Assessing Attack Surface with Component-Based Package Dependency

Su Zhang[1(✉)], Xinwen Zhang[2], Xinming Ou[3], Liqun Chen[4], Nigel Edwards[4], and Jing Jin[5]

[1] Symantec Corporation, California, USA
westlifezs@gmail.com
[2] Samsung Research America, Mountain View, USA
xinwenzhang@gmail.com
[3] University of South Florida, Tampa, USA
xou@usf.edu
[4] Hewlett-Packard Laboratories, Bristol, UK
{liqun.chen,nigel.edwards}@hp.com
[5] Intuit Inc., California, USA
wendy_jin@intuit.com

Abstract. Package dependency has been considered in many vulnerability assessment systems. However, existing approaches are either coarse-grained and do not accurately reveal the influence and severity of vulnerabilities, or do not provide comprehensive (both incoming and outgoing) analysis of attack surface through package dependency. We propose a systematic approach of measuring attack surface exposed by individual vulnerabilities through component level dependency analysis. The metric could potentially extended to calculate attack surfaces at component, package, and system levels. It could also be used to calculate both incoming and outgoing attack surfaces, which enables system administrators to accurately evaluate how much risk that a vulnerability, a component or a package to the complete system, and the risk that is injected to a component or package by packages it depends on in a given system. To our best knowledge, our approach is the first to quantitatively assess attack surfaces of vulnerabilities, components, packages, and systems through component level dependency.

1 Introduction

Attack surface usually refers to exploitable resource exposed to attackers [18,19]. 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) [21] – quantifies the risk

M. Qiu et al. (Eds.): NSS 2015, LNCS 9408, pp. 405–417, 2015.
DOI: 10.1007/978-3-319-25645-0_29

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.

Moreover, CVSS does not take into consideration of package dependency, which, based on our analysis in this paper, 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.

Researchers have proposed to measure risk with the consideration of package dependency. Neuhaus et al. [23] study package dependency on Red Hat systems, and infer beauty packages (with low risk) and beast packages (high risk) based on the inter-package dependencies and historical vulnerability information for each package. Their output can be used by developers to choose dependable packages with low risks or historical vulnerabilities. *But they only consider the number of historical vulnerabilities as the risk factor for each package, rather than measuring the attack surface brought by known vulnerabilities on a given system.* Raemaekers et al. [31] study the risk of a package brought by third party libraries. They evaluate potential risks from third party applications by considering if the referenced packages are well scrutinized, the number of referenced packages, and the number of classes with referenced libraries. *However, they only measure incoming risk (risk brought by third party libraries) at package level, and do not consider any finer-grained (component level) or coarser-grained (system level) incoming attack surface.* Moreover, this work *does not evaluate outgoing attack surfaces*, which are brought by individual vulnerabilities, components, and packages to a system, and are important inputs when prioritizing security related plans such as patching and hardening by system administrators, and choosing dependent packages for developers.

With our 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. Our solution also helps other stakeholders to observe the evolution of package dependency based attack surface for a given system.

2 Overview

2.1 A Real Motivating Example

To motivate our attack surface analysis with package dependency, we 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 our analysis of VMSA entries from July 2007 to December 2012, we find out that almost two thirds (56/90) of the VMSAs include vulnerabilities originated from third party applications that affect VMware products, as Table 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 [1]. 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 1 for details).

Table 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

Our analysis with VMSA motivates a security metric with the consideration of package dependency, which can help system administrator and software developer to identify vulnerabilities on highly depended programs (e.g., JRE and Linux_console) with larger attack surfaces, compared to others such as client side vulnerabilities (see Figure 1). Consequently, the system administrator may want

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

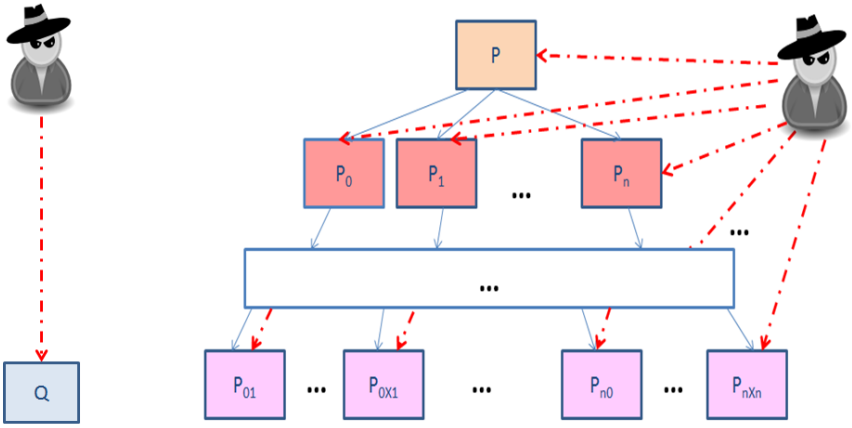

Fig. 1. Comparison of attack paths to a vulnerable client side application `Q` and a highly depended library `P`.

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.

2.2 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 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*, we 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 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.

Less complex dependency information than class level: We keep our 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

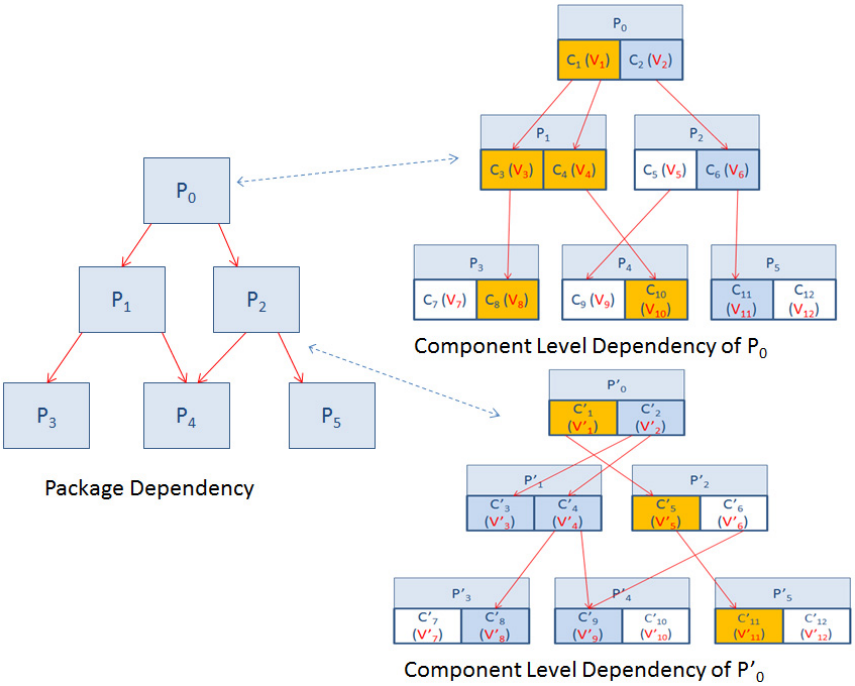

Fig. 2. One package level dependency with two different component level dependencies.

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 [25,26]. 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 [24]. Moreover, the complexity of a class level dependency map is exponentially higher compared to a component level dependency graph. We 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 Dependency-Based Attack Surface Analysis

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

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

3.1 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 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 our analysis, we 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, we assign an equal weight 1 to each dependency between two components in our analysis. But we still keep a weight variable in our algorithms just for future customization of the dependency weight based on different preferences.

3.2 Component-Based Attack Surface Analysis

Vulnerability Attack Surface. We 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 1 shows, for each vulnerability, we 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, we 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. We 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, we 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

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

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

this circumstance, we ignore the dependency $c_a \Longrightarrow c_b \Longrightarrow c_c$ but only consider $c_a \Longrightarrow c_c$.

We 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. We assign "1" to all impact values in our experiments because we treat all dependencies equally as mentioned in Section 3.1), 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.

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

4 Future Work

We propose an attack surface at vulnerability level. The metric could also be aggregated into higher levels. Component level attack surface will let state holders to know how much risk is brought by each individual component and plan hardening accordingly. Package level attack surface can be used to determine which package to depend upon among similar packages. System level attack surface can be used to indicate the health level of individual systems/images. This will help potential users to decide which image to use. Experiments can also be conducted under different environments [5,16,28–30,41,42,46,53,54,56,57] along with other approaches [14,15,32,34–40,44,45,48,51,52]. Moreover, presentation tools like attack graph [10,12,17,47,49,50,55] can be used to visualize risks from software dependencies.

5 Related Work

Risks from package dependency have been well researched [2,4,7,13,23,25,31,43,58]. Neuhaus et al. [23] 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. [31] explore the risk from third party applications. Instead of measuring

[6] We assign 0.1 as the damping factor for our experiments

[7] We assign "1" to all DIV as mentioned in Section 3.1

[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 1. Dependency-based Attack Surface Measurement for Individual Vulnerabilities: **VAS**(v_0, d)

Input: 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)).
Output: 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 we skip it}
    end if
    Q.enqueue($c_k, n+1$) {Update Q in order to process dependents of c_k if within our 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}

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 [2,13,43,58] study the importance of component level dependency when assessing software quality but no concrete security metric has been proposed. Chowdhury et al. [4] 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 [6,8,9,20,22,26,27]. Nasiri et al. [22] 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. [6] 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. [9] retrospect the evolution of security mechanism on Java in the past ten years at high level. Both Pérez et al. [27] and Goichon et al. [8] propose vulnerability detection approaches after scanning Java source code. Marouf [20] classifies vulnerabilities specific to Java and proposes possible countermeasures against these threats. Similarly, Parrend et al.[26] classify Java vulnerability at component level rather than source code level.

Work regarding attack surface evaluation have been conducted by researchers [3,11,18,19,24,24,33]. Neuhaus et al. [24] rank vulnerable components in Firefox based on historical detected vulnerabilities. Similar to us, they evaluate risk at component level. However, they consider these components as independent units rather than inter-depended nodes.

The definition of attack surface is also adapted in industry. Similar to [18], which evaluates attack surface over Linux systems, Microsoft attack surface[9] 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.

6 Conclusions

We define attack surface exposed through package dependency at vulnerability level. Besides outgoing attack surfaces, we propose algorithms calculating incoming attack surfaces injected through package dependency into individual components and packages. Our 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.

References

1. VMware ESX and VMware ESXi - The Market Leading Production-Proven Hypervisors. VMware Inc. (2009). http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf
2. Abate, P., Di Cosmo, R., Boender, J., Zacchiroli, S.: 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 (2009)
3. Cheng, P., Wang, L., Jajodia, S., Singhal, A.: 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 (2012)

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

4. Chowdhury, I., Zulkernine, M.: 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 (2010)
5. DeLoach, S.A., Ou, X., Zhuang, R., Zhang, S.: Model-driven, moving-target defense for enterprise network security. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time. LNCS, vol. 8378, pp. 137–161. Springer, Heidelberg (2014)
6. Drake, J.J.: Exploiting memory corruption vulnerabilities in the java runtime (2011)
7. Ellison, R.J., Goodenough, J.B., Weinstock, C.B., Woody, C.: Evaluating and mitigating software supply chain security risks. Technical report, DTIC Document (2010)
8. Goichon, F., Salagnac, G., Parrend, P., Frénot, S.: Static vulnerability detection in java service-oriented components. Journal in Computer Virology, 1–12 (2012)
9. Gong, L.: Java security: a ten year retrospective. In: Annual Computer Security Applications Conference, ACSAC 2009, pp. 395–405. IEEE (2009)
10. Homer, J., Zhang, S., Ou, X., Schmidt, D., Du, Y., Rajagopalan, S.R., Singhal, A.: Aggregating vulnerability metrics in enterprise networks using attack graphs. Journal of Computer Security **21**(4), 561–597 (2013)
11. Howard, M., Pincus, J., Wing, J.: Measuring relative attack surfaces. In: Computer Security in the 21st Century, pp. 109–137 (2005)
12. Huang, H., Zhang, S., Ou, X., Prakash, A., Sakallah, K.: Distilling critical attack graph surface iteratively through minimum-cost sat solving. In: Proceedings of the 27th Annual Computer Security Applications Conference, pp. 31–40. ACM (2011)
13. Khan, M.A., Mahmood, S.: A graph based requirements clustering approach for component selection. Advances in Engineering Software **54**, 1–16 (2012)
14. Li, T., Zhou, X., Brandstatter, K., Raicu, I.: Distributed key-value store on hpc and cloud systems. In: 2nd Greater Chicago Area System Research Workshop (GCASR). Citeseer (2013)
15. Li, T., Zhou, X., Brandstatter, K., Zhao, D., Wang, K., Rajendran, A., Zhang, Z., Raicu, I.: Zht: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table. In: 2013 IEEE 27th International Symposium on Parallel & Distributed Processing (IPDPS), pp. 775–787. IEEE (2013)
16. Liu, X., Edwards, S., Riga, N., Medhi, D.: Design of a software-defined resilient virtualized networking environment. In: 11th International Conference on the Design of Reliable Communication Networks (DRCN), pp. 111–114. IEEE (2015)
17. Lv, Z., Su, T.: 3D seabed modeling and visualization on ubiquitous context. In: SIGGRAPH Asia 2014 Posters, SA 2014, pp. 33:1–33:1. ACM, New York (2014)
18. Manadhata, P., Wing, J.M.: Measuring a system's attack surface. Technical report, DTIC Document (2004)
19. Manadhata, P.K., Wing, J.M.: An attack surface metric. IEEE Transactions on Software Engineering **37**(3), 371–386 (2011)
20. Marouf, S.M.: An Extensive Analysis of the Software Security Vulnerabilities that exist within the Java Software Execution Environment. PhD thesis, University of Wisconsin (2008)
21. Mell, P., Scarfone, K., Romanosky, S.: 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 (2007)
22. Nasiri, S., Azmi, R., Khalaj, R.: Adaptive and quantitative comparison of J2EE vs. net based on attack surface metric. In: 2010 5th International Symposium on Telecommunications (IST), pp. 199–205. IEEE (2010)

23. Neuhaus, S., Zimmermann, T.: The beauty and the beast: vulnerabilities in red hat's packages. In: Proceedings of the 2009 Conference on USENIX Annual Technical Conference, USENIX 2009, p. 30. USENIX Association, Berkeley (2009)
24. Neuhaus, S., Zimmermann, T., Holler, C., Zeller, A.: Predicting vulnerable software components. In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 529–540. ACM (2007)
25. Parrend, P.: Enhancing automated detection of vulnerabilities in java components. In: International Conference on Availability, Reliability and Security, ARES 2009, pp. 216–223. IEEE (2009)
26. Parrend, P., Frénot, S.: Classification of component vulnerabilities in java service oriented programming (SOP) platforms. In: Chaudron, M.R.V., Ren, X.-M., Reussner, R. (eds.) CBSE 2008. LNCS, vol. 5282, pp. 80–96. Springer, Heidelberg (2008)
27. Pérez, P.M., Filipiak, J., Sierra, J.M.: LAPSE+ static analysis security software: Vulnerabilities detection in java EE applications. In: Park, J.J., Yang, L.T., Lee, C. (eds.) FutureTech 2011, Part I. CCIS, vol. 184, pp. 148–156. Springer, Heidelberg (2011)
28. Qian, H., Andresen, D.: Jade: An efficient energy-aware computation offloading system with heterogeneous network interface bonding for ad-hoc networked mobile devices. In: 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) (2014)
29. Qian, H., Andresen, D.: Emerald: Enhance scientific workflow performance with computation offloading to the cloud. In: 2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS), pp. 443–448. IEEE (2015)
30. Qian, H., Andresen, D.: An energy-saving task scheduler for mobile devices. In: 2015 IEEE/ACIS 14th International Conference on Computer and Information Science (ICIS), pp. 423–430. IEEE (2015)
31. Raemaekers, S., van Deursen, A., Visser, J.: Exploring risks in the usage of third party libraries. In: The Goal of the BElgian-NEtherlands Software eVOLution Seminar, p. 31 (2011)
32. Su, Y., Wang, Y., Agrawal, G., Kettimuthu, R.: Sdquery dsi: integrating data management support with a wide area data transfer protocol. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 47. ACM (2013)
33. Vijayakumar, H., Jakka, G., Rueda, S., Schiffman, J., Jaeger, T.: 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), May 2012
34. Wang, J.J.-Y., Sun, Y., Gao, X.: Sparse structure regularized ranking. Multimedia Tools and Applications, 1–20 (2014)
35. Wang, K., Liu, N., Sadooghi, I., Yang, X., Zhou, X., Lang, M., Sun, X.-H., Raicu, I.: Overcoming hadoop scaling limitations through distributed task execution
36. Wang, K., Zhou, X., Chen, H., Lang, M., Raicu, I.: Next generation job management systems for extreme-scale ensemble computing. In: Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, pp. 111–114. ACM (2014)
37. Wang, K., Zhou, X., Qiao, K., Lang, M., McClelland, B., Raicu, I.: Towards scalable distributed workload manager with monitoring-based weakly consistent resource stealing. In: Proceedings of the 24rd International Symposium on High-Performance Parallel and Distributed Computing, pp. 219–222. ACM (2015)

38. Wang, K., Zhou, X., Li, T., Zhao, D., Lang, M., Raicu, I.: Optimizing load balancing and data-locality with data-aware scheduling. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 119–128. IEEE (2014)
39. Wang, Y., Nandi, A., Agrawal, G.: Saga: array storage as a DB with support for structural aggregations. In: Proceedings of the 26th International Conference on Scientific and Statistical Database Management, p. 9. ACM (2014)
40. Wang, Y., Su, Y., Agrawal, G.: Supporting a light-weight data management layer over hdf5. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 335–342. IEEE (2013)
41. Wei, F., Roy, S., Ou, X., Robby.: Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1329–1341. ACM (2014)
42. Xiong, H., Zheng, Q., Zhang, X., Yao, D.: Cloudsafe: Securing data processing within vulnerable virtualization environments in the cloud. In: 2013 IEEE Conference on Communications and Network Security (CNS), pp. 172–180. IEEE (2013)
43. Yamaguchi, F., Lindner, F., Rieck, K.: Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In: Proceedings of the 5th USENIX conference on Offensive Technologies, p. 13. USENIX Association (2011)
44. Zhang, H., Diao, Y., Immerman, N.: Recognizing patterns in streams with imprecise timestamps. Proceedings of the VLDB Endowment **3**(1–2), 244–255 (2010)
45. Zhang, H., Diao, Y., Immerman, N.: On complexity and optimization of expensive queries in complex event processing. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 217–228. ACM (2014)
46. Zhang, S.: Deep-diving into an easily-overlooked threat: Inter-vm attacks. Whitepaper, provided by Kansas State University, TechRepublic/US2012 (2013). http://www.techrepublic.com/resourcelibrary/whitepapers/deep-diving-into-an-easilyoverlooked-threat-inter-vm-attacks
47. Zhang, S.: Quantitative risk assessment under multi-context environments. PhD thesis, Kansas State University (2014)
48. Zhang, S., Caragea, D., Ou, X.: An empirical study on using the national vulnerability database to predict software vulnerabilities. In: Hameurlain, A., Liddle, S.W., Schewe, K.-D., Zhou, X. (eds.) DEXA 2011, Part I. LNCS, vol. 6860, pp. 217–231. Springer, Heidelberg (2011)
49. Zhang, S., Ou, X., Homer, J.: Effective network vulnerability assessment through model abstraction. In: Holz, T., Bos, H. (eds.) DIMVA 2011. LNCS, vol. 6739, pp. 17–34. Springer, Heidelberg (2011)
50. Zhang, S., Ou, X., Singhal, A., Homer, J.: An empirical study of a vulnerability metric aggregation method. In: The 2011 International Conference on Security and Management (SAM 2011), Special Track on Mission Assurance and Critical Infrastructure Protection (STMACIP 2011) (2011)
51. Zhang, S., Zhang, X., Ou, X.: 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, pp. 317–328. ACM (2014)
52. Zhao, D., Zhang, Z., Zhou, X., Li, T., Wang, K., Kimpe, D., Carns, P., Ross, R., Raicu, I.: Fusionfs: Toward supporting data-intensive scientific applications on extreme-scale high-performance computing systems. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 61–70. IEEE (2014)

53. Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., Zou, W.: Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In: Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM 2012, pp. 93–104. ACM, New York (2012)
54. Zheng, Q., Zhu, W., Zhu, J., Zhang, X.: Improved anonymous proxy re-encryption with cca security. In: Proceedings of the 9th ACM Symposium on Information Computer and Communications Security, ASIA CCS 2014, pp. 249–258. ACM, New York (2014)
55. Zhou, X., Sun, X., Sun, G., Yang, Y.: A combined static and dynamic software birthmark based on component dependence graph. In: International Conference on Intelligent Information Hiding and Multimedia Signal Processing, pp. 1416–1421. IEEE (2008)
56. Zhuang, R., Zhang, S., Bardas, A., DeLoach, S.A., Ou, X., Singhal, A.: Investigating the application of moving target defenses to network security. In: 2013 6th International Symposium on Resilient Control Systems (ISRCS), pp. 162–169. IEEE (2013)
57. Zhuang, R., Zhang, S., DeLoach, S.A., Ou, X., Singhal, A.: Simulation-based approaches to studying effectiveness of moving-target network defense. In: National Symposium on Moving Target Research (2012)
58. Zimmermann, T., Nagappan, N.: Predicting defects using network analysis on dependency graphs. In: ACM/IEEE 30th International Conference on Software Engineering, ICSE 2008, pp. 531–540. IEEE (2008)