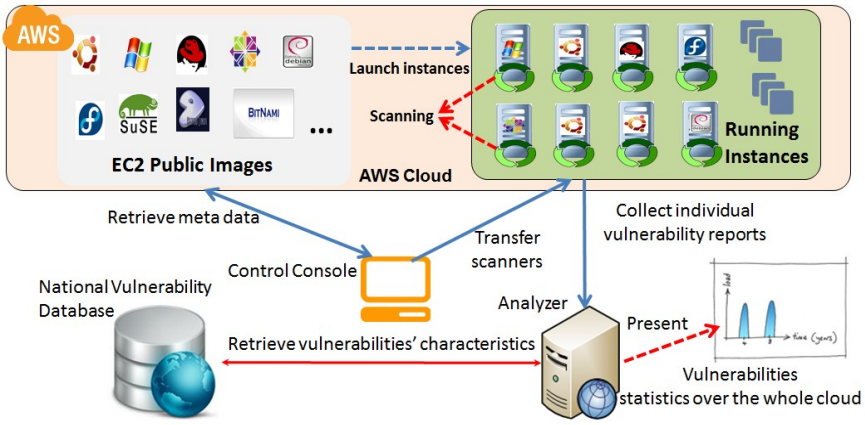


Figure 4.2: *Methodology of my empirical study.*

Adjacent Network (A), or Remote (R)) environment the vulnerability can be accessed and, authentication (Au) indicating required authentication times (Multiple (M), Singe (S), or None (N)) while exploiting. Impact vectors include the loss of confidentiality (C), integrity (I), and availability (A). There are three possible values associate with each impact metric: none (N), partial (P), or complete (C). For example, if a vulnerability's availability impact metric is complete, exploiting this vulnerability results in a completely availability loss on the target system. Besides CVSS metrics, NVD also provides affected applications, external references, and textual description regarding a given vulnerability.

4.1.2 Methodology

Penetration test over public images is a straightforward approach to identify prevalent vulnerabilities. Figure 4.2 illustrates my overall methodology. When scanning available public images on Amazon EC2, I first select a number of representative images to investigate, then launch one instance for each selected image. I then adopt a dedicated scanning server to transfer Nessus vulnerability scanner to each instance, and start scanning by running my script on each target instance [4]. After the scanning is complete, my script transfers all scanning reports to the scanning server. I retrieve the characteristics of each vulnera-

[4]Thanks for Amazon's approval for my scanning and penetration tests

bility by looking it up at the NVD. Based on the distribution of vulnerabilities and their characteristics, I obtain a single vulnerability report of all launched instances.

I launch and scan 80 public images in EC2. The selection of these images is based on the distribution of the operating system (OS) types and versions of public AMIs, with the assumption of the similar distribution of launched VM instances in the cloud.

4.1.3 What I Find

I summarize my finds in the following four aspects.

A considerable amount of prevalent vulnerabilities exist in AMIs

Similarly to what other researchers have found [10, 13], my scanning reveals a large number of vulnerabilities existing in public AMIs. Besides, I have identified several prevalent ones among all of these detected vulnerabilities. Table 4.1 lists the top prevalent vulnerabilities from my scanning. The prevalence indicates the probability of the vulnerability's existence among all images I have scanned. I find out that most (8 out 9) of them are critical vulnerabilities (with a CVSS score 7-10) by NVD standard, most (8 out of 9) of them can be accessed remotely, most (8 out 9) of them can be easily accessed, and most of them (7 out of 9) can be utilized by attackers to crush corresponding applications completely.

Attackers can identify prevalent vulnerabilities without scanning individual VM instances.

Amazon EC2 allows users to select public images based on platforms (OS types, versions, and pre-installed applications). Figure 4.3 shows the public images distribution based on OS types [5]. As we can see, more than half of the images are Ubuntu based. A closer look into the Ubuntu images indicates that more than half of them are either 10.04 or 12.04. Therefore under this circumstance, an attacker can keep monitoring newly released

[5]The data was collected in September 2012, which may change with new releases of AMIs.

Table 4.1: *Windows between Original Release and Amazon Announcement of Prevalent Known Vulnerabilities*

CVE	CVSS Base Score	Prevalence	Original Release	Amazon Announce	Attack Window in (days)
CVE-2012-4244	7.8	0.5915493	09/14/2012	09/28/2012	14
CVE-2012-3955	7.1	0.57746476	09/14/2012	N/A	> 26
CVE-2012-3817	7.8	0.52112675	07/25/2012	08/07/2012	13
CVE-2012-2807	10	0.49295774	09/07/2012	N/A	> 33
CVE-2012-2337	7.2	0.46478873	05/18/2012	07/30/2012	73
CVE-2011-3102	10	0.45070422	05/16/2012	N/A	> 117
CVE-2012-1033	5.0	0.45070422	02/08/2012	06/22/2012	135
CVE-2012-1667	8.5	0.45070422	06/05/2012	06/22/2012	17
CVE-2012-2110	7.5	0.33802816	04/19/2012	05/03/2012	15

vulnerabilities affecting these prevalent OSes and application frameworks. The attacker can also leverage known vulnerabilities that have not been patched by the publishers of the AMIs or the administrators of running instances, due to the patch window gap that I have observed in EC2 (explain shortly).

As a result, statistical analysis of OS and application distributions can help attackers in identifying the weaknesses and prevalent vulnerabilities in the cloud. This provides a scope of target victims and reduces the cost for large scale scanning and penetration. Attackers can roughly understand the overall potential weakness by simply noticing the latest vulnerabilities associated with the most prevalent OSes and applications installed in public images.

The patch window is long enough for attackers to exploit

I study several critical vulnerabilities and find that the gap between their original releases and Amazon's notifications is usually longer than two weeks (cf. Table 4.1). Attackers could easily launch 1-day exploit repeatedly in the entire cloud. The length of exploit window depends on the activities of cloud stakeholders (cloud provider and customers) such as the date of notification and their hardening and patching mechanisms. Moreover, not all known vulnerabilities can be easily detected by the cloud provider. As I have noticed, a

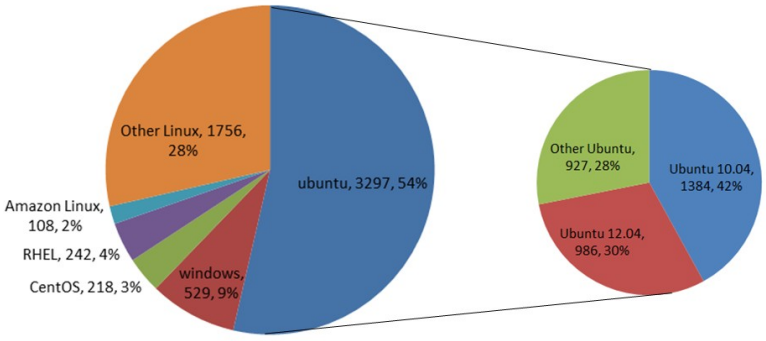


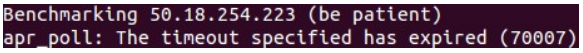
Figure 4.3: *Public images distribution by OS in Amazon EC2.*

large amount of exploitable vulnerabilities have not been notified by Amazon after a long time of their original releases. Therefore, attackers have enough time to prepare and launch attacks. Even worse, existing study has shown that more than 40% of small companies (under $50M revenue) do not have patch management [64] deployed in cloud, which consists of a considerable amount of current IaaS customers [3].

Running VMs in IaaS cloud offers more stable attack surfaces

VMs in IaaS cloud are more stable than traditional endpoints from an attacker' perspective. First of all, the IP range of each cloud provider is stable and can be predicted easily. Attackers could identify the location of their target VMs by playing several tricks [95]. Besides, a vast number of EC2 users are service providers with high availability requirement [3]. Therefore their applications and port configurations are relatively easy to detect. Attackers could reuse configuration information obtained previously to launch large scale attacks afterwards (for new vulnerabilities on the same or similar applications and systems). However, this does not work well under traditional in-house environments since the IP addresses of end hosts are changing more frequently, and most in-house servers are behind firewalls, and it is much more costly for an attacker to launch large scale attacks in order to locate a large number of victims under such heterogeneous environment.

100% of requests can be serviced before DoS attack.



The server stops responding after DoS attack.

Figure 4.4: *Benchmarking results of a server before and after launching apache killer*

4.1.4 Case Study: Penetration Testing on VMs in EC2

To confirm the viability of exploiting with prevalent vulnerabilities in EC2, I conduct a penetration test towards running VMs launched from vulnerable AMIs upon Amazon's approval. I first identify a prevalent vulnerability CVE-2011-3192, which is referred as "Apache Killer". I note that this vulnerability was not detected by Nessus in my scanning but it exists in 11 out of 20 AMIs [6] that I investigated with Ubuntu 10.04, most of which have been published for more than one year. Surprisingly, no security advisory on Amazon has been published for this vulnerability. I simply launch another instance in EC2 as an attacker with Metasploit [60] installed. By following the online instruction, I simply setup Metasploit with the number of packets sent to the target VM for DoS attack. I successfully crashed the Apache server running on all of the target VMs by sending 400 packets. The attack can be defended by running one line command (`sudo apt-get update`) to patch the vulnerability.

In order to verify the DoS attack, I use ApacheBench to test the response of the target server. As shown in Figure 4.4, before the exploit, all requests are served by the server; while after the attack, the ApacheBench could not receive any response, indicating the server is completely crashed.

I observe that the attack is very easy to launch with little interaction from the attacker.

[6] For safety reason I omit the AMI IDs here.

Therefore, crushing a large number of web service hosts is trivial from the attacker's perspective if any one of these vulnerable AMIs is widely used.

4.2 Static Cost-effectiveness Analyses

My empirical study has demonstrated that homogeneous settings in popular public cloud not only enhance the efficiency of computing power, but also bring new economic considerations for both attackers and defenders. Towards a first study on this, I do a comprehensive cost-effectiveness analysis by comparing exploiting prevalent vulnerabilities in public IaaS cloud and traditional in-house computing environments. While I refer a single *attacker* in both cases, a *defender* refers to service owners in traditional case and all cloud stakeholders (both cloud platform provider and cloud customers) in IaaS.

Assumptions: My analysis is based on the assumption that VM images are publicly available and used by cloud customers, but I do not require either each image or certain percentage of images are instantiated in the cloud. I further assume that prevalent types of images (OS types, versions, and application frameworks) are also prevalent in the VMs of the cloud.

Results: My analysis reveals that *both attack and defense are more cost-effective in cloud than in traditional in-house environment.* Attack surface under cloud environment has been enlarged with an increased density of potential victims. Moreover, attack cost has been decreased in cloud because the homogeneous nature of public cloud platforms reduces the effort required for target locating and vulnerability reconnaissance. On the other hand, cloud stakeholders (providers and customers) can manage patch with batch processing, which can patch larger attack surface per unit time than that in traditional environment.

4.2.1 Cost-effectiveness Analysis for Attacker

Cost of Attacker

A cyber attack usually involves the following costs [31, 91]: (1) locating target victims, (2) identifying vulnerabilities of victims, (3) choosing vulnerabilities , (4) obtaining exploits, and (5) dealing with defense mechanisms. For target victims in the cloud and traditional in-house environment, the costs (3) and (4) are the same. Therefore my analysis focuses on (1), (2), and (5), and my results indicate that IaaS cloud provides dramatically lower costs for attackers in these aspects.

Identifying victims. Under traditional environment, attackers could obtain target IP addresses in a straightforward way (e.g. by looking up DNS server). However, the external firewall deployed by most in-house servers may make the IP addresses untraceable. For certain types of threats like botnet or non-targeted DoS attacks by cyber terrorists, continuous (in terms of IP address) nodes with weak defending mechanisms but stable and high bandwidth are on the top of their target list. Consider that most bots in popular botnets such as "Conflicker" have small bandwidth only [90], I believe high quality bots in cloud are very appealing and can significantly increase the competitive strength of a bot master in botnet market, thus give strong incentive for attackers.

Consider a botnet master that needs to harvest N bots with a certain vulnerability v. Assuming for each reachable host, the probability of having v is ρ_v. Ideally, the search space of the vulnerable hosts under traditional environment is the whole IP address space (3,706,452,992), e.g., by generating random target IP addresses to exploit. Consider the factors that not every IP is assigned a host, and not each host is accessible, let δ_i be the probability that a single IP address is reachable in the Internet. Therefore the attacker needs to have at least $N/\rho_v\delta_i$ tries. However, under public IaaS cloud environment, the exploring range is significantly shrunk as the cloud provider offers the location and IP range publicly. For EC2, the total IP addresses is around 1,500,000 [5]. Besides, most of these IPs are located in a centralized manner as the IP addresses of VMs on the same data center are

usually assigned continuously [95]. With the high density of VMs running in a single data center, launching exploit to the cloud usually has much higher hit ratio δ_c. Therefore the attacker needs $N/\rho_v\delta_c$, where $\delta_c \gg \delta_i$, which indicates that the attacker needs dramatically less cost in cloud.

Identifying vulnerabilities. Under traditional environment shown in Figure 4.5, if the attacker wants to utilize known vulnerabilities to exploit a host, he may have to scan over the target machine, which can be easily blocked by firewalls. Researchers have proposed several passive scanning approaches in order to bypass IDS or firewall [31, 50], which may lower the scanning cost but still take a considerable amount of time and rely on some other assumptions (e.g., host administrators never modify packet headers).

On the other side, this vulnerability scanning cost can be reduced dramatically in public cloud environment (cf. Figure 4.6). As shown in my study in Amazon EC2, attackers could obtain the information of VM images (OS and applications installed) by browsing public image description pages. A brute force scanning on all images can help the attacker to decide the distributions of systems and applications in VMs, although in a rough manner. This information can reduce the cost to identify existing vulnerabilities of VMs running in the cloud. Furthermore, the attacker can keep tracking newly-released vulnerabilities associated with these prevalent OSes or applications in public images. Once a new vulnerability is released, it may exist on a large number of VMs in the cloud. Consider the usual patching window gap that I have observed in the last section, the attacker has plenty of time to develop and launch exploits, e.g., to harvest bots with vulnerable VMs. Therefore, identifying known vulnerabilities over the cloud is dramatically faster than that under traditional environment.

Dealing with hardening mechanisms of hosts. Customers on IaaS cloud usually have limited hardening support from the cloud provider, e.g,. Amazon EC2 only provides each instance an external firewall called security group, but no patching management. At the same time, a large number of cloud customers are small-sized service providers [64], and usually do not have strong motivation of hardening their systems as large companies. This

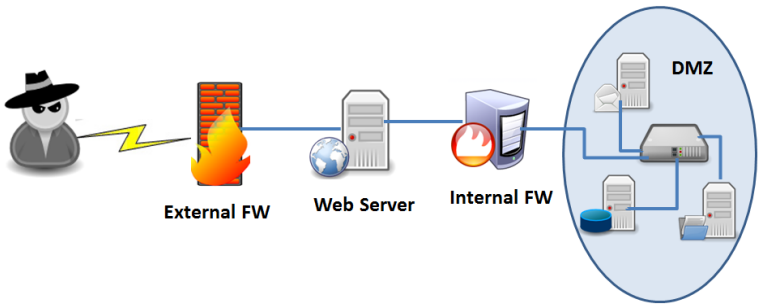


Figure 4.5: *Attacks under traditional environment*

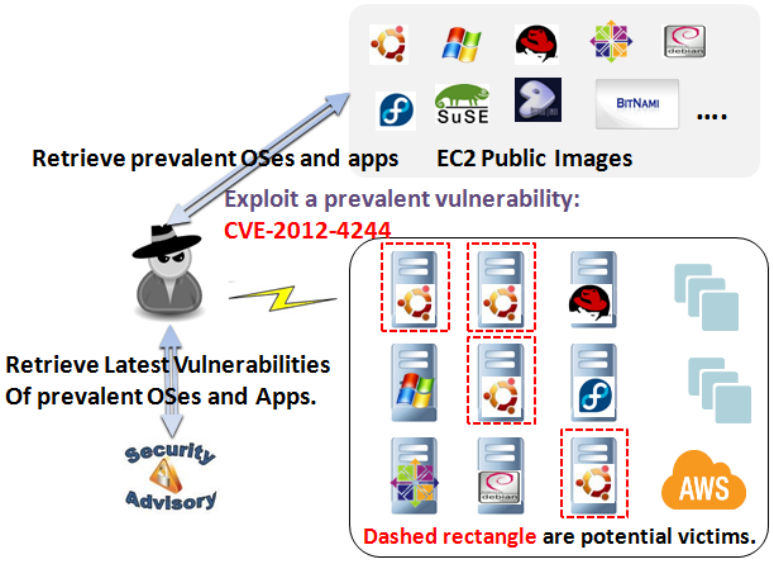


Figure 4.6: *Attacks under IaaS cloud*

results in a weak link for the cloud provider. Once an attacker has managed to exploit a prevalent vulnerability among the VMs of these small companies, a large scale of attack can result in loss for both the cloud customers and the cloud provider.

However, under traditional environment, an enterprise level service provider usually has dedicated team to maintain their platforms, which are usually hardened with several layers of firewalls (cf. Figure 4.5) in order to protect their data and infrastructure. The in-depth defense mechanisms increase the difficulty level for an attacker to compromise the server. It is extremely hard for an attacker to compromise a large number of hosts at the same time.

Therefore, I conjecture that compromising or bypassing hardening systems costs less under public cloud than that in traditional environment. Consider the cloud provider as a special service provider. Since it provides high flexibility of customizing infrastructure to its customers, its own defense mechanism is less tightly controlled compared to traditional in-house service providers, which makes it much easier to penetrate.

Gains of Attacker

An attacker could access confidential information for social or commercial benefits. Besides, the attacker could gain from the loss of his competitors by disrupting or disabling their services. These gains are the same under both cloud and traditional environment. One cloud specific gain is that upon compromising, high-quality bots on the cloud are denser than that in traditional computing environment with higher bandwidth and availability, which makes cyber terrorists easier to identify their targets.

Summary of Cost-effectiveness for Attacker

Considering similar gains of compromising a fixed set of hosts, the cost of the attacker is lowered by launching large scale attacks in an IaaS cloud, with lower costs in identifying enough number of vulnerable hosts, identifying exploiting vulnerabilities, and dealing with hardening mechanisms. Furthermore, exploiting prevalent vulnerabilities in the cloud usually brings the attacker more competing benefits with higher quality of bots than exploiting targets individually under traditional environment. Therefore, *the cost-effectiveness ratio for an attacker is lower in public cloud than that in traditional computing environment; that is, it is more economically efficient for an attacker to launch attacks in cloud.*

4.2.2 Cost-effectiveness Analysis for Defender

I refer the single term defender as all stakeholders that benefit from defending attacks, including the cloud provider and all of its customers. While facing attacks, the visible

cost paid by the defender is the hardening cost, and the gain is the loss of being exploited by attackers, or the commercial benefits from the services that otherwise are disrupted or disabled by attacks.

Costs for Defender

Hardening cost against known vulnerabilities is mainly from patching [85]. The cost per unit by patching in-house hosts is more pricey than batch patching over the cloud, since the batch processing lowers the hardening cost in cloud than in house servers [114].

Loss (or Gains) of Defender

Avoiding potential exploit effectiveness is the gain from the cloud provider's perspective. Exploiting effectiveness has a considerable overlap with an attacker's potential gains. Specifically, classical losses including that of service availability, data integrity, and confidentiality are the same for the defender in both cloud and traditional environment. Most of these losses are transferred to the attacker's benefit. However, there are cloud specific losses caused by large scale attacks, including neighborhood loss, user reputation loss for services, cloud provider reputation loss, and cloud utility misuse.

Neighborhood loss. As aforementioned, an attacker can lookup the IP range of a cloud provider's data center easily. The attacker could rent a VM and launch a large scale exploits to the VMs in the same data center. The attacker does not need to know the exact IP address of his target. Instead, all VMs on the same data center with the same vulnerability can be exploited. This expanded attack surface causes exponentially higher loss than that in traditional computing environment.

Reputation loss for cloud customers. Cloud customers usually are web service providers, and can lose their reputation from their own users upon being compromised. Even though this type of loss is invisible and indirect, it may completely affect the end users' confidence in continuing their services. Threats from prevalent vulnerabilities enlarge such fears as a

large number of services on the same cloud platform may exist.

Reputation loss of cloud provider. Even worse than user's reputation loss for cloud customers, the cloud provider's reputation can dramatically drop given a considerable amount of their VMs are compromised. Typically, the healthy including safety level of a cloud provider impacts the number of its users. A customer based survey [48] indicates that a cloud provider's reputation is the most important factor when a customer chooses which provider to go with.

Cloud utility misuse. Once an attacker has managed to deploy bots on one type of VMs in public cloud, he potentially could create a botnet with a large number of machines, which can be powerful enough for crushing other services over the Internet. This further enlarges the cloud provider's reputation loss. As for monetary loss, existing study has pointed out that a DDoS attack could cause up to $19M/hour loss for availability-sensitive services like E-banking. For each DDoS attack, the cost can be up to $100M [19].

Summary of Cost-effectiveness for Defender

The cloud provider can patch prevalent vulnerabilities with a cheaper unit cost than patching in-house servers individually. At the same time, the effectiveness of exploiting prevalent vulnerabilities in IaaS cloud is exponentially higher than the same attacks under traditional environment, consider much denser potential victims with the same vulnerabilities in cloud. Furthermore, the defender has extra cloud-specific losses such as cloud provider's reputation loss and cloud utility misuses. Therefore, my conclusion is that *the cloud defender has much lower cost-effectiveness ratio than in traditional computing environment, which indicates that with the same cost spent by the defender, he achieves more economic benefit in cloud.*

4.3 Tactical Game Modeling Between Attacker and Defender

Above cost-effectiveness analysis statically considers the costs and gains for both attackers and defenders. However, in real world several factors impact the relative costs and benefits of each side, and thus both rational attackers and defenders adjust their behaviors by considering these dynamic factors to achieve maximum benefits. Among these, the time-since-release has been considered as one of the main affecting factors that impacts the effectiveness of exploiting known vulnerabilities. This comes from an assumption that more VMs are patched for a given vulnerability as time goes by. Therefore, the sooner the attacker acts, the larger number of victim hosts can be hit with the same cost. On the other side, the sooner the defender acts, he can patch more VMs thus prevent more loss with lower cost. Moreover, patching a more prevalent vulnerability (by means of the vulnerability distribution in images and VMs) results in more cost-effectiveness ratio for both the attacker and defender, since it costs more for the attacker to identify vulnerable victims, and brings less gain for the defender to patch the vulnerability.

Therefore, I believe the dynamic cost-effectiveness ratios result in a game-based tactics between the attacker and defender. In this section, I construct a game theoretic model in order to illustrate the actions that rational attackers and defenders should take. I further map different cost-effectiveness scenarios into cost density functions to show their evolutions. The model indicates that both the attacker and defender have stronger incentive to act earlier, and their actions become less cost-effective as time goes by. After certain moment, the defender only needs to maintain the security level (the prevalence of the vulnerability) as the patching cost may exceed the cost from residual risk. The attacker may also lose the motivation of launching further attacks after certain point as the attack gain may not be able to compensate the attack cost due to the drop of the vulnerability prevalence. Therefore the threat from prevalent vulnerabilities can be greatly mitigated as long as the defender

patches security holes in a timely and proactive manner. However, cloud customers should be advised to protect their systems against targeted attacks as this is not a cloud specific threat.

4.3.1 Game Theory Background

An N-player game can be represented as a function G $(S_1, S_2, \ldots, S_N, u_1, u_2, \ldots, u_N)$, where S_i $(0 < i < N)$ is a strategy set $(s_{i1}, \ldots, s_{im})$ for player i, and s_j $(s_j \in S_i)$ is a complete strategy available for player i. Player i has a probability distribution $P_i = (p_{i1}, \ldots, p_{im})$, where p_{ik} is the probability of s_{ik} being adopted by player i. The payoff for player i is $u_i(S_1, \ldots, S_n)$ $(1 < j < n)$, where S_j is the strategy adopted by user j. For an N-player game theory, the expected payoff for player i is:

$$v_i(p_1, \ldots, p_n) = \sum_{m_1=1}^{M_1} \cdots \sum_{m_n=1}^{M_n} \left[\prod_{k=1}^{n} P_{km_k} \right] u_i(S_{1m_1}, \ldots, S_{nm_n}), \tag{4.1}$$

where M_j is the pure strategy numbers available to player j. For a 2-player game, the expected payoff of player 1 is:

$$v_1(p_1, p_2) = \sum_{m_1=1}^{M_1} \sum_{m_2=1}^{M_2} P_{1m_1} P_{2m_2} u_1(S_{1m_1}, S_{2m_2}), \tag{4.2}$$

where p_1 and p_2 are two sets of probability distributions adopted by the two players, respectively. Each distribution consists of a number of probabilities (sum up to 1), each of which indicates the chance of a strategy adopted by the player. S_{1m1} and S_{2m2} represent the strategies adopted by p_1 and p_2, respectively.

4.3.2 Game Theory Modeling

I consider player 1 as the attacker and player 2 as the defender. Player 1 has two strategies: attack (S_{11}) or stay idle (S_{12}). Player 2 also has two strategies: patching (S_{21}) or stay unpatched (S_{22}). P_{ij} indicates the probability of S_{ij} being adopted. I say that K_1 is a proactive action adopted by each player, meaning attack and patch for the attacker and the defender, respectively. K_2 means a passive action: stay idle for the attacker and leave the platform unpatched for the defender. Given each of the two players has two possible strategies, there are four conditions as follows.

- Both players choose K_1. The cloud defender needs to pay cost ($-CP$) in order to patch his platform in a timely manner. On the other side, the attacker has to pay the cost ($-AC$) of exploiting but without gaining from the hardened platform.
- When both players choose K_2, obviously both get 0.
- When the attacker chooses K_1 and the defender chooses K_2, the attacker gains ($+AG$) from exploiting by paying attack cost ($-AC$). The defender suffers the cost of being exploited ($-CD$).
- When the attacker chooses K_2 and the defender chooses K_1, the attacker gets 0 and the defender pays patch cost ($-CP$) to keep the platform up-to-date.

I use P_S and P_A to denote the probability of being proactive for the defender and the attacker, respectively. Given the four possible conditions, their expected payoffs (V_A and V_S) in the game are:

$$\begin{aligned} V_A &= -AC \times P_A P_S + 0 \times (1 - P_A) \times (1 - P_S) \\ &\quad + AG \times P_A \times (1 - P_S) + 0 \times (1 - P_A) \times P_S \\ &= AG \times P_A \times (1 - P_S) - AC \times P_A P_S \end{aligned} \tag{4.3}$$

$$
\begin{aligned}
V_S &= -CP \times P_A P_S + 0 \times (1 - P_A) \times (1 - P_S) \\
&\quad -CD \times P_A \times (1 - P_S) - CP \times (1 - P_A) \times P_S \\
&= -CD \times P_A \times (1 - P_S) - CP \times P_S
\end{aligned} \tag{4.4}
$$

The equations indicate that the expected payoffs of both players depend on both of their determinations of being proactive. Without exploiting intention, the attacker does not gain anything. When being more aggressive, he has an increased potential gain (when facing an unconscious defender) with the cost of launching attacks. A passive defender may end up losing nothing given the attacker is passive as well. However, this assumption is unrealistic as cyber attacks are ubiquitous. The defender (especially under cloud environment) should have a reasonable expectation on the density of attacks per unit time in order to balance the tradeoff between hardening cost and risk properly. Visualizing the game between the attacker and the defender can assist cloud stakeholders to better understand current security situation and make hardening plans accordingly.

4.3.3 Tactical Modeling between Attacker and Defender

We define risk as an instantiation of image with prevalent vulnerability. Risk density refers to the number of the risk per unit time. The relation between risk density from one vulnerability and time is mapped into an exponential function. The function can be expressed in Equation 4.5, where t is time and λ is the initial risk density. A larger λ means higher initial risk density of the vulnerability. As time goes by, the instantiation of the particular vulnerable image goes down proportionally as its popularity drops. After each time unit, the risk density becomes the product of the current risk density and e^{-d}, where d is a positive number. Therefore, the prevalence of the vulnerability determines the initial risk density

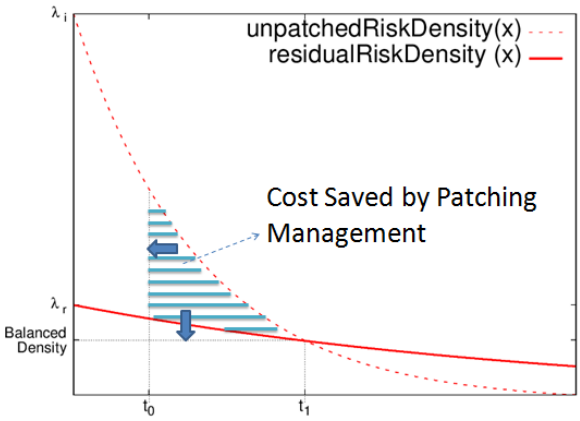


Figure 4.7: *Cost density distribution for cloud defender.*

value λ, and the exponential function represents the risk density trend over time.

$$f(t,\lambda) = \begin{cases} \lambda e^{-dt} & t \geq 0 \\ 0 & t < 0 \end{cases} \tag{4.5}$$

Tactical Modeling for Defender

Figure 4.7 illustrates the cost-effectiveness from the defender's perspective with two different strategies. The value of t_0 indicates the exploit window of a prevalent vulnerability. Therefore starting from t_0, the defender can choose to deploy patch to the vulnerability. Then the risk density is dropped to the patched risk density curve. A rational defender has a lower risk density because the sum of residual risk and patch cost is regarded lower than the unpatched risk cost density at the beginning. At the moment t_1, the two curves have a point of intersection. Starting from t_1, the defender only needs to maintain the security level. This is because the patch cost density has exceeded the residual risk density. The value of t_1 is can be calculated by Equation 4.6.

$$\begin{aligned} \lambda_1 e^{-d_1 t_1} &= \lambda_2 e^{-d_2 t_1} \\ t_1 &= \frac{\ln^{\lambda_i} - \ln^{\lambda_r}}{d_i - d_r} \end{aligned} \tag{4.6}$$

Therefore, the risk that could possibly be reduced by a rational defender can be expressed with Equation 4.7 and is marked in Figure 4.7, where λ_i and λ_r are initial risk density values of before and after the patch has been deployed. As indicated in Figure 4.7, minimizing the value of t_0 and initial risk density (λ_i) of the prevalent vulnerability can maximumly reduce the risk.

$$\begin{aligned} R(d_i, d_r) &= \int_{t_0}^{t_1} \lambda_i e^{-d_i t} dt - \int_{t_0}^{t_1} \lambda_r e^{-d_r t} dt \\ &= (1 - e^{-d_i t})\Big|_{t_0}^{t_1} - (1 - e^{-d_r t})\Big|_{t_0}^{t_1} \end{aligned} \tag{4.7}$$

Tactical Modeling for Attacker

The density of the attacker's potential gain through exploiting is similar as but slightly lower than the unpatched cloud loss due to being exploited, since some loss like neighborhood loss cannot be gained by the attacker. Therefore, the attacker's potential gain conforms to a exponential function as well. The attack cost has both maximum and minimum values given a fixed number of to-be exploited targets. The maximum value is paid while brute-forcing over the whole cloud platform (if there are not enough hosts that can be compromised in the whole platform) and the minimum cost is paid while exploiting each target with minimum cost (each attack succeeds at its first attempt). The cost is negatively correlated to the exploiting effectiveness because exploiting vulnerabilities with higher density costs less than utilizing sparsely distributed security holes. Therefore, the attack cost can be mapped to Equation 4.8. The function also has a base constant value B, which represents the basic cost of each attack, e.g., target reconnaissance and vulnerability detection. d_e refers to the

decaying factor of successful attack attempts. The chance of successfully exploiting a target machine is as small as the product of the original chance and e^{-d_e} after each time unit passed.

$$F(t, d_e) = \begin{cases} 1 - e^{-d_e t} + B & t \geq 0 \\ 0 & t < 0 \end{cases} \tag{4.8}$$

Figure 4.8 indicates the attacker's cost-effectiveness as time goes by. Consider a defender with strong security awareness which deploys patch at t_0, the attacker's gain is then decreased dramatically. If the defender is unconscious, attacks could last until t_2 as the gain through attacks after t_2 cannot compensate the attacker's cost of launching these exploits. Equations 4.9 indicates the attacker's gain by exploiting unpatched ($t_p = t_2$) and well hardened ($t_p = t_0$) platform, respectively. λ_g means the initial density of attacker's gain and λ_c means the initial attack cost density. In general the attacker loses the motivation of attacks after t_0 or t_2 whichever comes first. t_2 can be calculated through solving Equation 4.10. Given t_0 and t_2, the attacker's gain reduced by a rational defender can be obtained through Equations 4.9 and is the marked area in Figure 4.8.

$$\begin{aligned} G(d_g, d_c) &= \int_0^{t_p} \lambda_g e^{-d_g t} dt - \int_0^{t_p} (1 - e^{-d_c t}) dt - t_p \times B \\ &= (1 - e^{-d_g t})\Big|_0^{t_p} - (t + \frac{1}{d_c})\Big|_0^{t_p} \\ t_p &= \begin{cases} t_2 & t_0 \geq t_2 \\ t_0 & t_0 < t_2 \end{cases} \end{aligned} \tag{4.9}$$

$$\lambda e^{-d_g t_2} = (1 - e^{-d_c t_2}) \tag{4.10}$$

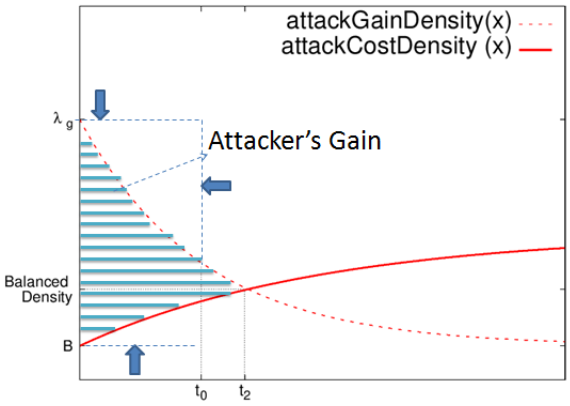


Figure 4.8: *Cost and gain density distribution for cloud attacker.*

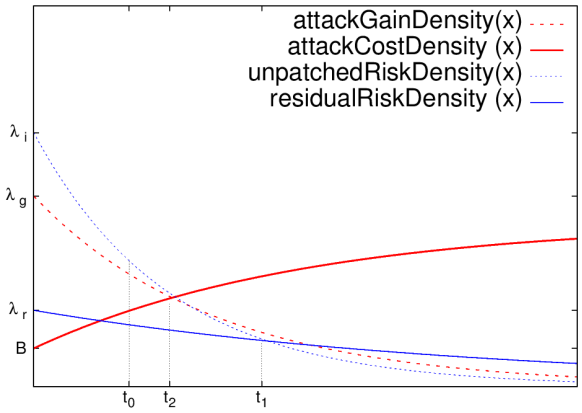


Figure 4.9: *Game between attacker and defender in cloud.*

Two-player Game

The game between the attacker and defender can be seen in Figure 4.9. At the starting point, both attack and defense are impacting. However, both of them become less cost-effective as time goes by. There are three noteworthy moments. t_2 is the moment when the attacker's cost and gain get balanced, after that a rational attacker stops launching attacks. t_1 is the moment when the cloud defender's cost is minimized and further patching costs higher than the residual risk. t_1 is not necessarily less than t_2. When $t_1 > t_2$, a rational attacker stops attacking but the cloud defender continues patching as the expected potential

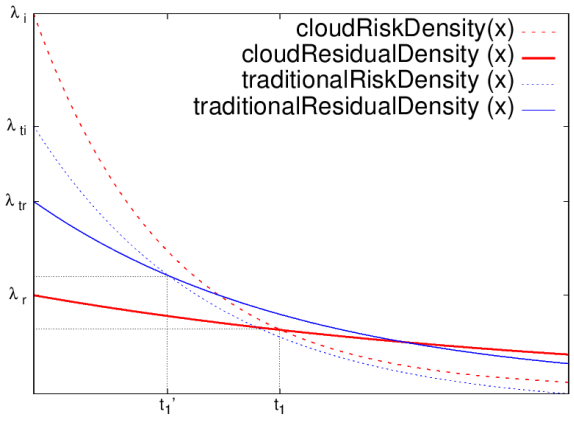


Figure 4.10: *Cost-effectiveness comparison between traditional and cloud environments.*

loss is greater than patching cost, and the cloud defender stops hardening its platform until t_2. When $t_1 \leq t_2$, the defender's cost and gain get balanced first, and he stops patching but maintains the security level. However, a rational attacker continues launching attacks under this circumstance as he can still obtain more than attack cost. When $t_1 \leq t_2$, we say there is a range for both the attacker and the defender can be satisfied. If $t_1 > t_2$, the attacker stops launching attacks as the expected gain cannot compensate attack cost. The defender only needs to maintain the security level under this circumstance. After t_0 the defender has patch available to the vulnerability. If the defender deploys patch, both t_1 and t_2 could arrive earlier up to t_0 because both exploiting effective density and attacker gain density drop. Therefore, not only could an smaller t_0 reduce the attacker's gain and exploiting effectiveness, it could also end the game between attacker and defender earlier.

Cost-effectiveness Comparison between Cloud and Traditional Computing Environment

In order to compare attacks under cloud and traditional computing environment, and investigate how much risk can be reduced by rational defenders, we model the cost-effectiveness from both the attacker's and the defender's perspective. We use the risk density functions (Equations 4.7 and 4.9) to answer the following questions:

- How much risk can be reduced by a rational defender in the cloud?
- How much risk can be reduced by a rational defender in traditional in-house environment?
- How much more gain can the attacker obtain when facing an unconscious cloud defender than a rational defender?
- How much more gain can the attacker obtain when facing an unconscious defender than a rational defender in traditional environment?

In Section 4.2 we have analyzed that prevalent vulnerabilities lead to lower patching expense but higher potential loss in cloud than under traditional environment. Therefore under the cloud environment, the risk density is higher but the patch cost density is lower than those under traditional environment. Figure 4.10 illustrates the comparison. For cloud environment, the initial risk and cost density value before and after patch deployed are denoted by λ_i and λ_r. While λ_{ti} and λ_{tr} represent initial risk and cost density under traditional environment (before and after patch deployed, respectively). As we can see, the defender in the cloud can achieve a better stable security level than in traditional because of the lower cost in patching. The time t_1' does not need to be greater than t_1. If the cloud defender can handle hardening work appropriately, it can achieve a lower security level with shorter period of time. Figure 4.10 also tells us that it is urgent for the cloud defender to harden the cloud platform as the gap between the potential loss and hardening cost is dramatically enlarged compared to that under traditional environment. Without patch deployed, the potential loss in the cloud is exponentially higher , i.e., $R(\lambda_i, \lambda_r) \gg R(\lambda_{ti}, \lambda_{tr})$.

4.3.4 Summary

Through dynamic cost-effectiveness analysis with gaming modeling, we have observed that both attack and defense are more cost-effective in the cloud, and they become less cost-effective as time goes by. Three factors determines the game between the two parties: the

defender's willingness (P_S), responsiveness (t_0), and activeness (λ_r). In a nutshell, the cloud defender should be willing to harden its cloud platform in a timely and proactive manner.

4.4 Countermeasures

My gaming modeling indicates that in order to minimize attacker's gain, which is the cost marked in Figure 4.8, three parameters (t_0, λ_g and B) can be tuned. This means that the defender should be more responsive and proactive to known vulnerabilities. Minimizing t_0 means eliminating 1-day exploits, i.e., keeping all instances up-to-date and maintaining running instances without severe known security holes. Minimize λ_g, which means reduce the risk density at the beginning (or as soon as possible). Increase B, which means increase the base cost of each attempt of attack. Based on the three directions. We propose the following three countermeasures for cloud providers and customers.

Patching public VM images. Public VM images like AMIs should be up-to-date when being launched by users. The cloud provider should setup policy in order to make sure unpatched images should not be launched. Two options are available in order to achieve this requirement: the provider can either force a user to update the image before it is launched, or the provider or image publishers can update public images offline periodically [114]. This could reduce the boot time at user end during launching. This countermeasure will reduce the value of t_0 in Figure 4.8.

Maintaining running instances. Every time when launching an image, the image should be required to check against a configuration file in order to make sure all default apps are up to date. The configuration file can be provided by the image publisher. Cloud users can customize the configuration file on their own. Cloud users should also be responsible for the applications installed by themselves. Similar to the above countermeasure, this will also reduce the value of t_0 in Figure 4.8.

Give patching priority to prevalent vulnerabilities. As Figure 4.7 indicates, higher

prevalence is more time sensitive. Therefore, considering the prevalence (along with impact factors) of vulnerabilities is needed when making patching plans. The moment (t_d) of deploying a patch to a specific vulnerability can be inferred from a preset threshold of tolerable risk. Given similar impact, a higher prevalent vulnerability usually has a smaller t_d. This countermeasure will also reduce the initial risk density (λ_g) in Figure 4.8.

Shuffling cloud infrastructure smartly. Introduce a new defensive mechanism, which could make the configuration of the cloud platform as an animation rather than a static picture. Similar to patching vulnerabilities periodically, configurations (e.g.IP address, topology or applications) can be changed time to time. This type of moving target defense paradigms [23, 97, 115, 116] could significantly mitigate security holes on the cloud. It could also increase the base cost for each attempt of attack (value of B) in Figure 4.8.

Secure data in the cloud. More hardening approaches like seamless encryption in the cloud [55, 102, 104, 113]. Attacker will be slowed down as less information can be obtained after attacker compromised one instances in the cloud. Also attacker aims at stealing sensitive data will become less motivated because decrypting data will be a nontrivial task after he compromise the target machine. Therefore this countermeasure will also increase the base cost per each attack.

4.5 Discussions

I identify the threat of exploiting prevalent vulnerabilities in IaaS cloud with an empirical study and real penetration test in Amazon EC2. I pinpoint that such threat exponentially increases the risk level of cloud due to two factors: the prevalent vulnerabilities can spread quickly on public cloud as one image could potentially be instantiated by a large number of users, and the nature of the cloud enables more cost-effective attacks than traditional in-house computing environment. I analyze the cost and effectiveness of exploiting and defending prevalent vulnerabilities under traditional and cloud environments. The results

indicate that both cloud attackers and defenders have lower cost-effectiveness ratio, which enables a game-like tactical scenario between them. To further illustrate the influence of dynamic cost-effectiveness nature, I build a 2-player game theoretic model and a risk-gain analysis to capture the risks associated with two types (rational or unconscious) of attackers and defenders. The result reveals that both attack and defense become less cost-effective in cloud as time goes by, which suggests the defender should be more responsive and proactive under cloud environment. I stir up them with a number of possible countermeasures against such threat. However, the following limitations need to be overcome in order to provide better results.

First of all, I only consider prevalent vulnerabilities in large-scale cloud. I do not provide an aggregated metric indicating the security level of the whole cloud platform and individual VM systems. Cloud stakeholders can use my model to further calculate a metric for finer-grained modeling. The calculation can be based on environmental factor or impact factor of identified vulnerabilities.

Secondly, I do not have accurate statistics regarding cloud customers' system information of running VMs, e.g., exactly how many VMs are launched for a particular image, and how many users usually patch their VMs in timely manner. In the model I take a simple estimation by assuming that the prevalence of instances is similar to that of public images.

Chapter 5

Zero-day Risk Assessment

All of the previous chapters focus on known vulnerabilities evaluation. Known vulnerabilities are obvious and their characteristics are easy to be retrieved. However, the risk from unknown vulnerabilities cannot be ignored. Each year a large number of new software vulnerabilities are discovered in various applications (see Figure 5.1). Moreover, A zero-day vulnerability could last a long period of time (*e.g.* in 2010 Microsoft confirmed a vulnerability in Internet Explorer, which affected some versions that were released in 2001). Therefore, in order to have more accurate results on network security evaluation, one must consider the effect from zero-day vulnerabilities. The National Vulnerability Database (NVD) is a well-known data source for vulnerability information, which could be useful to estimate the likelihood that a specific application contains zero-day vulnerabilities based on historical information.

In this chapter, I will present my empirical experience of applying data-mining techniques on NVD data in order to build a prediction model for an unknown risk metric per application - Time to Next Vulnerability (TTNV). I conduct a rigorous data analysis and experiment with a number of feature construction schemes and learning algorithms. The results show that the data in NVD generally have poor prediction capability, with the exception of a few vendors and software applications. In the rest of the chapter I will explain the features

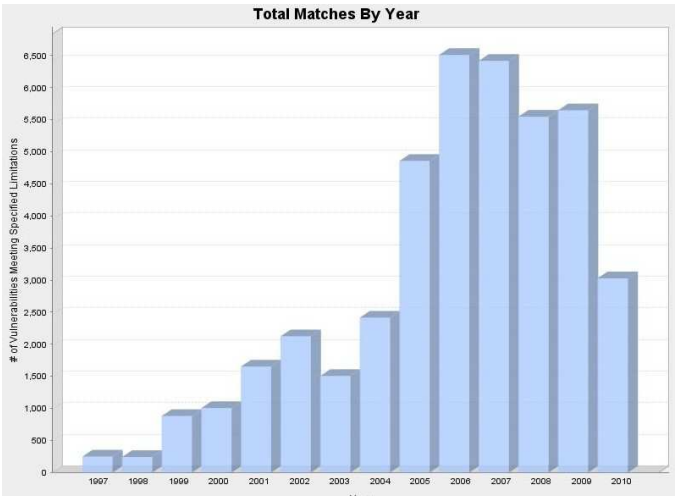


Figure 5.1: *The trend of vulnerability numbers*

I have constructed and the various approaches I have taken in my attempts to build the prediction model. While it is generally a difficult task to show that data has no utility, my experience does indicate a number of reasons why it is unlikely to construct a reliable prediction model for TTNV given the information available in NVD.

5.1 Data Source – National Vulnerability Database

Each data entry in NVD consists of a large number of fields. I represent them as <D, CPE, CVSS>. D is a set of data including published time, summary of the vulnerability and external links about each vulnerability. CPE [14] and CVSS [63] will be described below.

5.1.1 CPE (Common Platform Enumeration)

CPE is *an open framework for communicating the characteristics and impacts of IT vulnerabilities.* It provides us with information on a piece of software, including version, edition, language, *etc.* An example is shown below:

```
cpe:/a:acme:product:1.0:update2:pro:en-us
```

```
Professional edition of the "Acme Product 1.0 Update 2 English".
```

5.1.2 CVSS (Common Vulnerability Scoring System)

CVSS is a vulnerability scoring system designed to provide an open and standardized method for rating IT vulnerabilities. CVSS helps organizations prioritize and coordinate a joint response to security vulnerabilities by communicating the base, temporal and environmental properties of a vulnerability. Currently NVD only provides the Base group in its metric vector. Some components of the vector are explained below.

- *Access Complexity* indicates the difficulty level of the attack required to exploit the vulnerability once an attacker has gained access to it. It includes three levels: High, Medium, and Low.
- *Authentication* indicates whether an attacker must authenticate in order to exploit a vulnerability. It includes two levels: Authentication Required (R), and Authentication Not Required (NR).
- *Confidentiality, Integrity and Availability* are three loss types of attacks. Confidentiality loss means information will be leaked to people who are not supposed to know it. Integrity loss means the data can be modified illegally. Availability loss means the compromised system cannot perform its intended task or will crash. Each of the three loss types have the three levels: None (N), Partial (P), and Complete (C).

The *CVSS Score* is calculated based on the metric vector, with the objective of indicating the severity of a vulnerability.

5.2 My Approach

I choose TTNV (time to next vulnerability) as the predicted feature. The predictive attributes are time, versiondiff (the distance between two different versions by certain measurement), software name and CVSS. All are derived or extracted directly from NVD.

5.2.1 Data Preparation and Preprocessing

Division of training/test data:

As the prediction model is intended to be used to forecast future vulnerabilities, I divide the NVD data into training and test data sets based on the time the vulnerabilities were published. The ratio of the amount of training to test data is 2. I chose to use the data starting from 2005, as the data prior to this year looks unstable (see Figure 5.1).

Removing obvious errors:

Some NVD entries are obviously erroneous (*e.g.* in one entry for Linux the kernel version was given as 390). To prevent these entries from polluting the learning process, I removed them from the database.

5.2.2 Feature Construction and Transformation

Identifying and constructing predictive features is of vital importance to data-mining. For the NVD data, intuitively Time and Version are two useful features. As we want to predict time to next vulnerability, the published time for each past vulnerability will be a useful information source. Likewise, the version information in each reported vulnerability could indicate the trend of vulnerability discovery, as new versions are released. Although both Time and Version are useful information sources, they need to be transformed to provide the expected prediction behavior. For example, both features in their raw form increase monotonically. Directly using the raw features will provide little prediction capability for future vulnerabilities. Thus, I introduce several feature construction schemes for the two fields and studied them experimentally.

Time:

I investigated two schemes for constructing time features. One is epoch time, the other is using month and day separately without year. Like explained before, the epoch time is unlikely to provide useful prediction capability, as it increases monotonically. Intuitively, the second scheme shall be better, as the month and day on which a vulnerability is published may show some repeating pattern, even in future years.

Version:

I calculate the difference between the versions of two adjacent instances and use the versiondiff as a predictive feature. An instance here refers to an entry where a specific version of an application contains a specific vulnerability. The rationale for using versiondiff as a predictive feature is that we want to use the trend of the versions with time to estimate future situations. Two versiondiff schemas are introduced in my approach. The first one is calculating the versiondiff based on version counters (rank), while the second is calculating the versiondiff by radix.

Counter versiondiff: For this versiondiff schema, differences between minor versions and differences between major versions are treated similarly. For example, if one software has three versions: 1.1, 1.2, 2.0, then the versions will be assigned counters 1, 2, 3 based on the rank of their values. Therefore, the versiondiff between 1.1 and 1.2 is the same as the one between 1.2 and 2.0.

Radix-based versiondiff: Intuitively, the difference between major versions is more significant than the difference between minor versions. Thus, when calculating versiondiff, we need to assign a higher weight to relatively major version changes and lower weight to relatively minor version changes. For example, for the three versions 1.0, 1.1, 2.0, if we assign a weight of 10 to the major version and a weight of 1 to each minor version, the versiondiff between 1.1 and 1.0 will be 1, while the versiondiff between 2.0 and 1.1 will be 9.

When analyzing the data, I found out that versiondiff did not work very well for the problem because, in most cases, the new vulnerabilities affect all previous versions as well. Therefore, most values of versiondiff are zero, as the new vulnerability instance must affect an older version that also exists in the previous instance, thus, resulting in a versiondiff of zero. In order to mitigate this limitation, I created another predictive feature for my later experiments. The additional feature that I constructed is the number of occurrences of a certain version of each software. More details will be provided in Section 5.3.

5.2.3 Machine Learning Functions

I used either classification or regression functions for the prediction, depending on how I define the predicted feature. The TTNV could be a number representing how many days we need to wait until the occurrence of the next vulnerability. Or it could be binned and each bin stands for values within a range. For the former case, I used regression functions. For the latter case, I used classification functions. I used WEKA [12] implementations of machine learning algorithms to build predictive models for the data. For both regression and classification cases, I explored all of the functions compatible to our data type, with default parameters. In the case of the regression problem, the compatible functions are: linear regression, least median square, multi-layer perceptron, RBF network, SMO regression, and Gaussian processes. In the case of classification, the compatible functions are: logistic, least median square, multi-layer perceptron, RBF network, SMO, and simple logistic.

5.3 Experimental Results

I conducted the experiments on my department's computer cluster - Beocat. I used a single node and 4G RAM for each experiment. As mentioned above, WEKA [12], a data-mining suite, was used in all experiments.

5.3.1 Evaluation Metrics

A number of metrics are used to evaluate the performance of the predictive models learned.

Correlation Coefficient:

The correlation coefficient is a measure of how well trends in the predicted values follow trends in actual values. It is a measure of how well the predicted values from a forecast model "fit" the real-life data. The correlation coefficient is a number between -1 and 1. If there is no relationship between the predicted values and the actual values, the correlation coefficient is close to 0 (i.e., the predicted values are no better than random numbers). As the strength of the relationship between the predicted values and actual values increases, so does the correlation coefficient. A perfect fit gives a coefficient of 1.0. Opposite but correlated trends result in a correlation coefficient value close to -1. Thus, the higher the absolute value of the correlation coefficient, the better; however, when learning a predictive model, negative correlation values are not usually expected. I generate correlation coefficient values as part of the evaluation of the regression algorithms used in my study.

Root Mean Squared Error:

The mean squared error (MSE) of a predictive regression model is another way to quantify the difference between a set of predicted values, x_p, and the set of actual (target) values, x_t, of the attributed being predicted. The root mean squared error (RMSE) can be defined as:

$$\text{RMSE}(x_p,\, x_t) = \sqrt{MSE(x_p, x_t)} = \sqrt{E[(x_p - x_t)^2]} = \sqrt{\frac{\sum_{i=1}^{n}(x_{p,i} - x_{t,i})^2}{n}}$$

Root Relative Squared Error:

According to [1], the root relative squared error (RRSE) is relative to what the error would have been if a simple predictor had been used. The simple predictor is considered to be the mean/majority of the actual values. Thus, the relative squared error takes the total squared

error and normalizes it by dividing by the total squared error of the simple predictor. By taking the root of the relative squared error one reduces the error to the same dimensions as the quantity being predicted.

$$\text{RRSE}(x_p,\, x_t) = \sqrt{\frac{\sum_{i=1}^{n} (x_{p,i} - x_{t,i})^2}{\sum_{i=1}^{n} (x_{t,i} - \bar{x})^2}}$$

Correctly Classified Rate:

To evaluate the classification algorithms investigated in this work, I use a metric called correctly classified rate. This metric is obtained by dividing correctly classified instances by all instances. Obviously, a higher value suggests a more accurate classification model.

5.3.2 Experiments

I performed a large number of experiments, by using different versiondiff schemes, different time schemes, and by including CVSS metrics or not. For different software, different feature combinations produce the best results. Hence, I believe it is not effective to build a single model for all the software. Instead, I build separate models for different software. This way, I also avoid potential scalability issues due to the large number of nominal type values from vendor names and software names.

Given the large number of vendors in the data, I did not run experiments for all of them. I focused especially on several major vendors (Linux, Microsoft, Mozilla and Google) and built vendor-specific models. For three vendors (Linux, Microsoft and Mozilla), I also built software-specific models. For other vendors (Apple, Sun and Cisco), I bundled all their software in one experiment.

5.3.3 Results

Linux:

I used two versiondiff schemes, specifically counter-based and radix-based, to find out which one is more effective for the model construction. I also compared two different time schemes (epoch time, and using month and day separately). In a first set of experiments, I predicted TTNV based on regression models. In a second set of experiments, I grouped the predictive feature (TTNV) values into bins, as I observed that the TTNV distribution shows several distinct clusters, and solved a classification problem.

Table 5.1 shows the results obtained using the *epoch time* scheme versus the results obtained using the *month and day* scheme, in terms of correlation coefficient, for regression models. As can be seen, the results of the experiments did not show a significant difference

Table 5.1: *Correlation Coefficient for Linux Vulnerability Regression Models Using Two Time Schemes*

Regression Functions	Epoch time		Month and day	
	training	test	training	test
Linear regression	0.3104	0.1741	0.6167	-0.0242
Least mean square	0.1002	0.1154	0.1718	0.1768
Multi-layer perceptron	0.2943	0.1995	0.584	-0.015
RBF network	0.2428	0	0.1347	0.181
SMO regression	0.2991	0.2186	0.2838	0.0347
Gaussian processes	0.3768	-0.0201	0.8168	0.0791

between the two time schemes that I used, although I expected the month and day feature to provide better results than the absolute epoch time, as explained in Section 5.2.2. Thus, neither scheme has acceptable correlation capability on the test data. I adapted the month and day time schema for all of the following experiments.

Table 5.2 shows a comparison of the results of the two different versiondiff schemes. As can be seen, both perform poorly as well. Given the unsatisfactory results, I believed that the large number of Linux sub-versions could be potentially a problem. Thus, I also investigated constructing the versiondiff feature by binning versions of the Linux kernel (to

Table 5.2: *Correlation Coefficient for Linux Vulnerability Regression Models Using Two Versiondiff Schemes*

Regression Functions	Version counter		Radix based	
	training	test	training	test
Linear regression	0.6167	-0.0242	0.6113	0.0414
Least mean square	0.1718	0.1768	0.4977	-0.0223
Multi-layer perceptron	0.584	-0.015	0.6162	0.1922
RBF network	0.1347	0.181	0.23	0.0394
SMO regression	0.2838	0.0347	0.2861	0.034
Gaussian processes	0.8168	0.0791	0.6341	0.1435

obtained a smaller set of sub-versions). I round each sub-version to its third significant major version (*e.g.* Bin(2.6.3.1) = 2.6.3). I bin based on the first three most significant versions because more than half of the instances (31834 out of 56925) have version longer than 3, and Only 1% (665 out of 56925) versions are longer than 4. Also, the difference on the third subversion will be regarded as a huge dissimilarity for Linux kernels. I should note that the sub-version problem may not exist for other vendors, such as Microsoft, where the versions of the software are naturally discrete (all Microsoft products have versions less than 20). Table 5.3 shows the comparisons between regression models that use binned versions versus regression models that do not use binned versions. The results are still not good enough as many of the versiondiff values are zero, as explained in Section 3.2 (new vulnerabilities affect affect previous versions as well).

Table 5.3: *Correlation Coefficient for Linux Vulnerability Regression Models Using Binned Versions versus Non-Binned Versions*

Regression Functions	Non-binned versions		Binned versions	
	training	test	training	test
Linear regression	0.6113	0.0414	0.6111	0.0471
Least mean square	0.4977	-0.0223	0.5149	0.0103
Multi-layer perceptron	0.6162	0.1922	0.615	0.0334
RBF network	0.23	0.0394	0.0077	-0.0063
SMO regression	0.2861	0.034	0.285	0.0301
Gaussian processes	0.6341	0.1435	0.6204	0.1369

TTNV Binning: Since I found that the feature (TTNV) of Linux shows distinct clusters, I divided the feature values into two categories, more than 10 days and no more than 10 days, thus transforming the original regression problem into an easier binary classification problem. The resulting models are evaluated in terms of corrected classified rates, shown in Table 5.4. While the models are better in this case, the false positive rates are still high (typically above 0.4). In this case, as before, I used default parameters for all classification functions. However, for the SMO function, I also used the Gaussian (RBF) kernel. The results of the SMO (RBF kernel) classifier are better than the results of most other classifiers, in terms of correctly classified rate. However, even this model has a false positive rate of 0.436, which is far from acceptable.

Table 5.4: *Correctly Classified Rates for Linux Vulnerability Classification Models Using Binned TTNV*

Classification Functions	Correctly classified		FPR	TPR
	training	test		
Simple logistic	97.6101%	69.6121%	0.372	0.709
Logistic regression	97.9856%	57.9542%	0.777	0.647
Multi-layer perceptron	98.13%	64.88%	0.689	0.712
RBF network	95.083%	55.18%	0.76	0.61
SMO	97.9061%	61.8259%	0.595	0.658
SMO (RBF kernel)	96.8303%	62.8392%	0.436	0.641

CVSS Metrics: In all cases, I also perform experiments by adding CVSS metrics as predictive features. However, I did not see much differences.

Microsoft:

As we have already observed the limitation of versiondiff scheme in the analysis of Linux vulnerabilities, for Microsoft instances, I use only the number of occurrences of a certain version of a software or occurrences of a certain software, instead of using versiondiff, as described below. I analyzed the set of instances and found out that more than half of the instances do not have version information. Most of these case are Windows instances. Most

of the non-Windows instances (more than 70%) have version information. Therefore, I used two different occurrence features for these two different types of instances. For Windows instances, I used the occurrence of each software as a predictive feature. For non-Windows instances, I used the occurrence of each version of the software as a predictive feature.

Also based on my observations for Linux, I used only the month and day scheme, and did not use the epoch time scheme in the set of experiments I performed for Windows. I analyzed instances to identify potential clusters of TTNV values. However, I did not find any obvious clusters for either windows or non-windows instances. Therefore, I only used regression functions. The results obtained using the aforementioned features for both Windows and non-Windows instances are presented in Table 5.5. As can be seen, the correlation coefficients are still less than 0.4.

Table 5.5: *Correlation Coefficient for Windows and Non-Windows Vulnerability Regression Models, Using Occurrence Version/Software Features and Day and Month Time Scheme*

Regression Functions	Win Instances		Non-win Instances	
	training	test	training	test
Linear regression	0.4609	0.1535	0.5561	0.0323
Least mean square	0.227	0.3041	0.2396	0.1706
Multi-layer perceptron	0.7473	0.0535	0.5866	0.0965
RBF network	0.1644	0.1794	0.1302	-0.2028
SMO regression	0.378	0.0998	0.4013	-0.0467
Gaussian processes	0.7032	-0.0344	0.7313	-0.0567

I further investigated the effect of building models for individual non-Windows applications. For example, I extracted Internet Explorer (IE) instances and build several models for this set. When CVSS metrics are included, the correlation coefficient is approximately 0.7. This is better than when CVSS metrics are not included, in which case, the correlation coefficient is approximately 0.3. The results showing the comparison between IE models with and without CVSS metrics is shown in Table 5.6. I tried to performed a similar experiment for Office. However, there are only 300 instances for Office. Other office-related instances are about individual software such as Word, PowerPoint, Excel and Access, *etc*, and each has less than 300 instances. Given the small number of instances, I could not build

models for Office.

Table 5.6: *Correlation Coefficient for IE Vulnerability Regression Models, with and without CVSS Metrics*

Regression Functions	With CVSS		Without CVSS	
	training	test	training	test
Linear regression	0.8023	0.6717	0.7018	0.3892
Least mean square	0.6054	0.6968	0.4044	0.0473
Multi-layer perceptron	0.9929	0.6366	0.9518	0.0933
RBF network	0.1381	0.0118	0.151	-0.1116
SMO regression	0.7332	0.5876	0.5673	0.4813
Gaussian processes	0.9803	0.6048	0.9352	0.0851

Mozilla:

At last, I built classification models for Firefox, with and without the CVSS metrics. The results are shown in Table 5.7. As can be seen, the correctly classified rates are relatively good (approximately 0.7) in both cases. However, the number of instances in this dataset is rather small (less than 5000), therefore it is unclear how stable the prediction model is.

Table 5.7: *Correctly Classified Rate for Firefox Vulnerability Models with and without CVSS Metrics*

Classification Functions	With CVSS		Without CVSS	
	training	test	training	test
Simple logistic	97.5%	71.4%	97.5%	71.4%
Logistic regression	97.5%	70%	97.8%	70.5%
Multi-layer perceptron	99.5%	68.4%	99.4%	68.3%
RBF network	94.3%	71.9%	93.9%	67.1%
SMO	97.9%	55.3%	97.4%	55.3%

5.3.4 Parameter Tuning

As mentioned above, I used default parameters for all regression and classification models that I built. To investigate if different parameter settings could produce better results, I chose to tune parameters for the support vector machines algorithm (SVM), whose WEKA

implementations for classifications and regression are called SMO and SMO regression, respectively. There are two main parameters that can be tuned for SVM, denoted by C and σ. The C parameter is a cost parameter which controls the trade-off between model complexity and training error, while σ controls the width of the Gaussian kernel [2].

To find the best combination of values for C and σ, I generated a grid consisting of the following values for C: 0.5, 1.0, 2.0, 3.0, 5.0, 7.0, 10, 15, 20 and the following values for σ: 0, 0.05, 0.1, 0.2, 0.3, 0.5, 1.0, 2.0, 5.0, and run the SVM algorithm for all possible combinations. I used a separate validation set to select the combination of values that gives the best values for correlation coefficient, and root squared mean error and root relative squared error together. The validation and test datasets have approximately equal sizes; the test set consists of chronologically newer data, as compared to the validation data, while the validation data is newer than the training data.

Table 5.8 shows the best parameter values when tuning was performed based on the correlation coefficient, together with results corresponding to these parameter values, in terms of correlation coefficient, RRSE and RMSE (for both validation and test datasets). Table 5.9 shows similar results when parameters are tuned on RRSE and RMSE together.

Table 5.8: *Parameter Tuning Based on Correlation Coefficient*

Group Targeted	Parameters		Validation			Test		
	C	G	RMSE	RRSE	CC	RMSE	RRSE	CC
Adobe CVSS	3.0	2.0	75.2347	329.2137%	0.7399	82.2344	187.6%	0.4161
IE CVSS	1.0	1.0	8.4737	74.8534%	0.4516	11.6035	92.2%	-0.3396
Non-Windows	1.0	0.05	92.3105	101.0356%	0.1897	123.4387	100.7%	0.223
Linux CVSS	15.0	0.1	12.6302	130.8731%	0.1933	45.0535	378.3%	0.2992
Adobe	0.5	0.05	43.007	188.1909%	0.5274	78.2092	178.5%	0.1664
IE	7.0	0.05	13.8438	122.2905%	0.2824	14.5263	115.5%	-0.0898
Apple Separate	3.0	0.05	73.9528	104.0767%	0.2009	91.1742	116.4%	-0.4736
Apple Single	0.5	0.0	493.6879	694.7868%	0	521.228	1401.6%	0
Linux Separate	2.0	0.05	16.2225	188.6665%	0.3105	49.8645	418.7%	-0.111
Linux Single	1.0	0.05	11.3774	83.2248%	0.5465	9.4743	79.6%	0.3084
Linux Binned	2.0	0.05	16.2225	188.6665%	0	49.8645	418.7%	-0.111
Windows	5	0.05	21.0706	97.4323%	0.1974	72.1904	103.1%	0.1135

Table 5.9: *Parameter Tuning Based on RMSE and RRSE*

Group Targeted	Parameters		Validation			Test		
	C	G	RMSE	RRSE	CC	RMSE	RRSE	CC
Adobe CVSS	0.5	0.2	19.4018	84.8989%	0.2083	61.2009	139.6667%	0.5236
IE CVSS	2.0	1.0	8.4729	74.8645%	0.4466	11.4604	91.1018%	-0.3329
Non-Windows	0.5	0.1	91.1683	99.7855%	0.188	123.5291	100.7%	0.2117
Linux CVSS	2.0	0.5	7.83	81.1399%	0.1087	19.1453	160.8%	0.3002
Adobe	1.0	0.5	19.5024	85.3392%	-0.4387	106.2898	242.5%	0.547
IE	0.5	0.3	12.4578	110.0474%	0.2169	13.5771	107.9%	-0.1126
Apple Separate	7.0	1.0	70.7617	99.5857%	0.1325	80.2045	102.4%	-0.0406
Apple Single	0.5	0.05	75.9574	106.8979%	-0.3533	82.649	105.5%	-0.4429
Linux Separate	0.5	2.0	14.5428	106.3799%	0.2326	18.5708	155.9%	0.1236
Linux Single	5.0	0.5	10.7041	78.2999%	0.4752	12.3339	103.6%	0.3259
Linux Binned	0.5	2.0	14.5428	106.3799%	0.2326	18.5708	155.9%	0.1236
Windows	5.0	0.05	21.0706	97.4323%	0.1974	72.1904	103%	0.1135

5.3.5 Summary

The experiments above indicate that it is hard to build good prediction models based on the limited data available in NVD. For example, there is no version information for most Microsoft instances (especially, Windows instances). Some results look promising (*e.g.* the models I built for Firefox), but they are far from usable in practice. Below, I discuss what I believe to be the main reasons for the difficulty of building good prediction models for TTNV from NVD.

5.3.6 Discussion

I present my effort in building prediction models for zero-day vulnerabilities based on the information contained in the National Vulnerability Database. My research found that due to a number of limitations of this data source, it is unlikely that one can build a practically usable prediction model at this time. I presented my rigorous evaluation of various feature construction schemes and parameter tuning for learning algorithms, and notice that none of the results obtained shows acceptable performance.

I believe the main factor affecting the predictive power of the models is the low quality

of the data from the National Vulnerability Database. Following are several limitations of the data:

- Missing information: most instances of Microsoft do not have the version information, without which I could not observe how the number of vulnerabilities evolves over versions.

- "Zero" versiondiffs: most of versiondiff values are zero because earlier-version applications are also affected by the later-found vulnerabilities (this is assumed by a number of large companies, *e.g.* Microsoft and Adobe) and significantly reduces the utility of this feature.

- Vulnerability release time: The release date of vulnerability could largely be affected by vendors' practices. For example, Microsoft usually releases their vulnerability and patch information on the second Tuesday of each month, which may not accurately reflect the discovery date of the vulnerabilities.

- Data error: I found a number of obvious errors in NVD, such as the aforementioned Linux kernel version error.

Chapter 6

Conclusion

In this dissertation, I have presented a number of quantitative risk assessment approaches under different cyber environments. This work is significantly based upon attack graph and standard risk assessment methodology - Common Vulnerability Scoring System (CVSS).

In Chapter 2, I introduced a network abstraction model, through which the size of attack graph can be significantly reduced. This abstraction model could increase both visualization of attack graph and accuracy of risk assessment approaches. This approach can be widely applicable as the clustering management becomes common. It is straightforward to abstract a node which could represent a large set of hosts with similar configurations and reachabilities. Moreover, the abstraction level can be further customized by system administrators due to their available resource or preference. A balance of accuracy and scalability can be archived by adapting the model.

In Chapter 3, I introduced a risk assessment approach at a microscope - system level. Similar to the inter-host dependency mentioned in the previous chapter, the dependency among packages within the same system deeply affect the attack surface of the whole system. I captured such dependency by constructing a set of systematic risk assessment approaches. Each of these approach is able to output a risk metric at different granularities (vulnerability level, component level, package level or system level). These approaches bridge the gap

between the standard vulnerability assessment approach and system wide attack surface evaluation. Not only the approach can help image publisher to track the risk trend of their images, but also can help users to choose a reliable images among a considerable amount of candidates.

In Chapter 4, I presented our find about the attack cost effectiveness change over the IaaS cloud compared to traditional network environment. I did a penetration test on Amazon EC2 to show how easy to make big impact over a public cloud. I created a game theoretical model to model the game between attacker and defender. The model indicate defender needs to have stronger motivation of hardening their environment over the cloud. I also constructed a risk density model over time. The model can be used to model the cost and risk/gain for both defender and attacker. Not only can this model illustrate the cost effectiveness rate has significantly lowered for attacker under the cloud environment, but also can provide us three directions to minimize attacker's gain. Based on the three directions, four countermeasures are recommended to shrink attacker's potential gain. The model can be more accurate once certain type of data (e.g. image instantiation rate, OS distribution over running instances) from cloud provider's side become available.

In Chapter 5, I introduced a data-mining based approach to evaluate potential risk per application. I tried to use data in National Vulnerability Database to predict time to next vulnerability for each application. I leverage WEKA a data-mining tool to train the data. I did extensive experiments by using different combinations of features and I tune the parameters with different algorithms. Part of the result is promising but most of them do not have strong predictive power. The model is hard to be feasible as for now because of the low quality of the available data.

Each of the above mentioned approaches is an attempt to bridge the gap between industry standard vulnerability assessment approaches and risk assessment methodologies under different classic contexts. My network risk assessment approach and system level software dependency risk assessment approach is able to provide security metrics with higher accu-

racy under their contexts compared to standard risk assessment metrics. Due to the lack of reliable data, my cloud computing risk assessment approach and zero-day risk assessment approach is currently in between qualitative stage and quantitative stage. I have built the mathematical models, but more data is needed to train the models to make them output security metrics with sufficient accuracy. My objective is to build a seamless risk assessment approach under complex cyber environments. When more high quality data become available, I am positive that we can normalize these approaches together and provide a comprehensive risk metric for stake holders to understand the risk situation over a complicated cyber environment. For example, the microscope (system level) metric can be applied by risk assessment approaches under macroscope (enterprise network or cloud) environment. Unknown risk can serve as one factor and adjust the risk from known vulnerabilities in order to provide a more comprehensive risk metric. Asset value associated with each component under the cyber environment can be used to calculate and normalize the security metric per ecosystem (the entire corporate cyber environment).

Bibliography

[1] Root relative squared error. Website. http://www.gepsoft.com/gxpt4kb/Chapter10/Section1/SS07.htm.

[2] Support vector machines. Website. http://www.dtreg.com/svm.htm.

[3] (2008). *Customer Success. Powered by the AWS Cloud.* http://aws.amazon.com/solutions/case-studies/: Amazon Web Services LLC.

[4] (2009). *VMware ESX and VMware ESXi – The Market Leading Production-Proven Hypervisors.* http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf: VMware Inc.

[5] (2012). *Amazon Web Services Discussion Forums.* https://forums.aws.amazon.com/forum.jspa?forumID=30: Amazon Web Services LLC.

[6] Abate, P., R. Di Cosmo, J. Boender, and S. Zacchiroli (2009). Strong dependencies between software components. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 89–99. IEEE Computer Society.

[7] Alhazmi, O. H. and Y. K. Malaiya. (2006). Prediction capabilities of vulnerability discovery models. In *Annual Reliability and Maintainability Symposium (RAMS)*.

[8] Ammann, P., D. Wijesekera, and S. Kaushik. (2002). Scalable, graph-based network vulnerability analysis. In *9th ACM Conference on Computer and Communications Security(CCS)*.

[9] Anderson, R. and T. Moore (2007). Information security economics–and beyond. *Advances in Cryptology-CRYPTO 2007*, 68–91.

[10] Balduzzi, M., J. Zaddach, D. Balzarotti, E. Kirda, and S. Loureiro (2012). A security analysis of amazon's elastic compute cloud service. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pp. 1427–1434. ACM.

[11] Bilge, L. and T. Dumitras (2012). Before we knew it: an empirical study of zero-day attacks in the real world. In *ACM Conference on Computer and Communications Security*, pp. 833–844.

[12] Bouckaert, R. R., E. Frank, M. Hall, R. Kirkby, P. Reutemann, A. Seewald, and D. Scuse. (2010). *WEKA Manual for Version 3.7. The University of Waikato.* The University of Waikato.

[13] Bugiel, S., S. Nürnberger, T. Pöppelmann, A. Sadeghi, and T. Schneider (2011). Amazonia: when elasticity snaps back. In *Proceedings of the 18th ACM conference on Computer and communications security*, pp. 389–400. ACM.

[14] Buttner, A. and N. Ziring. (2009). Common platform enumeration (cpe) specification. Technical report, The MITRE Corporation AND National Security Agency.

[15] Cavusoglu, H., H. Cavusoglu, and J. Zhang (2006). Economics of security patch management. In *The fifth workshop on the economics of information security (WEIS 2006).*

[16] Cavusoglu, H., H. Cavusoglu, and J. Zhang (2008). Security patch management: Share the burden or share the damage? *Management Science 54*(4), 657–670.

[17] Cheng, P., L. Wang, S. Jajodia, and A. Singhal (2012). Aggregating cvss base scores for semantics-rich network security metrics. In *Proceedings of the 31st IEEE International Symposium on Reliable Distributed Systems (SRDS 2012).* IEEE Computer Society.

[18] Chowdhury, I. and M. Zulkernine (2010). Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 1963–1969. ACM.

[19] Consulting, F. (2009, January). *DDoS: A Threat You Can't Afford To Ignore*. https://www.verisign.com/static/VeriSign_DDoS.pdf.

[20] Dacey, R. (2003). *Information security: effective patch management is critical to mitigating software vulnerabilities*. General Accounting Office.

[21] Dacier, M., Y. Deswarte, and M. Kaâniche (1996). Models and tools for quantitative assessment of operational security. In *IFIP SEC*.

[22] Dawkins, J. and J. Hale (2004, April). A systematic approach to multi-stage network attack analysis. In *Proceedings of Second IEEE International Information Assurance Workshop*, pp. 48 – 56.

[23] DeLoach, S. A., X. Ou, R. Zhuang, and S. Zhang (2014). Model-driven, moving-target defense for enterprise network security. In *Models@ run. time*, pp. 137–161. Springer.

[24] Dewri, R., N. Poolsappasit, I. Ray, and D. Whitley (2007). Optimal security hardening using multi-objective optimization on attack tree models of networks. In *14th ACM Conference on Computer and Communications Security (CCS)*.

[25] Drake, J. (2011). Exploiting memory corruption vulnerabilities in the java runtime.

[26] Ellison, R., J. Goodenough, C. Weinstock, and C. Woody (2010). Evaluating and mitigating software supply chain security risks. Technical report, DTIC Document.

[27] Forbath, T., P. Kalaher, and T. O'Grady (2005). The total cost of security patch management. Technical report, Wipro Product Strategy & Architecture.

[28] Frei, S. (2009). *Security Econometrics - The Dynamics of (In)Security*. Eth zurich, dissertation 18197, ETH Zurich. ISBN 1-4392-5409-5, ISBN-13: 9781439254097.

[29] Frei, S., M. May, U. Fiedler, and B. Plattner (2006). Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pp. 131–138. ACM.

[30] Frigault, M., L. Wang, A. Singhal, and S. Jajodia (2008). Measuring network security using dynamic Bayesian network. In *Proceedings of the 4th ACM workshop on Quality of protection*.

[31] Ghosh, A., S. Noel, and S. Jajodia (2011). Mapping attack paths in black-box networks through passive vulnerability inference. Technical report, DTIC Document.

[32] Goichon, F., G. Salagnac, P. Parrend, and S. Frénot (2012). Static vulnerability detection in java service-oriented components. *Journal in Computer Virology*, 1–12.

[33] Gong, L. (2009). Java security: a ten year retrospective. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pp. 395–405. IEEE.

[34] Grobauer, B., T. Walloschek, and E. Stocker (2011). Understanding cloud computing vulnerabilities. *Security & Privacy, IEEE 9*(2), 50–57.

[35] Homer, J., X. Ou, and D. Schmidt (2009a). A sound and practical approach to quantifying security risk in enterprise networks. Technical report, Kansas State University.

[36] Homer, J., X. Ou, and D. Schmidt (2009b). A sound and practical approach to quantifying security risk in enterprise networks. Technical report, Kansas State University.

[37] Homer, J., A. Varikuti, X. Ou, and M. A. McQueen (2008). Improving attack graph visualization through data reduction and attack grouping. In *The 5th International Workshop on Visualization for Cyber Security (VizSEC)*.

[38] Homer, J., S. Zhang, X. Ou, D. Schmidt, Y. Du, S. R. Rajagopalan, and A. Singhal (2013). Aggregating vulnerability metrics in enterprise networks using attack graphs. *Journal of Computer Security 21*(4), 561–597.

[39] Howard, M., J. Pincus, and J. Wing (2005). Measuring relative attack surfaces. *Computer Security in the 21st Century*, 109–137.

[40] Huang, H., S. Zhang, X. Ou, A. Prakash, and K. Sakallah (2011). Distilling critical attack graph surface iteratively through minimum-cost sat solving. In *Proceedings of the 27th Annual Computer Security Applications Conference (ACSAC)*, pp. 31–40. ACM.

[41] Ingols, K., M. Chu, R. Lippmann, S. Webster, and S. Boyer (2009). Modeling modern network attacks and countermeasures using attack graphs. In *25th Annual Computer Security Applications Conference (ACSAC)*.

[42] Ingols, K., R. Lippmann, and K. Piwowarski (2006, December). Practical attack graph generation for network defense. In *22nd Annual Computer Security Applications Conference (ACSAC)*, Miami Beach, Florida.

[43] Jajodia, S. and S. Noel (2010, March). Advanced cyber attack modeling analysis and visualization. Technical Report AFRL-RI-RS-TR-2010-078, Air Force Research Laboratory.

[44] Jajodia, S., S. Noel, and B. O'Berry (2003). Topological analysis of network attack vulnerability. In V. Kumar, J. Srivastava, and A. Lazarevic (Eds.), *Managing Cyber Threats: Issues, Approaches and Challanges*, Chapter 5. Kluwer Academic Publisher.

[45] Jormakka, J. and J. Mölsä (2005). Modelling information warfare as a game. *Journal of information warfare 4*(2), 12–25.

[46] Khan, M. and S. Mahmood (2012). A graph based requirements clustering approach for component selection. *Advances in Engineering Software 54*, 1–16.

[47] Khirwadkar, T. (2011). *Defense against network attacks using game theory.* Ph. D. thesis, University of Illinois.

[48] Koehler, P., A. Anandasivam, M. Dan, and C. Weinhardt (2010). Cloud services from a consumer perspective. *AMCIS 2010 Proceedings*, 329.

[49] Li, W., R. B. Vaughn, and Y. S. Dandass (2006). An approach to model network exploitations using exploitation graphs. *SIMULATION 82*(8), 523–541.

[50] Lippmann, R., D. Fried, K. Piwowarski, and W. Streilein (2003). Passive operating system identification from tcp/ip packet headers. In *Workshop on Data Mining for Computer Security*, pp. 40. Citeseer.

[51] Lippmann, R. and K. W. Ingols (2005, March). An annotated review of past papers on attack graphs. Technical report, MIT Lincoln Laboratory.

[52] Lippmann, R. P., K. W. Ingols, C. Scott, K. Piwowarski, K. Kratkiewicz, M. Artz, and R. Cunningham (2005, October). Evaluating and strengthening enterprise network security using attack graphs. Technical Report ESC-TR-2005-064, MIT Lincoln Laboratory.

[53] Litty, L. and D. Lie (2011). Patch auditing in infrastructure as a service clouds. In *ACM SIGPLAN Notices*, Volume 46, pp. 145–156. ACM.

[54] Liu, H. (2010). A new form of dos attack in a cloud and its avoidance mechanism. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, pp. 65–76. ACM.

[55] Luo, W., L. Xu, Z. Zhan, Q. Zheng, and S. Xu (2014). Federated cloud security architecture for secure and agile clouds. In *High Performance Cloud Auditing and Applications*, pp. 169–188. Springer.

[56] Manadhata, P. and J. Wing (2004). Measuring a system's attack surface. Technical report, DTIC Document.

[57] Manadhata, P. and J. Wing (2011). An attack surface metric. *Software Engineering, IEEE Transactions on 37*(3), 371–386.

[58] Marouf, S. (2008). *An Extensive Analysis of the Software Security Vulnerabilities that exist within the Java Software Execution Environment*. Ph. D. thesis, University of Wisconsin.

[59] Massacci, F. and V. H. Nguyen. (2010). Which is the right source for vulnerability studies? an empirical analysis on mozilla firefox. In *MetriSec*.

[60] Maynor, D. (2007). *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress.

[61] McQueen, M., T. McQueen, W. Boyer, and M. Chaffin (2009). Empirical estimates and observations of 0day vulnerabilities. In *42nd Hawaii International Conference on System Sciences*.

[62] Mell, P., K. Scarfone, and S. Romanosky (2007a, June). *A Complete Guide to the Common Vulnerability Scoring System Version 2.0*. Forum of Incident Response and Security Teams (FIRST).

[63] Mell, P., K. Scarfone, and S. Romanosky (2007b). A complete guide to the common vulnerability scoring system version 2.0. In *Published by FIRST-Forum of Incident Response and Security Teams*, pp. 1–23.

[64] Mellberg, J. (2012). *Take patch management to the next level*. https://www.brighttalk.com/webcast/8113/54861: Secunia.

[65] Nasiri, S., R. Azmi, and R. Khalaj (2010). Adaptive and quantitative comparison of j2ee vs.. net based on attack surface metric. In *Telecommunications (IST), 2010 5th International Symposium on*, pp. 199–205. IEEE.

[66] Neuhaus, S. and T. Zimmermann (2009). The beauty and the beast: vulnerabilities in red hat's packages. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, Berkeley, CA, USA, pp. 30–30. USENIX Association.

[67] Neuhaus, S., T. Zimmermann, C. Holler, and A. Zeller (2007). Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 529–540. ACM.

[68] Nguyen, V. H. and L. M. S. Tran. (2010). Predicting vulnerable software components with dependency graphs. In *MetriSec*.

[69] Noel, S. and S. Jajodia (2004). Managing attack graph complexity through visual hierarchical aggregation. In *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, New York, NY, USA, pp. 109–118. ACM Press.

[70] Noel, S., S. Jajodia, L. Wang, and A. Singhal (2010, July). Measuring security risk of networks using attack graphs. *International Journal of Next-Generation Computing 1*(1).

[71] Oh, J. (2009). Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *Blackhat technical Security Conference*.

[72] Ortalo, R., Y. Deswarte, and M. Kaâniche (1999). Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering 25*(5).

[73] Ou, X., W. F. Boyer, and M. A. McQueen (2006). A scalable approach to attack graph generation. In *13th ACM Conference on Computer and Communications Security (CCS)*, pp. 336–345.

[74] Ou, X., S. Govindavajhala, and A. W. Appel (2005). MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium*.

[75] Ozment, A. (2007a). Improving vulnerability discovery models analyzer. In *QoP07*.

[76] Ozment, A. (2007b). *Vulnerability Discovery & Software Security*. Ph. D. thesis, University of Cambridge.

[77] Parrend, P. (2009). Enhancing automated detection of vulnerabilities in java components. In *Availability, Reliability and Security, 2009. ARES'09. International Conference on*, pp. 216–223. IEEE.

[78] Parrend, P. and S. Frénot (2008). Classification of component vulnerabilities in java service oriented programming (sop) platforms. *Component-Based Software Engineering*, 80–96.

[79] Pérez, P., J. Filipiak, and J. Sierra (2011). Lapse+ static analysis security software: Vulnerabilities detection in java ee applications. *Future Information Technology*, 148–156.

[80] Phillips, C. and L. P. Swiler (1998). A graph-based system for network-vulnerability analysis. In *NSPW '98: Proceedings of the 1998 workshop on New security paradigms*, pp. 71–79. ACM Press.

[81] Raemaekers, S., A. van Deursen, and J. Visser (2011). Exploring risks in the usage of third party libraries. In *of the BElgian-NEtherlands software eVOLution seminar*, pp. 31.

[82] Rajbhandari, L. and E. Snekkenes (2011a). An approach to measure effectiveness of control for risk analysis with game theory. In *Socio-Technical Aspects in Security and Trust (STAST), 2011 1st Workshop on*, pp. 24–29. IEEE.

[83] Rajbhandari, L. and E. Snekkenes (2011b). Mapping between classical risk management and game theoretical approaches. In *Communications and Multimedia Security*, pp. 147–154. Springer.

[84] Ransbotham, S., S. Mitra, and J. Ramsey (2012). Are markets for vulnerabilities effective? *MIS Quarterly-Management Information Systems 36*(1), 43.

[85] Richardson, R. (2011). 15th annual 2010/2011 computer crime and security survey. Technical report, Computer Security Institute.

[86] Roy, S., C. Ellis, S. Shiva, D. Dasgupta, V. Shandilya, and Q. Wu (2010). A survey of game theory as applied to network security. In *System Sciences (HICSS), 2010 43rd Hawaii International Conference on*, pp. 1–10. IEEE.

[87] Saha, D. (2008). Extending logical attack graphs for efficient vulnerability analysis. In *Proceedings of the 15th ACM conference on Computer and Communications Security (CCS)*.

[88] Sawilla, R. and X. Ou (2008, October). Identifying critical attack assets in dependency attack graphs. In *13th European Symposium on Research in Computer Security (ESORICS)*, Malaga, Spain.

[89] Sheyner, O., J. Haines, S. Jha, R. Lippmann, and J. M. Wing (2002). Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 254–265.

[90] Shin, S. and G. Gu (2010). Conficker and beyond: a large-scale empirical study. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 151–160. ACM.

[91] Shrobe, H. (2012). What if we got a do-over? In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pp. 55–56. ACM.

[92] Sommestad*, T., M. Ekstedt, and P. Johnson (2010). A probabilistic relational model for security risk analysis. *Computer & Security 29*, 659–679.

[93] Studer, R. (2011). Economic and technical analysis of botnets and denial-of-service attacks. *Communication systems IV*, 19.

[94] Swiler, L. P., C. Phillips, D. Ellis, and S. Chakerian (2001, June). Computer-attack graph generation tool. In *DARPA Information Survivability Conference and Exposition (DISCEX II'01)*, Volume 2.

[95] Talbot, D. (2009). Vulnerability seen in amazon's cloud-computing. Technical report, MIT Tech Review.

[96] Tidwell, T., R. Larson, K. Fitch, and J. Hale (2001, June). Modeling Internet attacks. In *Proceedings of the 2001 IEEE Workshop on Information Assurance and Security*, West Point, NY.

[97] Unruh, I., A. G. Bardas, R. Zhuang, X. Ou, and S. A. DeLoach (2013). Compiling abstract specifications into concrete systems–bringing order to the cloud. Technical report, Kansas State University.

[98] Vijayakumar, H., G. Jakka, S. Rueda, J. Schiffman, and T. Jaeger (2012, May). Integrity walls: Finding attack surfaces from mandatory access control policies. In *Proceedings of the 7th ACM Symposium on Information, Computer, and Communications Security (ASIACCS 2012)*.

[99] Wang, L., T. Islam, T. Long, A. Singhal, and S. Jajodia (2008). An attack graph-based probabilistic security metric. In *Proceedings of The 22nd Annual IFIP WG 11.3 Working Conference on Data and Applications Security (DBSEC'08)*.

[100] Wang, L., A. Singhal, and S. Jajodia (2007a). Measuring network security using attack graphs. In *Third Workshop on Quality of Protection (QoP)*.

[101] Wang, L., A. Singhal, and S. Jajodia (2007b). Measuring the overall security of network configurations using attack graphs. In *Proceedings of 21th IFIP WG 11.3 Working Conference on Data and Applications Security (DBSEC'07)*.

[102] Wen, Y., J. Lee, Z. Liu, Q. Zheng, W. Shi, S. Xu, and T. Suh (2013). Multi-processor architectural support for protecting virtual machine privacy in untrusted cloud environment. In *Proceedings of the ACM International Conference on Computing Frontiers*, pp. 25. ACM.

[103] Williams, L., R. Lippmann, and K. Ingols (2007). An interactive attack graph cascade and reachability display. In *IEEE Workshop on Visualization for Computer Security (VizSEC 2007)*.

[104] Xiong, H., Q. Zheng, X. Zhang, and D. Yao (2013). Cloudsafe: Securing data processing within vulnerable virtualization environments in the cloud. In *Communications and Network Security (CNS), 2013 IEEE Conference on*, pp. 172–180. IEEE.

[105] Xu, Y., M. Bailey, E. V. Weele, and F. Jahanian (2010, November). Canvus: Context-aware network vulnerability scanning. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID))*.

[106] Yamaguchi, F., F. Lindner, and K. Rieck (2011). Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pp. 13–13. USENIX Association.

[107] Yan, G., R. Lee, A. Kent, and D. H. Wolpert (2012). Towards a bayesian network game framework for evaluating ddos attacks and defense. In *ACM Conference on Computer and Communications Security*, pp. 553–566.

[108] Zhang, S. Deep-diving into an easily-overlooked threat: Inter-vm attacks.

[109] Zhang, S., D. Caragea, and X. Ou (2011). An empirical study on using the national vulnerability database to predict software vulnerabilities. In *Database and Expert Systems Applications (DEXA)*, pp. 217–231. Springer.

[110] Zhang, S., X. Ou, and J. Homer (2011). Effective network vulnerability assessment through model abstraction. In *Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment (DIMVA)*, pp. 17–34. Springer-Verlag.

[111] Zhang, S., X. Ou, A. Singhal, and J. Homer (2011). An empirical study of a vulnerability metric aggregation method.

[112] Zhang, S., X. Zhang, and X. Ou (2014). After we knew it: empirical study and modeling of cost-effectiveness of exploiting prevalent known vulnerabilities across iaas cloud. In *Proceedings of the 9th ACM symposium on Information, computer and communications security (ASIACCS)*, pp. 317–328. ACM.

[113] Zheng, Q., S. Xu, and G. Ateniese (2012). Efficient query integrity for outsourced dynamic databases. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop*, pp. 71–82. ACM.

[114] Zhou, W., P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala (2010). Always up-to-date: scalable offline patching of vm images in a compute cloud. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pp. 377–386. ACM.

[115] Zhuang, R., S. Zhang, A. Bardas, S. A. DeLoach, X. Ou, and A. Singhal (2013). Investigating the application of moving target defenses to network security. In *Resilient Control Systems (ISRCS), 2013 6th International Symposium on*, pp. 162–169. IEEE.

[116] Zhuang, R., S. Zhang, S. A. DeLoach, X. Ou, and A. Singhal (2012). Simulation-based approaches to studying effectiveness of moving-target network defense. In *Proceedings of the National Symposium on Moving Target Research*.

[117] Zimmermann, T. and N. Nagappan (2008). Predicting defects using network analysis on dependency graphs. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pp. 531–540. IEEE.