# Network Security Metrics: Estimating the Resilience of Networks Against Zero Day Attacks

Mengyuan Zhang

A thesis
in
The Concordia Institute
for
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Information and Systems Engineering) at
Concordia University
Montréal, Québec, Canada

December 2017

© Mengyuan Zhang, 2017

# CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Ms. Mengyuan Zhang**

Entitled: **Network Security Metrics: Estimating the Resilience of Networks Against Zero Day Attacks**

and submitted in partial fulfillment of the requirements for the degree of

**Doctor of Philosophy (Information and Systems Engineering)**

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair
Dr. E. Shihab

_____ External Examiner
Dr. M. Zulkernine

_____ External to Program
Dr. D. Qiu

_____ Examiner
Dr. C. Assi

_____ Examiner
Dr. M. Mannan

_____ Thesis Supervisor
Dr. L. Wang

Approved by _____
Dr. Chadi Assi, Graduate Program Director

February 21, 2018 _____
Dr. Amir Asif, Dean
Faculty of Engineering and Computer Science

# Abstract

Network Security Metrics: Estimating the Resilience of Networks Against Zero Day Attacks

**Mengyuan Zhang, Ph.D.**

**Concordia University, 2017**

Computer networks are playing the role of nervous systems in many critical infrastructures, governmental and military organizations, and enterprises today. Protecting such mission critical networks means more than just patching known vulnerabilities and deploying firewalls or IDSs. Proper metrics are needed in evaluating the security level of networks and provide security enhanced solutions. However, without considering unknown zero-day vulnerabilities, security metrics are insufficient to capture the true security level of a network. My doctoral work is aiming to develop a series of novel network security metrics with a special focus on modeling zero day attacks and study the relationships between software features and vulnerabilities.

In the first work, we take the first step toward formally modeling network diversity as a security metric by designing and evaluating a series of diversity metrics. In particular, we first devise a biodiversity-inspired metric based on the effective number of distinct resources. We then propose two complementary diversity metrics, based on the least and the average attacking efforts, respectively.

In the second topic, we lift the attack surface concept, which calculates the intrinsic properties of software applications, to the network level as a security metric for evaluating the resilience of networks against potential zero day attacks. First, we develop models for

aggregating the attack surface among different resources inside a network. Second, we design heuristic algorithms to avoid the costly calculation of attack surface.

Predicting and studying the software vulnerability both help administrators to improve security deployment for their organizations and to choose the right applications among those with similar functionality, and for the software vendors to estimate the security level of their software applications. In the third topic, we perform a large-scale empirical study on datasets from GitHub and different versions of Chrome to study the relationship between software features and the number of vulnerabilities. This study quantitatively demonstrates the importance of features in the vulnerability discovery process based on machine learning techniques, which provides inputs for network level security metrics. Those features could serve as inputs for future network security metrics.

# Acknowledgments

I would like to express my deepest thanks to my supervisor, Dr. Lingyu Wang, for his heartily guidance, endless support and enduring patience during the whole process of this research. Without him, this thesis would not have been possible. I have been extremely lucky to have a supervisor who lightened my research all the time, and who responded to my questions and queries so promptly.

I also wish to express my appreciation to all the faculty and staff at CIISE department for having such a cozy and warm working environment.

My deepest thanks to my research collaborators, especially Xavier de Carné de Carnavalet, who has supported me for my last research topic. I could not have completed this thesis without his valuable comments and straightforward advice.

Finally, I would like to extend thanks and appreciations to my friends, parents and sister for listening to me, supporting me, and encouraging me for all the time.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Motivation

Today's computer networks are playing the role of nervous systems in many critical infrastructures, governmental and military organizations, and enterprises. Protecting such a mission critical network means more than just patching known vulnerabilities and deploying firewalls and IDSes. The network's resilience against potential zero day attacks exploiting unknown vulnerabilities is equally important. The high profile incident of the Stuxnet botnet [60], which employ four zero day vulnerabilities to target industrial control systems, has clearly demonstrated the real world significance of evaluating and improving the security of networks against zero day attacks.

Since *you cannot improve what you cannot measure*, the lack of effective security metrics is the key obstacle toward developing systematic approaches to evaluating and consequently improving the relative effectiveness of security solutions. In well-established domains, such as physical science, the first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it. The concept of metric is coined in many different forms under different contexts; here we quote the definition given by SANS [90]: *"Metrics can be*

*an effective tool for security managers to discern the effectiveness of various components of their security programs; the security of a specific system, product or process. Metrics can also help identify the level of risk in not taking a given action, and in that way provide guidance in prioritizing corrective actions."*

In general, security metrics are divided into four categories [1], *process security metrics*, metrics are designed for measuring processes and procedures, *network security metrics*, metrics that evaluate security level on entire networks, *software security metrics*, metrics that measure the possibility to have flaws in software applications, and *people security metrics*, metrics that involve human experts:

- *Process Security Metrics*: This type of metrics are designed for measuring processes and procedures. It implies high utility of security policies and processes, but the relationship between metrics and the level of security is not clearly defined, for instance, the percentage of a system with tested security controls, or the number of identified risks and their severities.

- *Network Security Metrics*: Existing metrics in this category are usually dependent on specific products (e.g., firewalls, IDS, model checker), and are widely used for network systems. Those metrics are usually illustrated with charts and through interfaces, e.g., for the number of viruses blocked, the number of patches applied, and traffic analysis.

- *Software Security Metrics*: This type of metrics usually come with many limitations (e.g., line number of code (LOC), function points (FPs), high complexity), and are context-sensitive, environment- and architecture-dependent, e.g., size and complexity, defects (severity, type) over time.

- *People Security Metrics*: This category includes security metrics that involve human experts.

There exist many research efforts on network security metrics (Chapter 2 will review related work in more details). Notably, Leversage et al. propose a metric based on the MTTC (Mean Time-to-Compromise) concept as a measurement of the average effort required for attackers to compromise a network [63], and Wang et al. aggregate metric scores of individual vulnerabilities derived from the Common Vulnerability Scoring System (CVSS) [73] as a metric for the overall security of networks [129].

However, one key limitation of most existing efforts is the lack of consideration for unknown zero day vulnerabilities. Existing efforts on network security metrics typically assign numerical scores to vulnerabilities based on known facts about vulnerabilities. Such a methodology is no longer applicable when we consider zero day attacks. In fact, a popular criticism of past efforts on security metrics is that they cannot deal with unknown vulnerabilities, which are generally believed to be unmeasurable [70]. Unfortunately, without considering unknown vulnerabilities, a security metric will only be of a questionable value at best, as it may determine a network configuration to be more secure than it actually is. We thus fall into the agnosticism that security is not quantifiable until we can fix all potential security flaws; although by then, we certainly no longer need security metrics.

## 1.2 Objective and Contributions

This research aims to provide a series of useful metrics and formal models for evaluating the risk of zero-day vulnerabilities in computer networks and study the relationships between software features and vulnerabilities. Throughout this work, we make the following contributions.

- The main contribution for the first research topic is as follows. To the best of our knowledge, this is the first effort on systematically modeling network diversity as a security metric. As we demonstrate an intuitive notion of diversity can usually cause

misleading results, whereas our formal model of network diversity enables a better understanding of the effect of diversity on security. Our work is also among the first efforts on borrowing the biodiversity concepts from ecology and applying it to network security, and we believe this initial effort may generate further interest in this direction. Finally, the guidelines for instantiating the models and related case studies and simulation results will ease the transition from theoretical results to practical solutions based on the proposed metrics.

- The main contribution of the second research topic is twofold. First, to the best of our knowledge, this is the first effort on lifting the attack surface concept to the network level as a formally defined security metric. We believe such a metric may serve as the foundation of many useful analyses for quantitatively designing, evaluating, and improving network security. Second, our simulation results show that the proposed algorithms can produce relatively accurate results with a significant reduction in the costly calculation of attack surface, paving the way for practical applications.

- The main contribution for the third research topic is as follows. To the best of our knowledge, this is one of the first studies that applies machine learning techniques to study the relationship between features and the number of CVE vulnerabilities in software applications. This study points out the importance of the features that affect the discovery of the vulnerabilities, which serves as important inputs to other network security metrics.

# Chapter 2

# Literature Review

In this chapter, we provide a review of related literature.

## 2.1 Attack Graph

Today's security problems in a network system are not only caused by single vulnerabilities, but the combination of multi-vulnerabilities between multi-hosts [104]. Before defining security metric on a network level, a pre-condition is to understand how vulnerabilities from hosts may be combined as attack paths. Attack graph is a well established model for this purpose. Even well administered networks may still face the challenge of defending against network attacks. Many services are safer when they work alone, however, multiple services may help each other to execute exploitable vulnerabilities. Examining or scanning the host individually is insufficient in evaluating the security of networks; relationships between hosts and vulnerabilities should be considered into measuring security. To this end, Ritchey et al. [104] propose a method using model checking to analyze the network vulnerabilities. The model checker is considered as an attack path generating system with input information from the network environment. Four main descriptions are chosen as inputs to a model checking to find out whether an attackable path is existent. With the definition of

preconditions and goals, model checkers give concrete attack paths of examined network systems. This is the first work related to attack graphs, which opens a new era of network security analysis. Observing that finding one attack path in a network system is not enough, Sheyner et al. propose the idea of finding all attackable paths in an integrated system [110]. This work uses a model checking algorithm to replace the former hand-making attack graph method. Several algorithms are proposed to help to generate attack graphs more efficiently, Ammann et al. [6] reduce the complexity of this analysis from exponential to polynomial by assuming that an attacker never needs to go back to restore a lost privilege.

Tools and model checker information are presented in [30], [93], [139], [118]. Farmer et al. [30] describe a tool, COPs, which is used to reconfigure sets of programs and shell scripts to help administrators to check for potential security holes in systems. COPs computerize oracle and password system, which is used to detect potential security problems and to report to system administrators. Phillips et al. [93] present a graph-based method to analyze network vulnerabilities. This analysis system identifies highly probable attack paths, which could be used to test a system when certain changes are applied, e.g., configuration changes, deployment of an IDS. Zerkle et al. [139] present a useful model-checker to help to implement attack graphs and to validate of network security metrics. Swiler et al. develop a robust attack graph tool with graphical interface for linking vulnerabilities and configuring attack graphs [118]. Furthermore, Sheyner et al. [111] present toolkit for generating and analyzing attack graphs.

## 2.2 Security Metrics

Ortalo et al. [85] model the minimal effort that an attacker needs to exploit vulnerabilities as a metric to evaluate the security level of systems. Similar to Ortalo et al. 's work, Balzarotti et al. [8] propose minimum efforts required for executing each exploit as a metric. To consider attack path as security metric, Pamula et al. [89] propose a metric to measure the

security strength of a network regrading the strength of the weakest attacker (who has the lowest abilities to compromise this system). With a customized algorithm, this metric computes minimal sets of required initial attributes for the lowest ability attacker to successfully compromise a system. These metrics are designed from the attackers' perspective.

Using the lowest ability of the attacker as the measurement for systems cannot represent the average ability of the attackers. Leversage et al. [63] use the mean time-to-compromise, which learn from physical security, as a security metric to measure systems. Decision-maker learns improvement decision by simply calculating the MMTC, which is the longer the safer. By considering investments and reductions in expected loss, Ryan et al. [105] design a quantitative risk management metric, which uses the reduction in expected loss to measure the success of information security investments.

Most of the above metrics are relatively simple and are designed at an early stage of the research on network security metric. Simple metrics are easy to apply and to understand, but they may not be suitable for complicated and ever changing network system environments. Wang et al. [129] define the attack resistance metric for assessing and comparing the security of different network configurations. Attack resistance metric measures ability of systems defending against attacks, which could suggest administrators potential better configurations in network systems.

Wang et al. [125] propose aggregating vulnerability probabilities from the Common Vulnerability Scoring System (CVSS). The probability of a vulnerability indicates the successful exploitation rates for a given vulnerability. However, this work focuses only on vulnerabilities, which ignores the interdependencies inside attack graph. To address this limitation, Frigault et al. [31] provide the Bayesian Networks-based model to aggregate vulnerabilities inside network systems. Bayesian networks are defined with nodes representing variables and arcs representing conditional independence among the variables. However, the formal notion of this metric is not defined in this work. Frigault et al. [32]

extend their work from Bayesian-networks to Dynamic Bayesian Networks to cope with the ever changing network environments. Although the dynamic situation has some limitations, the clear definition in Bayesian-network inspires later work in network security metrics.

Noel et al. [82] propose a metric, computing minimum required resources to compromise a system for hardening purpose. The previous hardening methods usually break attack paths to harden a network, however, this work focuses to find the initial necessary set for the attackers. Therefore, the attack actions would be break easily by disabling the initial conditions. Wang et al. [128] improve this solution by defining a clear notation of minimum-cost network hardening based on attack graphs.

Security metrics are also developed for specific applications, such as the information-theoretic metrics for measuring the effectiveness of IDSs [61], [37]. Gu et al. build a formal framework using information theory for analyzing and quantifying the effectiveness of the IDS. This work presents a formal IDS model, analyzing the information-theoretic approach.

Similar works in other areas exist, e.g., some of the design principles are proposed for developing metrics for trust [100], [99]. Authenticating entities in large-scale system always use authentication, which means each entity is able to authenticate the next path. This technology has been extended to multiple paths. Reiter et al. propose metrics to evaluate the confidence afforded by a set of paths. These two papers illustrate the principles for designing confidentiality metrics. They proposed a novel metric, which satisfies the principles.

## 2.3 Network Diversity Metric

The research on security metrics has attracted much attention lately. Unlike existing work which aim to measure the amount of network security [43, 130], the network diversity

metrics focus on diversity as one particular property of networks which may affect security. Nonetheless, our work borrows from the popular software security metric, attack surface [67], the general idea of focusing on interfaces (remotely accessible resources) rather than internal details (e.g., local applications). Our least attacking effort-based diversity metric is derived from the $k$-zero day safety metric [127, 126], and our probabilistic diversity metric is based on the attack likelihood metric [32, 129]. Another notable work evaluates security metrics against real attacks in a controlled environment [41], which provides a future direction to better evaluate our work. One limitation of our work lies in the high complexity of analyzing a resource graph; high level models of resource dependencies [56] may provide coarser but more efficient solutions to modeling diversity.

The idea of using design diversity for fault tolerance has been investigated for a long time. The N-version programming approach generates $N \geq 2$ functionally equivalent programs and compares their results to determine a faulty version [7], with metrics defined for measuring the diversity of software and faults [76]. The main limitation of design diversity lies in the high complexity of creating different versions, which may not justify the benefit [64]. The use of design diversity as a security mechanism has also attracted much attention [68]. The general principles of design diversity is shown to be applicable to security as well in [65]. The N-Variant system extends the idea of N-version programming to detect intrusions [20], and the concept of behavioral distance takes it beyond output voting [34]. Different randomization techniques have been used to automatically generate diversity [12, 55, 108, 13].

In addition to design diversity and generated diversity, recent work employ opportunistic diversity which already exists among different software systems. The practicality of employing OS diversity for intrusion tolerance is evaluated and the feasibility of using opportunistic diversity already existing between different OSes to tolerate intrusions is demonstrated in [35]. Diversity has also been applied to intrusion tolerant systems which

usually implement some kinds of Byzantine Fault Tolerant (BFT) replication as fault tolerance solutions [19]. A generic architecture for implementing intrusion-tolerant Web servers based on redundancy and diversification principles is introduced in [107]. Components-off-the-shelf (COTS) diversity is employed to provide an implicit reference model, instead of the explicit model usually required, for anomaly detection in Web servers [121]. Diversity could play an important role in addressing various security issues in cloud computing [101], such as using diverse authorities for efficient decryption and revocation in cloud storage [134] and using diverse access policies for increasing the security of cloud data [133].

## 2.4 Network Attack Surface Metric

The concept of attack surface is originally proposed for specific software and requires domain-specific expertise to formulate and implement [42]. Later on, the concept is generalized using formal models and becomes applicable to all software [87]. Furthermore, it is refined and applied to large scale software, and its calculation can be assisted by automatically generated call graphs [88, 67]. Attack surface has attracted significant attentions over the years. It is used as a metric to evaluate Android's message-passing system [54], in kernel tailing [59], and also serves as a foundation in Moving Target Defense, which basically aims to change the attack surface over time [49, 48]. Others aim to expand the scope of this concept in other domains, such as the six-way attack surfaces between users, services, and cloud systems [36], and the approximation of attack surface for modern automobiles [17]. The study on automating the calculation of attack surface is another interesting domain, e.g., COPES uses static analysis from bytecode to calculate attack surface and to secure permission-based software[9]. Stack traces from user crash reports is used to approximate attack surface automatically [120]. Despite such tremendous interest in the attack surface concept, to the best of our knowledge, little work exists on formally defining attack surface

at the network level. The correlation between attack surface and vulnerabilities has also been investigated, such as using attack surface entry points and reachability to assess the risk of vulnerability [138]. A study about the relationship between attack surface and the vulnerability density is given in [137], although the result is only based on two releases of Apache HTTP Server, which gives little clue to the general existence of such a correlation.

As to security metrics in general, there exist standardization efforts on vulnerability assessment including the Common Vulnerability Scoring System (CVSS) [73], which measures vulnerabilities in isolation. The NIST's efforts on standardizing security metrics are also given in [79] and more recently in [117]. The research on security metrics has attracted much attention lately [51]. Earlier work include the a metric in terms of time and efforts based on a Markov model [22]. More recently, several security metrics are proposed by combining CVSS scores based on attack graphs [125, 32]. The minimum efforts required for executing each exploit is used as a metric in [8, 89]. A mean time-to-compromise metric is proposed based on the predator state-space model (SSM) used in the biological sciences in [63]. While those metrics are mostly developed for known vulnerabilities, fewer work are capable of dealing with zero day attacks. A few exceptions include an empirical study of the total number of zero day vulnerabilities available on a single day based on existing data [71], an effort on ordering different applications in a system by the seriousness of consequences of having a single zero day vulnerability [44], and more recently the $k$-zero day safety model [127, 126] and the network diversity model [131, 140] both attempt to model the risk of zero day vulnerabilities, but their common limitation is the lack of capability in distinguishing different resources' likelihood of having such vulnerabilities. The network attack surface metric distinguish the resources with attack likelihood from the three dimensions in attack surface concept and aggregate the attack likelihood on the network level to model the risk of the zero day vulnerabilities, which address the limitations from the previous works.

## 2.5 Vulnerability Discovery Model

Two major vulnerability discovery models (VDM) have been studied in the literature; one focuses on studying the features that correlate with the vulnerable components in a software application; the other one focuses on using mathematic models to fit the vulnerability discovery model with the historical data to predict the future number of vulnerability for one application.

Zimmermann et al. [142] analyze the possibility of predicting vulnerable component in Windows Vista by using Logistic Regression for five groups of metrics, churn, complexity, coverage, dependency, and organizational structure of the company. The binary results have been evaluated with tenfold cross-validation, which yields precision below 67% and recall below 21%. Meneely et al. [74] study the developer-activity metrics and software vulnerabilities. The precision and recall from the Bayesian network predictive model are between 12%-29% and 32% to 56%, respectively. Doyle et al. [25] study the relationships between software metrics and vulnerable components in 14 open-source web applications. Spearman's rank correlation is computed between the metrics and security resources indicator (SRI), which is defined by the author and obtained from security scanners. Shin and Williams [112] perform binary classification using Logistic Regression with tenfold cross-validation to analyze the relationship between complexity, code-churn and developer-activity (CCD) metrics and the vulnerabilities. A later work [113] studies the relationship between 18 complexity metrics, five code churn metrics and fault history metric and the vulnerable components. The recall and the precision from this study are 83% and 11% respectively.

Perl et al. [92] analyze the effects of meta data in the code repositories with code metrics to predict the vulnerable commits. The precision of VCCFinder is 60% when recall is 24%. Younis et al. [136] study the relationship between software metrics and the vulnerable function in existing exploits. In total 183 vulnerabilities from National Vulnerability

Database for Linux Kernel and Apache HTTP server have been examined in their study. Stuckman et al. [116] add the token that generated from source code in the study to identify the vulnerable components. Walden et al. [124] compare the predictive powers between software metrics and text mining in predicting vulnerable components. Davari et al. [24] study the Hardly Reproducible Vulnerabilities (HRV) in the code level, and they achieve the precision of 82% and recall of 84% to classify vulnerable files into HRV-prone or non HRV-prone files.

Mathematic VDM focuses on modeling the discovery process of software vulnerabilities by evaluating the number of vulnerabilities with time. The existing models are Linear [77], Exponential [102], Alhazmi Malaiya Logistic (AML) [4], and effort based model. VDMs are usually mathematic models with parameters in which real vulnerability histories are required in obtaining the model. Those models are specific to software applications due to the fact that the real vulnerability history data is needed in the model establishing. Normally, large history data are needed to obtain better fitted model.

Different than the existing models, our approaches gather the features from five different metrics to prediction the number of vulnerabilities in application level (not the vulnerable component). Unlike the fact that mathematic VDMs requires large amount of historical data, our models study the important features from the metrics to dependent our prediction from the historical data.

# Chapter 3

# Network Diversity: A Security Metric for Evaluating the Resilience of Networks Against Zero-Day Attacks

In this chapter, we describe our efforts on the first proposed research topic, i.e., network diversity metrics.

## 3.1 Introduction

Dealing with unknown vulnerabilities is clearly a challenging task. Although we already have many effective solutions (e.g., IDS/IPS, firewalls, antivirus, software upgrading and patching) for tackling known attacks, zero day attacks are known to be difficult to mitigate due to lack of information.

To this end, diversity has long been regarded as a valuable solution because it may improve the resilience of a software system against both known and unknown vulnerabilities [65]. Security attacks exploiting unknown vulnerabilities may be detected and tolerated

as Byzantine faults by comparing either the outputs [20] or behaviors [34] of multiple software replicas or variants [19]. Although the earlier diversity-by-design approaches usually suffer from prohibitive development and deployment cost, recent works show more promising results on employing either opportunistic diversity [35] or automatically generated diversity [12, 55, 13]. More recently, diversity has found new applications in cloud computing security [101], Moving Target Defense (MTD) [49], resisting sensor worms [135], and network routing [16]. Most of those existing efforts rely on either intuitive notions of diversity or models mostly designed for a single system running diverse software replicas or variants.

However, at a higher abstraction level, as a global property of an entire network, the concept of *network diversity* and its effect on security has received limited attention. In this topic, we take the first step towards formally modeling network diversity as a security metric for the purpose of evaluating the resilience of networks with respect to zero day attacks.

We describe several use cases in order to motivate our study and illustrate various requirements and challenges in modeling network diversity.

## 3.2 Use Cases

We describe several use cases in order to motivate our study and illustrate various requirements and challenges in modeling network diversity. Some of those use cases will also be revisited in later sections.

**Use Case 1: Stuxnet and SCADA Security** Stuxnet is one of the first malware that employ multiple (four) different zero day attacks [29]. This indicates, in a mission critical system, such as supervisory control and data acquisition (SCADA) in this case, the risk of zero day attacks and multiple unknown vulnerabilities is real, and consequently network

administrators would need a systematic way for evaluating such a risk. However, this is a challenging task due to the lack of prior knowledge about vulnerabilities or attacking methods.

A closer look at Stuxnet's attack strategies would reveal how network diversity may help here. Stuxnet targets the programmable logic controllers (PLCs) on control systems of gas pipelines or power plants [29], which are mostly programmed using Windows machines not connected to networks. Therefore, Stuxnet adopts a multi-stage approach, by first infecting Windows machines owned by third parties (e.g., contractors), next spreading to internal Windows machines through the LAN, and finally covering the last hop through removable flash drives [29]. Clearly, the degree of software diversity along potential attack paths leading from the network perimeter to the PLCs can be regarded as a critical metric of the network's resilience against a threat likes Stuxnet. Our objective in this chapter is to provide a rigorous study of such network diversity metrics.

**Use Case 2: Worm Propagation** To make our discussion more concrete, we will refer to the running example shown in Figure 1 from now on. In this use case, our main concern is the potential propagation of worms or bots inside the network. A common belief here is that we can simply count the number (percentage) of distinct resources in the network as diversity. Although such a definition is natural and intuitive, it clearly has limitations.

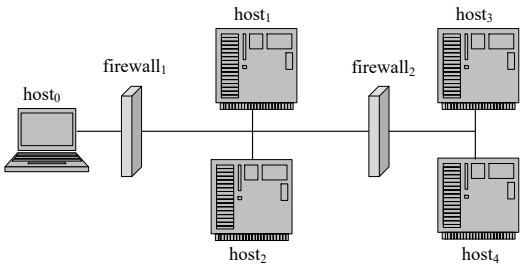



Figure 1: The Running Example

For example, suppose host 1, 2, and 3 are Web servers running IIS, all of which access

files stored on host 4. Clearly, the above count-based metric would indicate a lack of diversity and suggest replacing IIS with other software to prevent a worm from infecting all three at once. However, it is easy to see that, even if a worm can only infect one Web server after such a diversification effort (e.g., it can infect IIS but not Apache), it can still propagate to all four hosts through the network share on host 4 (e.g., it may infect certain executable files stored on host 4, which are subsequently accessed by all Web servers). The reason that this naive approach fails in this case is that it ignores the existence of causal relationships between resources (due to the network share). Therefore, after we discuss the count-based metric in Section 3.3, we will address this limitation with a *goal oriented* approach in Section 3.4.

**Use Case 3: Targeted Attack** Suppose that we are more concerned with a targeted attack on the storage server, host 4. Following above discussions, an intuitive solution is to diversify resources along any *path* leading to the critical asset (host 4), e.g., between hosts 1 (or 2, 3) and host 4. Although this is a valid observation, realizing it requires a rigorous study of the causal relationships between different resources, because host 4 is only as secure as the weakest path (representing the least attacking effort) leading to it. We will propose a formal metric based on such an intuition in Section 3.4.

On the other hand, the least attacking effort by itself only provides a partial picture. Suppose now host 1 and 2 are diversified to run IIS and Apache, respectively, and firewall 2 will only allow host 1 and 2 to reach host 4. Although the least attacking effort has not changed, this diversification effort has actually provided attackers more opportunities to reach host 4 (by exploiting either IIS or Apache). That is, misplaced diversity may in fact hurt security. This will be captured by a probabilistic metric in Section 3.5.

**Use Case 4: MTD** Moving Target Defense (MTD) can be considered as a different approach to applying diversity to security, since it diversifies resources along the time dimension [49]. However, most existing work on MTD rely on intuitive notion of diversity which may lead to misleading results. This next case demonstrates the usefulness of our proposed metrics particularly for MTD. In this case, suppose host 1 and 2 are Web servers, host 3 an application server, and host 4 a database server. A MTD will attempt to achieve better security by varying in time the software components at different tiers. A common misconception here is that the combination of different components at different tiers will increase diversity, and the degree of diversity is equal to the product of diversity at those tiers. However, this is usually not the case. For example, a single flaw in the application server (host 3) may result in a SQL injection that compromises the database server (host 4) and consequently leaks the root user's password. Also, similar to the previous case, more diversity over time may actually provide attackers more opportunities to find flaws. The lesson here is again that an intuitive observation may be misleading, and formally modeling network diversity is necessary.

## 3.3 Biodiversity-Inspired Network Diversity Metric

Although the notion of network diversity has attracted limited attention, its counterpart in ecology, *biodiversity*, and its positive impact on the ecosystem's stability has been investigated for many decades [27]. While many lessons may potentially be borrowed from the rich literature of biodiversity, in this chapter we will focus on adapting existing mathematical models of biodiversity for modeling network diversity.

Specifically, the number of different species in an ecosystem is known as *species richness* [94]. Similarly, given a set of distinct resource types $R$ (we will consider similarity between resources later) in a network, we call the cardinality $|R|$ the *richness* of resources in the network. An obvious limitation of this richness metric is that it ignores the relative

abundance of each resource type. For example, the two sets $\{r_1, r_1, r_2, r_2\}$ and $\{r_1, r_2, r_2, r_2\}$ share the same richness of *2* but clearly different levels of diversity.

To address this limitation, the Shannon-Wiener index, which is essentially the Shannon entropy using natural logarithm, is used as a *diversity index* to group all systems with the same level of diversity. The exponential of the diversity index is regarded as the *effective number* metric [39]. The effective number basically allows us to always measure diversity in terms of the number of equally-common species, even if in reality those species may not be equally common. In the following equation, we borrow this concept to define the effective resource richness and our first diversity metric.

**Definition 1** (Effective Richness and $d_1$-Diversity)**.** *In a network $G$ with the set of hosts $H = \{h_1, h_2, \ldots, h_n\}$, set of resource types $R = \{r_1, r_2, \ldots, r_m\}$, and the resource mapping $res(.) : H \rightarrow 2^R$ (here $2^R$ denotes the power set of $R$), let $t = \sum_{i=1}^{n} \mid res(h_i) \mid$ (total number of resource instances), and let $p_j = \frac{|\{h_i : r_j \in res(h_i)\}|}{t} (1 \leq i \leq n, 1 \leq j \leq m)$ (relative frequency of each resource). We define the network's diversity as $d_1 = \frac{r(G)}{t}$, where $r(G)$ is the network's effective richness of resources, defined as*

$$r(G) = \frac{1}{\prod_1^n p_i^{p_i}}$$

One limitation of the effective number-based metric is that similarity between different resource types is not taken into account and all resource types are assumed to be entirely different, which is not realistic (e.g., the same application can be configured to fulfill totally different roles, such as Nginx as a reverse proxy or a web server, respectively, in which case these should be regarded as different resources with high similarity). Therefore, we borrow the similarity-sensitive biodiversity metric recently introduced in [62] to re-define resource richness. With this new definition, the above diversity metric $d_1$ can now handle similarity between resources.

**Definition 2** (Similarity-Sensitive Richness)**.** *In Definition 1, suppose a similarity function is given as* $z(.) : [1,m] \times [1,m] \rightarrow [0,1]$ *(a larger value denoting higher similarity and* $z(i,i) = 1$ *for all* $1 \leq i \leq m$*), let* $zp_i = \sum_{j=1}^{m} z(i,j)p_j$*. We define the network's effective richness of resources, considering the similarity function, as*

$$r(G) = \frac{1}{\prod_1^n zp_i^{p_i}}$$

The effective richness-based network diversity metric $d_1$ is only suitable for cases where all resources may be treated equally, and causal relationships between resources either do not exist or may be safely ignored. On the other hand, this metric may also be used as a building block inside other network diversity metrics, in the sense that we may simply say "the number of distinct resources" without worrying about uneven distribution of resource types or similarity between resources, thanks to the effective richness concepts given in Definition 1 and 2.

The effect of biodiversity on the stability of an ecosystem has been shown to critically depend on the interaction of different species inside a food Web [69]. Although such interaction typically takes the form of a "feed-on" relationship between different species, which does not directly apply to computer networks, this observation has inspired us to model diversity based on the structural relationship between resources, which will be detailed in the coming sections.

## 3.4 Least Attacking Effort-Based Network Diversity Metric

This section models network diversity based on the least attacking effort.The heuristic algorithm to find $d_2$ are described in [131]. We don't put detailed algorithm here.

### 3.4.1 The Model

In order to model diversity based on the least attacking effort while considering causal relationships between different resources, we first need a model of such relationships and possible zero day attacks. Our model is similar to the *attack graph* model [110, 6], although our model focuses on remotely accessible resources (e.g., services or applications that are reachable from other hosts in the network), which will be regarded as placeholders for potential zero day vulnerabilities instead of known vulnerabilities as in attack graphs.

To build intuitions, we revisit Figure 1 by making the following assumptions. Accesses from outside firewall 1 are allowed to host 1 but blocked to host 2; accesses from host 1 or 2 are allowed to host 3 but blocked to host 4 by firewall 2; hosts 1 and 2 provide *http* service; host 3 provides *ssh* service; Host 4 provides both *http* and *rsh* services.

Figure 2 depicts a corresponding *resource graph*, which is syntactically equivalent to an attack graph, but models zero day attacks rather than known vulnerabilities. Each pair in plaintext is a self-explanatory security-related condition (e.g., connectivity $\langle$ *source, destination* $\rangle$ or privilege $\langle privilege, host \rangle$), and each triple inside a box is a potential exploit of resource $\langle resource$, *source host*$, destination\ host \rangle$; the edges point from the pre-conditions to a zero day exploit (e.g., from $\langle 0,1 \rangle$ and $\langle user, 0 \rangle$ to $\langle http, 0, 1 \rangle$), and from that exploit to its post-conditions (e.g., from $\langle http, 0, 1 \rangle$ to $\langle user, 1 \rangle$). Exploits or conditions involving firewall 2 are omitted for simplicity.

We simply regard resources of different types as entirely different (their similarity can be handled using the effective resource richness given in Definition 2). Also, we take the conservative approach of considering all resources (services and firewalls) to be potentially vulnerable to zero day attacks. Definition 3 formally introduces the concept of resource graph.

**Definition 3** (Resource Graph)**.** *Given a network with the set of hosts H, set of resources R with the resource mapping* $res(.) : H \rightarrow 2^R$*, set of zero day exploits* $E = \{\langle r, h_s, h_d \rangle \mid h_s \in$

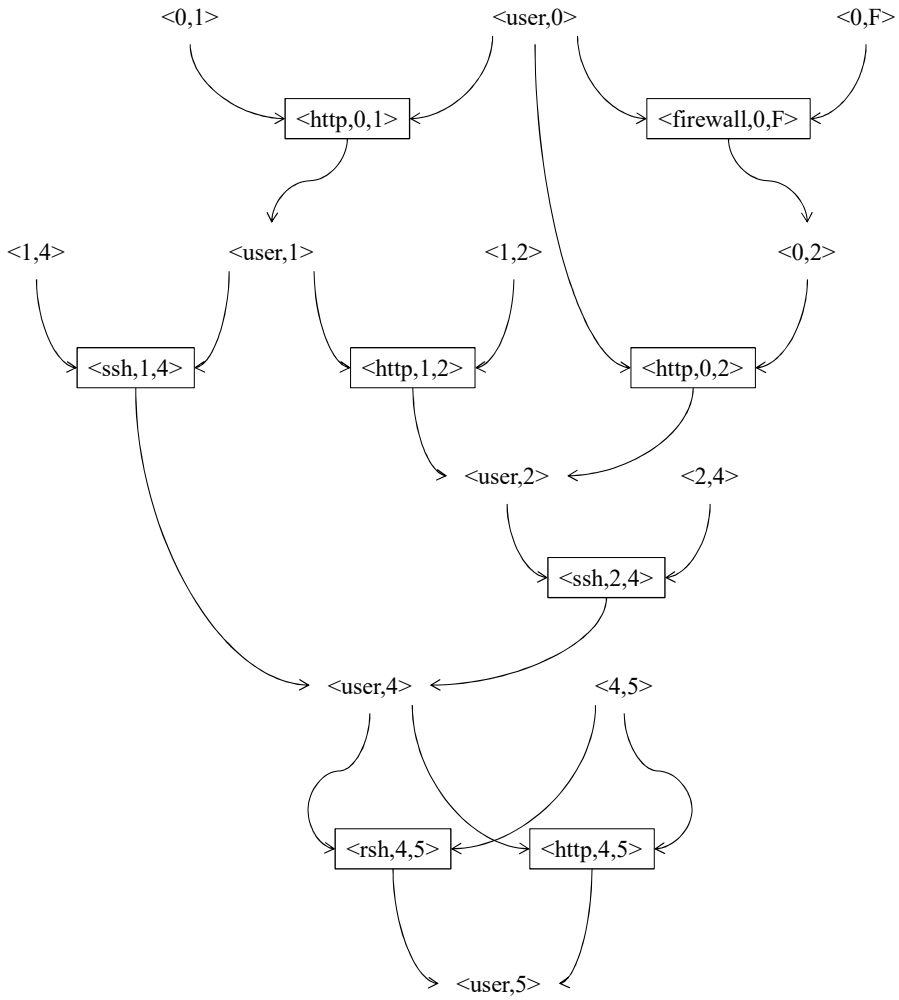



Figure 2: An Example Resource Graph

$H, h_d \in H, r \in res(h_d)\}$ *and their pre- and post-conditions C, a resource graph is a directed graph* $G(E \cup C, R_r \cup R_i)$ *where* $R_r \subseteq C \times E$ *and* $R_i \subseteq E \times C$ *are the pre- and post-condition relations, respectively.*

Next, we consider how attackers may potentially attack a critical network asset, modeled as a goal condition, with the least effort. In Figure 2, by following the simple rule that an exploit may be executed if all the pre-conditions are satisfied, and executing that exploit will cause all the post-conditions to be satisfied, we may observe six *attack paths*, as shown in Table 1 (the second and third columns can be ignored for now and will be explained shortly). Definition 4 formally introduces the concept of attack path.

| Attack Path | # of Steps | # of Resources |
|---|---|---|
| 1. $\langle http,0,1\rangle \rightarrow \langle ssh,1,4\rangle \rightarrow \langle rsh,4,5\rangle$ | 3 | 3 |
| 2. $\langle http,0,1\rangle \rightarrow \langle ssh,1,4\rangle \rightarrow \langle http,4,5\rangle$ | 3 | 2 |
| 3. $\langle http,0,1\rangle \rightarrow \langle http,1,2\rangle \rightarrow \langle ssh,2,4\rangle \rightarrow \langle rsh,4,5\rangle$ | 4 | 3 |
| 4. $\langle http,0,1\rangle \rightarrow \langle http,1,2\rangle \rightarrow \langle ssh,2,4\rangle \rightarrow \langle http,4,5\rangle$ | 4 | 2 |
| 5. $\langle firewall,0,F\rangle \rightarrow \langle http,0,2\rangle \rightarrow \langle ssh,2,4\rangle \rightarrow \langle rsh,4,5\rangle$ | 4 | 4 |
| 6. $\langle firewall,0,F\rangle \rightarrow \langle http,0,2\rangle \rightarrow \langle ssh,2,4\rangle \rightarrow \langle http,4,5\rangle$ | 4 | 3 |

Table 1: Attack Paths

**Definition 4** (Attack Path)**.** *Given a resource graph $G(E \cup C, R_r \cup R_i)$, we call $C_I = \{c : c \in C, (\nexists e \in E)(\langle e,c\rangle \in R_i)\}$ the set of initial conditions. Any sequence of zero day exploits $e_1, e_2, \ldots, e_n$ is called an attack path in G, if $(\forall i \in [1,n])(\langle c,e_i\rangle \in R_r \rightarrow (c \in C_i \vee (\exists j \in [1,i-1])(\langle e_j,c\rangle \in R_i)))$, and for any $c \in C$, we use $seq(c)$ for the set of attack paths $\{e_1,e_2,\ldots,e_n : \langle e_n,c\rangle \in R_i\}$.*

We are now ready to consider how diversity could be defined based on the least attacking effort (the shortest path). There are actually several possible ways for choosing such shortest paths and for defining the metric, as we will illustrate through our running example in the following.

- First, as shown in the second column of Table 1, path 1 and 2 are the shortest in terms of the *steps* (i.e., the number of zero day exploits). Clearly, those do not reflect the least attacking effort, since path 4 may actually take less effort than path 1, as attackers may reuse their exploit code, tools, and skills while exploiting the same *http* service on three different hosts.

- Next, as shown in the third column, path 2 and 4 are the shortest in terms of the number of distinct resources (or effective richness). This seems more reasonable since it captures the saved effort in reusing exploits. However, although path 2 and 4 have the same number of distinct resources (2), they clearly reflect different diversity.

- Another seemingly valid solution is to base on the minimum ratio $\frac{\#\ of\ resources}{\#\ of\ steps}$ (which

is given by path 4 in this example), since such a ratio reflects the potential improvements in terms of diversity (e.g., the ratio $\frac{2}{4}$ of path 4 indicates 50% potential improvement in diversity). However, we can easily imagine a very long attack path minimizing such a ratio but does not reflect the least attacking effort (e.g., an attack path with 9 steps and 3 distinct resources will yield a ratio of $\frac{1}{3}$, less than $\frac{2}{4}$, but clearly requires more effort than path 4).

- Finally, yet another option is to choose the shortest path that minimizes both the number of distinct resources (path 2 and 4) and the above ratio $\frac{\#\ of\ resources}{\#\ of\ steps}$ (path 4). However, a closer look will reveal that, although path 4 does represent the least attacking effort, it does not represent the maximum amount of potential improvement in diversity, because once we start to diversify path 4, the shortest path may change to be path 1 or 2.

Based on these discussions, we define network diversity by combining the first two options above. Specifically, the network diversity is defined as the ratio between the minimum number of distinct resources on a path and the minimum number of steps on a path (note these can be different paths). Going back to our running example above, we find path 2 and 4 to have the minimum number of distinct resources (two), and also path 1 and 2 to have the minimum number of steps (three), so the network diversity in this example is equal to $\frac{2}{3}$ (note that it is a simple fact that this ratio will never exceed 1). Intuitively, the numerator 2 denotes the network's current level of robustness against zero day exploits (no more than 2 different attacks), whereas the denominator 3 denotes the network's maximum potential of robustness (tolerating no more than 3 different attacks) by increasing the amount of diversity (from $\frac{2}{3}$ to 1). More formally, we introduce our second network diversity metric in Definition 5 (note that, for simplicity, we only consider a single goal condition for representing the given critical asset, which is not a limitation since multiple goal conditions can be easily handled through adding a few dummy conditions [3]).

**Definition 5** ($d_2$-Diversity)**.** *Given a resource graph $G(E \cup C, R_r \cup R_i)$ and a goal condition $c_g \in C$, for each $c \in C$ and $q \in seq(c)$, denote $R(q)$ for $\{r : r \in R, r \text{ appears in } q\}$, the network diversity is defined as (where $min(.)$ returns the minimum value in a set)*

$$d_2 = \frac{min_{q \in seq(c_g)} \mid R(q) \mid}{min_{q' \in seq(c_g)} \mid q' \mid}$$

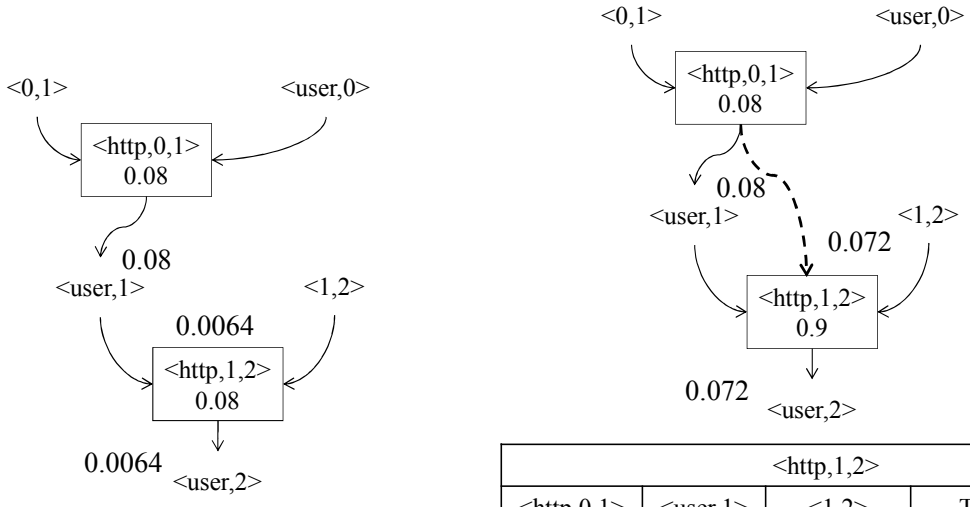
## 3.5 Probabilistic Network Diversity

In this section, we develop a probabilistic metric to capture the effect of diversity based on average attacking effort by combining all attack paths. The preliminary version of this paper [131] has proposed a probabilistic metric model for this purpose. We will first identify important limitations in this model, and then provide a redesigned model to address them.

### 3.5.1 Overview

This section first reviews the probabilistic model of network diversity introduced in [131] and then points out its limitations. This model defines network diversity as the ratio between two probabilities, namely, the probability that given critical assets may be compromised, and the same probability but with an additional assumption that all resource instances are distinct (which means attackers cannot reuse any exploit). Both probabilities represent the *attack likelihood* with respect to goal conditions, which can be modeled using a Bayesian network constructed based on the resource graph [32].

For example, Figure 3 demonstrates this model based on our running example (only part of the example is shown for simplicity). The left-hand side represents the case in which the effect of reusing an exploit is not considered, that is, the two *http* service instances are assumed to be distinct. The right-hand side considers that effect and models it as the conditional probability that the lower *http* service may be exploited given that the upper




| <http,1,2> | | | |
|---|---|---|---|
| <user,1> | <1,2> | T | F |
| T | T | 0.08 | 0.92 |
| T | F | 0 | 1 |
| F | T | 0 | 1 |
| F | F | 0 | 1 |

| <http,1,2> | | | | |
|---|---|---|---|---|
| <http,0,1> | <user,1> | <1,2> | T | F |
| T | T | T | 0.9 | 0.1 |
| T | T | F | 0 | 1 |
| T | F | T | 0 | 1 |
| F | T | T | 0 | 1 |
| F | T | F | 0 | 1 |
| F | F | F | 0 | 1 |

Figure 3: Modeling Network Diversity Using Bayesian Networks

one is already exploited (represented using a dotted line). The two conditional probability tables (CPTs) illustrate the effect of reusing the *http* exploit (e.g., probability 0.9 in the right CPT), and not reusing it (e.g., probability 0.08 in the left CPT), respectively. The network diversity in this case will be calculated as the ratio $d_3 = \frac{0.0064}{0.072}$.

We realized that the above model has certain limitations when a few invalid results (larger than 1) were returned during our simulations. More specifically, in the above model, modeling the effect of reusing exploits as a conditional probability (that a resource may be exploited given that some other instances of the same type are already exploited) essentially assumes a total order over different instances of the same resource type in any resource graph, which comprises a major limitation. For example, in Figure 4 (the dashed line and box, and the CPT table may be ignored for the time being), although the reused *http* exploit

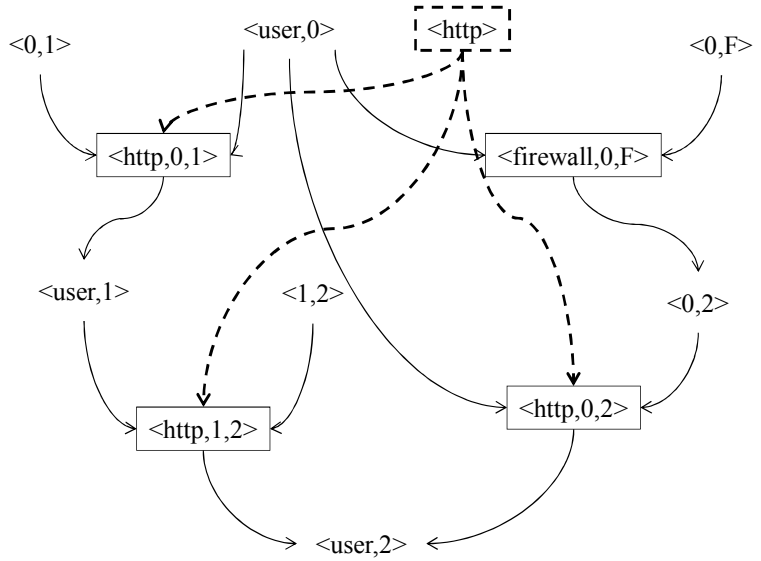



Figure 4: The Redesigned Model

$\langle http, 1, 2\rangle$ (after exploiting $\langle http, 0, 1\rangle$) may be handled using the above model by adding a dotted line pointing to it from its ancestor $\langle http, 0, 1\rangle$, the same method will not work for the other potentially reused *http* exploit $\langle http, 0, 2\rangle$, since there does not exist a definite order between $\langle http, 0, 1\rangle$ and $\langle http, 0, 2\rangle$, which means an attacker may reach $\langle http, 0, 2\rangle$ before, or after, reaching $\langle http, 0, 1\rangle$. Therefore, we cannot easily assume one of them to be exploited first. Considering that the resource graph model is defined based on a Bayesian network, which by definition requires acyclic graphs, we cannot add bi-directional dotted lines between exploits, either.

Another related limitation is that, once exploits are considered to be partially ordered, the attack likelihood will not necessarily be the lowest when all the resources are assumed to be distinct. For example, in Figure 4, an attacker may reach condition $\langle user, 2\rangle$ through two paths, $\langle http, 0, 1\rangle \rightarrow \langle http, 1, 2\rangle$ and $\langle firewall, 0, F\rangle \rightarrow \langle http, 0, 2\rangle$. Intuitively, the attack likelihood will actually be higher if the *http* exploits in the two paths are assumed to be distinct, since now an attacker would have more choices in reaching the goal condition $\langle user, 2\rangle$. Those limitations will be addressed in following sections through a redesigned model.

### 3.5.2 Redesigning $d_3$ Metric

To address the aforementioned limitations of the original $d_3$ metric [131], we redesign the model of reusing exploits of the same resource type. Intuitively, what allows an attacker to more likely succeed in exploiting a previously exploited type of resources is the knowledge, skills, or exploit code he/she has obtained. Therefore, instead of directly modeling the casual relationship between reused exploits, we explicitly model such advantages of the attacker as separate events, and model their effect of increasing the likelihood of success in subsequent exploits as conditional probabilities.

More specifically, a new parent node common to exploits of the same resource type will be added to the resource graph, as demonstrated in Figure 4 using dashed lines and box. This common parent node represents the event that an attacker has the capability to exploit that type of resources. However, unlike nodes representing initial conditions, which will be treated as evidence for calculating the posterior probability of the goal condition, the event that an attacker can exploit a type of resources will not be considered observable. Adding a common parent node to exploits of the same resource type will introduce probabilistic dependence between the children nodes such that satisfying one child node will increase the likelihood of others, which models the effect of reusing exploits.

For example, in Figure 4, the dashed line box indicates a new node $\langle http \rangle$ representing the event that an attacker has the capability to exploit *http* resources. The dashed lines represent conditional probabilities that an attacker can exploit each *http* instance, and the CPT table shows an example of such conditional probability for $\langle http, 1, 2 \rangle$. The marginal probability 0.08 assigned to $\langle http \rangle$ represents the likelihood that an attacker has the capability of exploiting *http* resources, and the conditional probability 0.9 assigned to $\langle http, 1, 2 \rangle$ represents the likelihood for the same attacker to exploit that particular instance. The existence of such a common parent will introduce dependence between those *http* exploits, such that satisfying one will increase others' likelihood.

Formally, Definition 6 characterizes network diversity using this approach. In the definition, the second set of conditional probabilities represent the probability that an attacker is capable of exploiting each type of resources. The third and fourth sets together represent the semantics of a resource graph. Finally, the fifth set represents the conditional probability that an exploit may be executed when its pre-conditions are satisfied (including the condition that represents the corresponding resource type).

**Definition 6** ($d_3$ Diversity)**.** *Given a resource graph $G(E \cup C, R_r \cup R_i)$, let $R' \subseteq R$ be the set of resource types each of which is shared by at least two exploits in $E$, and let $R_s = \{(r, \langle r, h_s, h_d \rangle) : r \in R', \langle r, h_s, h_d \rangle \in E\}$ (that is, edges from resource types to resource instances). Construct a Bayesian network $B = (G'(E \cup C \cup R', R_r \cup R_i \cup R_s), \theta)$, where $G'$ is obtained by injecting $R'$ and $R_s$ into the resource graph $G$, and regarding each node as a discrete random variable with two states $T$ and $F$, and $\theta$ is the set of parameters of the Bayesian network given as follows.*

*I.* $P(c = T) = 1$ *for all the initial conditions* $c \in C_I$.

*II.* $P(r = T)$ *are given for all the shared resource types* $r \in R'$.

*III.* $P(e \mid \exists c_{\langle c,e \rangle \in R_r} = F) = 0$ *(that is, an exploit cannot be executed until all of its pre-conditions are satisfied).*

*IV.* $P(c \mid \exists e_{\langle e,c \rangle \in R_i} = T) = 1$ *(that is, a post-condition can be satisfied by any exploit alone).*

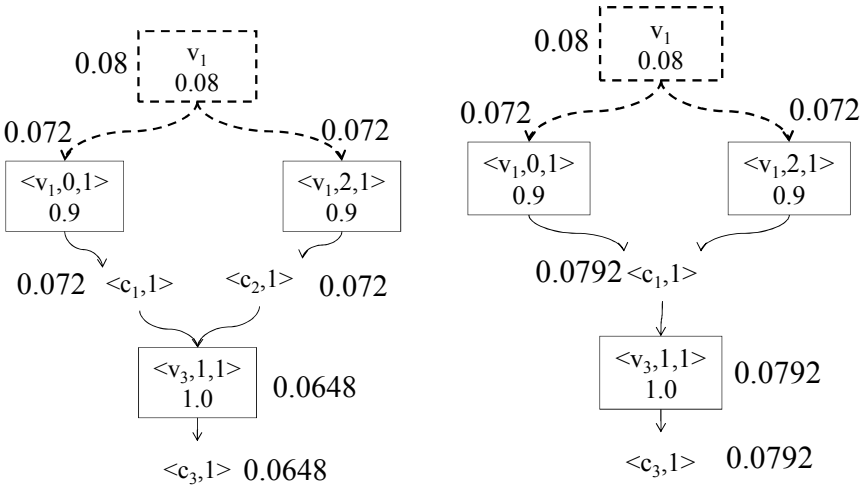
*V.* $P(e \mid \forall c_{\langle c,e \rangle \in R_r \cup R_s} = T)$ *are given for all* $e \in E$ *(that is, the probability of successfully executing an exploit when its pre-conditions have all been satisfied).*

*Given any $c_g \in C$, the network diversity $d_3$ is defined as $d_3 = \frac{p'}{p}$ where $p = P(c_g \mid \forall c_{c \in CI} = T)$ (that is, the conditional probability of $c_g$ being satisfied given that all the initial conditions are true), and $p'$ denotes the minimum possible value of $p$ when some edges are*

*deleted from $R_s$ (that is, the lowest attack likelihood by assuming certain resource types are no longer shared by exploits).*

Figure 5 shows two simple examples in which the first depicts a conjunction relationship between the two exploits (in the sense that both upper exploits must be executed before the lower exploit can be reached), whereas the second a disjunction relationship (any of the two upper exploits can alone lead to the lower exploit). In both cases, assuming $c_g = \langle c_3, 1\rangle$, the probability $p = P(c_g \mid \forall c_{c\in CI} = T)$ is shown in the figure. We now consider how to calculate the normalizing constant $p'$. For the left-hand side case, the probability $p = P(c_g \mid \forall c_{c\in CI} = T)$ would be minimized if we delete both edges from the top node ($v_1$) to its two children (that is, those two exploits no longer share the same resource type). It can be calculated that $p' = 0.0064$, and hence the diversity $d_3 = \frac{0.0064}{0.0648}$ in this case. The right-hand case is more interesting, since it turns out that $p$ is already minimized because deleting edges from the top node ($v_1$) will only result in a higher value of $p$ (since an attacker would have two different ways for reaching the lower exploit), which can be calculated as 0.1536. Therefore, diversity in this case is $d_3 = \frac{0.0792}{0.0792}$, that is, improving diversity will not enhance (in fact it hurts) security in this case. This example also confirms our earlier observation that assuming all resources to be distinct does not necessarily lead to the lowest attack likelihood.

The above example also leads to the observation that the normalizing constant $p'$ may not always be straightforward to calculate since finding the case in which $p$ is minimized essentially means we need to optimize a network's diversity for improving its security, which itself comprises an interesting future direction. Instead, we propose an approximated version of normalizing constant $p'$ based on following observations from the above example. In Figure 5, we can see that the right-hand side contains two attack paths leading to the goal condition $\langle c_3, 1\rangle$ (since each of the upper exploits alone is sufficient to lead to the lower exploit). We have shown previously that deleting dashed lines will only increase the




| <v3,1,1> | | | |
|---|---|---|---|
| <v1,0,1> | <v1,2,1> | T | F |
| T | T | 1 | 0 |
| T | F | 0 | 1 |
| F | T | 0 | 1 |
| F | F | 0 | 1 |

| <v3,1,1> | | | |
|---|---|---|---|
| <v1,0,1> | <v1,2,1> | T | F |
| T | T | 1 | 0 |
| T | F | 1 | 0 |
| F | T | 1 | 0 |
| F | F | 0 | 1 |

Figure 5: Two Examples of Applying $d_3$

probability $p$ (of reaching the goal condition). However, we can easily see that, whether we delete the dashed lines or not, the probability $p$ would always be minimized if there were only one path (e.g., by deleting $\langle v_1, 2, 1\rangle$ from the figure). Note that, for the left-hand side, there is already only one path since both upper exploits are required to reach the lower exploit, so $p$ is minimized when the two upper exploits are assumed to be distinct. Intuitively, the network's security can never exceed the case in which only the shortest path (in terms of the number of steps) remains in the resource graph, with no resource being shared along the path. This intuition leads to following result.

**Proposition 1.** *The normalizing constant $p'$ in Definition 6 always satisfies $p' \geq p''$ where $p''$ is the probability $P(c_g \mid \forall c_{c \in C_I} = T)$ calculated on the shortest attack path in terms of steps (see Section 3.4.1).*

*Proof (Sketch):* We prove the result by mathematical induction on $i$, the number of steps in the shortest path. The base case $i = 1$ is trivial. For the inductive case, suppose the result

holds for any resource graph with shortest path no longer than $k$. Given a resource graph $G$ whose shortest path has $k+1$ steps, let the set of exploits that are directly adjacent to the goal condition $c_g$ be $E_{k+1}$. Clearly, $P(c_g \mid \forall c_{c \in CI} = T) \geq P(e \mid \forall c_{c \in CI} = T)$ holds for all $e \in E_{k+1}$ since $c_g$ can be satisfied by any exploit in $E_{k+1}$ (and the probability of the disjunction of events cannot be smaller than the probability of any event). Without loss of generality, suppose $e_{k+1} \in E_{k+1}$ is the exploit next to $c_g$ on the shortest path, and we have $P(c_g \mid \forall c_{c \in CI} = T) \geq P(e_{k+1} \mid \forall c_{c \in CI} = T)$. Let $E_k$ be the set of exploits closest to $e_{k+1}$, $E \subseteq E_k$ be the set of exploits on the shortest path, and $e_k \in E$ be the exploit next to $e_{k+1}$. There cannot be any conjunctive relationships between the exploits in $E$ and any other exploit in $E_k \setminus E$ with respect to $e_{k+1}$, because otherwise the shortest path would have more than $k+1$ steps, contradicting our assumption. Therefore, we have that $P(c_g \mid \forall c_{c \in CI} = T) \geq P(e_{k+1} \mid \forall c_{c \in CI} = T) \geq P(e_k \mid \forall c_{c \in CI} = T) \cdot P(e_{k+1} \mid \forall c_{\langle c,e \rangle \in R_r \cup R_s} = T)$. Then by our inductive hypothesis, we have that $P(e_k \mid \forall c_{c \in CI} = T)$ must be no less than the same probability calculated on the shortest path (of length $k$), and hence conclude the proof. □

The above result simplifies the application of $d_3$ since the shortest path can be easily obtained using the heuristic algorithm mentioned in [131]. We apply the approach to our running example, as shown in Figure 2. Based on Table 1, the first and second attack paths have the lowest number of steps. The left-hand side of Figure 6 depicts the first path. The normalizing constant can be calculated based on this path as $p' = 5.12 * 10^{-4}$. The right-hand side depicts the application of our model for reusing exploits, which adds a common parent for the same type of resources, represented using dotted lines and boxes. There are two types of resources that are reused in this resource graph, *http* and *ssh*. By applying the method described above, we obtain the attack likelihood $p = 0.0052$, and therefore the network diversity can be calculated as $d_3 = \frac{p'}{p} = \frac{5.12 * 10^{-4}}{0.0052}$.

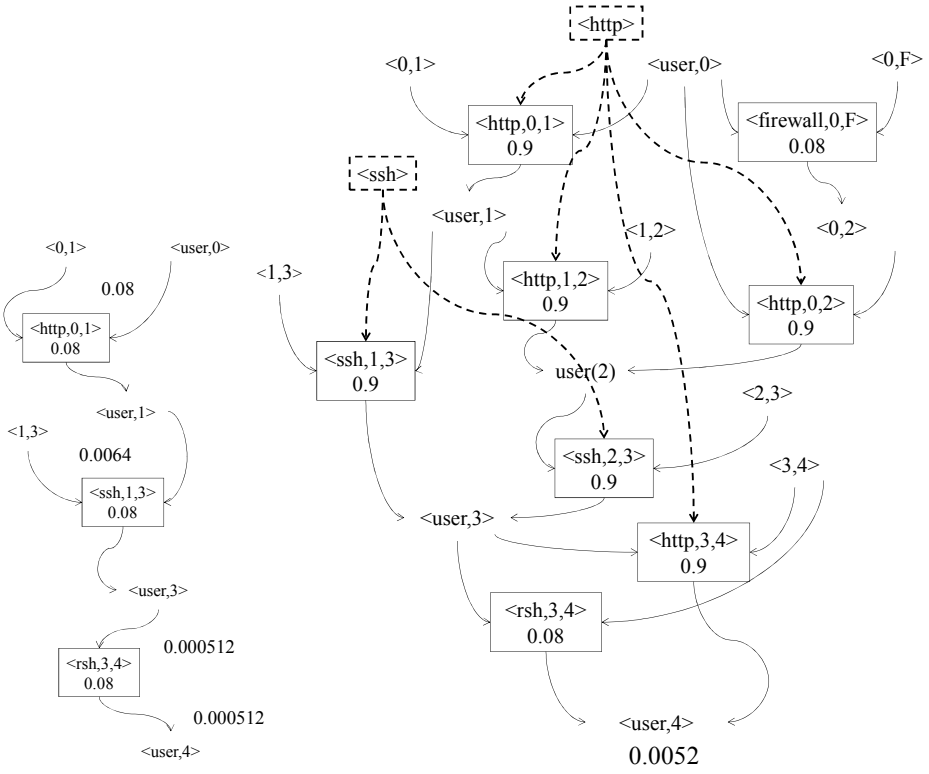



Figure 6: Applying $d_3$ on the Running Example

## 3.6 Applying the Network Diversity Metrics

The network diversity metrics we have proposed are based on abstract models of networks and attacks. How to instantiate such models for a given network is equally important. This section discusses various practical issues in applying the metrics and provides a case study on instantiating the models.

### 3.6.1 Guidelines for Instantiating the Network Diversity Models

To apply the proposed network diversity metrics, necessary input information needs to be collected. We describe how such inputs may be collected from a given network and discusses the practicality and scalability.

1 The $d_1$ Diversity Metric To instantiate $d_1$, we need to collect information about

- hosts (e.g., computers, routers, switches, firewalls),
- resources (e.g., remotely accessible services), and
- similarity between resources.

Information about hosts and resources is typically already available to administrators in the form of a network map. A network scanning will assist in collecting or verifying information about active services. A close examination of host configurations (e.g., the status of services and firewall rules) may also be necessary since a network scanning may not reveal services that are currently disabled or hidden by security mechanisms (e.g., firewalls) but may be re-enabled once the security mechanisms are compromised.

Collecting and updating such information for a large network certainly demands substantial time and efforts. Automated network scanning or host-based tools exist to help simplify such tasks. Moreover, focusing on remotely accessible resources allows our model to stay relatively manageable and scalable, since most hosts typically only have a few open ports but tens or even hundreds of local applications. A challenge is to determine the similarity of different but related resources, which will be discussed in further details in Section 3.8.

2 The $d_2$-Diversity Metric To instantiate the least attacking effort-based $d_2$ network diversity metric, we need to collect the following, in addition to what is already required by $d_1$,

- connectivity between hosts,
- security conditions either required for, or implied by, the resources (e.g., privileges, trust relationships, etc.), and
- critical assets.

The connectivity information is typically already available as part of the network map. A network scanner may help to verify such information. A close examination of host configurations (e.g., firewall rules) and application settings (e.g., authentication policies) is usually sufficient to identify the requirements for accessing a resource (pre-conditions), and an assessment of privilege levels of applications and the strength of isolation around such applications will reveal the consequences of compromising a resource (post-conditions). Critical assets can be identified based on an organization's needs and priority.

The amount of additional information required for applying $d_2$ is comparable to that required for $d_1$, since a resource typically has a small number of pre- and post-conditions. Once such information is collected, we can construct a resource graph using existing tools for constructing traditional attack graphs due to their syntactic equivalence, and the latter is known to be practical for realistic applications [86, 47].

3 The $d_3$-Diversity Metric

To instantiate the probabilistic network diversity metric $d_3$, we need to collect the following, in addition to what is already required for $d_2$,

- marginal probabilities of shared resource types, and
- conditional probabilities that resources can be compromised when all the pre-conditions are satisfied.

Both groups of probabilities represent the likelihood that attackers have the capability of compromising certain resources. A different likelihood may be assigned to each resource type, if this can be estimated based on experiences or reputations (e.g., the history of past vulnerabilities found in the same or similar resource). When such an estimation is not possible or desirable (note that any assumption about attackers' capabilities may weaken security if the assumption turns to be invalid), we can assign

the same nominal value as follows. Since a zero day vulnerability is commonly interpreted as a vulnerability not publicly known or announced, it can be characterized using the CVSS base metrics [73], as a vulnerability with a remediation level *unavailable*, a report confidence *unconfirmed*, and a maximum overall base score (and hence produce a conservative metric value). We therefore obtain a nominal value of 0.8, converting to a probability of 0.08. For reference purpose, the lowest existing CVSS score [81] is currently 1.7, so 0.08 is reasonably low for a zero day vulnerability. Once the probabilities are determined, applying $d_3$ amounts to constructing Bayesian networks and making probabilistic inferences based on the networks, which can be achieved using many existing tools (e.g., we use OpenBayes [33]). Although it is a well known fact that inference using Bayesian networks is generally intractable, our simulation results have shown that the particular inference required for applying the $d_3$ metric can actually be achieved under reasonable computational cost [131].

### 3.6.2 Case Study

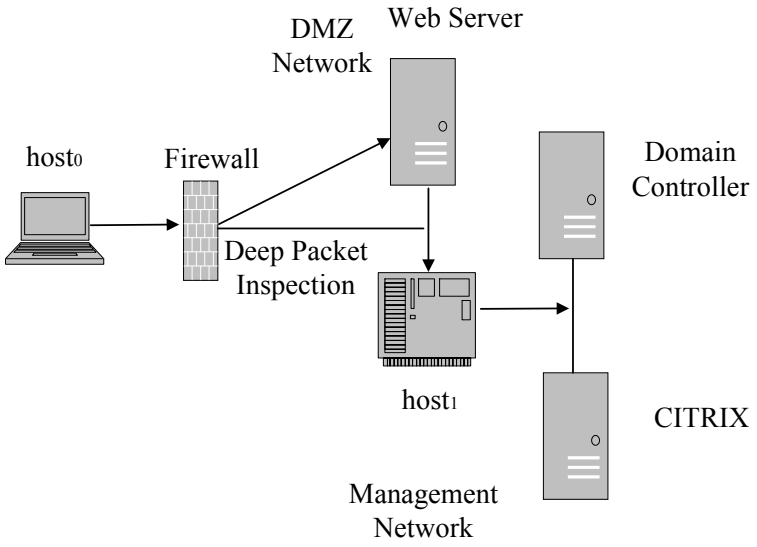



Figure 7: An Example Network [83]

We present a case study to demonstrate how our models may be instantiated by following the guidelines provided in previous section. The case study is based on a network configuration from the Penetration Testing Virtual Labs [83]. Despite its relatively small scale, the network configuration mimics a typical enterprise network, e.g., with DMZ, Web server behind firewall accessible from public Internet, and a private management network protected by the same firewall but with deep packet inspection and equipped with a domain controller and CITRIX server, as shown in Figure 7. The following describes in details how we collect necessary input information for instantiating each metric, and the collected information is listed in Table 2.

| Hosts | Connectivity | Ports | Resources | Security Conditions |
|---|---|---|---|---|
| Firewall | Web Server, $host_1$ | - | Egress traffic filtered, Deep content inspection rules | - |
| Web server | firewall,$host_1$ | 80,43 | Http Services, SQLite1.2.4 , Ubuntu 11.04 | user, root |
| $Host_1$ | firewall, web server, domain controller, citrix | 80,3389 | File Sharing, RDP Service, Windows 7, | domain user, local administrator |
| Citrix | domain controller, $host_1$ | 80,3389 | Http Services, Citrix Xen App, RDP Service | user, local administrator |
| Domain Controller | citrix,$host_1$ | 3389 | RDP Service | user,domain administrator |

Table 2: Collected Information

1 The $d_1$ Metric The information we collect for instantiating $d_1$ includes:

- Hosts: The network topological map clearly indicates these are the hosts: Firewall, Web Server, host$_1$, Citrix, and Domain Controller.

- Resources: The network configuration description indicates the firewall used in this network is Symantec Endpoint Protection, which deploys two different rules, for the DMZ network with egress traffic filtered and for the Management network with deep content inspection. We use nmap to scan the internal network in order to collect information about opening ports, applications running on the hosts and operating systems' information on the hosts, etc. For example, we determined that the public web server has opening ports 80 and 43, with SQLite and Appache running on top of Ubuntu 11.04.
- Similarity between resources: We take a simplistic approach of regarding resources in this network as either identical or different so the similarity score is either 1 or 0 (this can be refined by leveraging existing tools, as discussed in Section 3.8).

2 The $d_2$ Metric

To instantiate $d_2$, we need to collect the following, in addition to what is already collected for $d_1$:

- Connectivity: the network topological map clearly provides the connectivity between hosts.
- Security conditions: we study the applications and their existing vulnerabilities in order to collect corresponding security-related pre- and post- conditions. For example, SQLiteManager version 1.2.4 runs under user privilege on Web server, which indicates a post-condition of user privilege on the host, whereas Ubuntu 11.04 has root privilege as its post-condition due to potential privilege escalation vulnerabilities (there in fact exist such vulnerabilities [21]).
- Critical assets: in this network we consider the Domain Controller as critical asset due to its special role (actual system administrators will be in a better

position to designate their critical assets).

3 The $d_3$ Metric To instantiate $d_3$, we need to collect the following, in addition to what is already collected for $d_2$,

- Marginal probabilities of shared resource types and conditional probabilities that resources can be compromised when all the pre-conditions are satisfied: we assign 0.08 as a nominal value for both probabilities which may certainly be refined if additional information is available to administrators (see Section 3.6.1 for details).

## 3.7 Simulation

In this section, we study the three proposed metrics by applying them to different use cases through simulations. All simulation results are collected using a computer equipped with a 3.0 GHz CPU and 8GB RAM in the Python environment under Ubuntu 12.04 LTS. The Bayesian network-based metric is implemented using OpenBayes [33]. To generate a large number of resource graphs for simulations, we first construct a small number of seed graphs based on real networks, and then generate larger graphs from those seed graphs by injecting new hosts and assigning resources in a random but realistic fashion (e.g., we vary the number of pre-conditions of each exploit within a small range since real world exploits usually have a small number of pre-conditions).

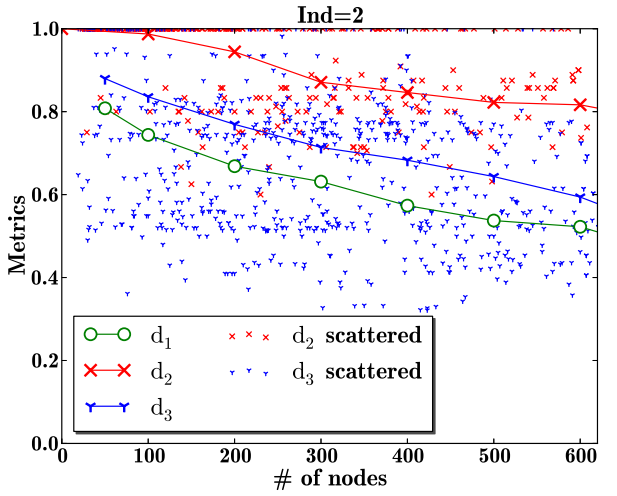
We apply the three network diversity metrics to different use cases, as presented in Section 3.2. Our objective is to evaluate the three metrics through numerical results and to examine those results together with statistically expected results represented by different attack scenarios.

The first two simulations compare the results of all three metrics to examine their different trends as graph sizes increase and as diversity increases. First of all, to convert the

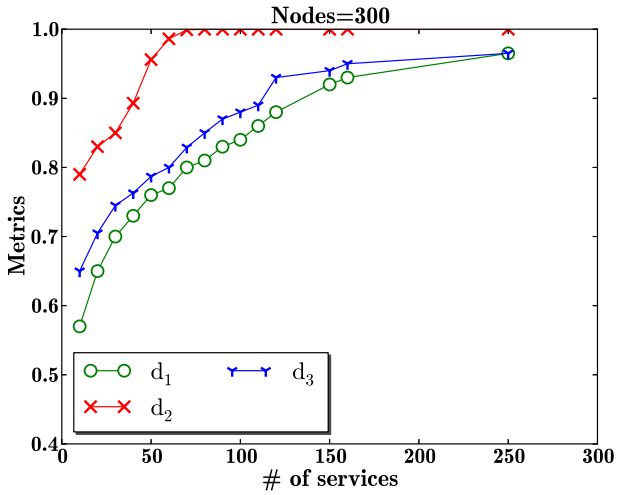
Bayesian network-based metric $d_3$ to a comparable scale of the other two, we use $\frac{\log_{0.08}(p')}{\log_{0.08}(p)}$ (i.e., the ratio based on equivalent numbers of zero day exploits) instead of $d_3$. In the left-hand side of Figure 8, the scatter points marked with $X$ in the red color are the individual values of $d_2$. The blue points marked with $Y$ are the values of $d_3$ (converted as above). Also shown are their average values, and the average value of the effective richness-based metric $d_1$. The right figure shows the average value of the three metrics in increasing number of distinct resources for resource graphs of a fixed size.

*Results and Implications:* Both simulations show that, while all three metrics follow a similar trend (in the left figure, diversity will decrease in larger graphs since there will be more duplicated resources) and capture the same effect of increasing diversity (in the right figure), the Bayesian network-based metric $d_3$ somehow reflects an intermediate result between the two other extremes ($d_1$ can be considered as the average over all resources, whereas $d_2$ only depends on the shortest path). Those results show that applying all three metrics may yield consistent results and motivates us to compare them through further simulations.

Next we examine the metric results under different use cases, as described in Section 3.2. The first use case considers worms characterized as follows. First, each worm can only exploit a small number of vulnerabilities. In our implementation, we randomly choose one to three resource types as the capability of each worm. Second, the goal of a worm is infecting as many hosts as possible, does not need specific targets. Although some worms or bots may indeed in reality have a target, it is usually still necessary for them to first compromise a large number of machines before the target can be reached (e.g., Stuxnet [29]). In Figure 9, the $X$-axis is the ratio of the number of resource types to the number of resource instances, which roughly represents the level of diversity in terms of richness (it can be observed that $d_1$ is close to a straight line). $Y$-axis shows the results of the three metrics as well as the ratio of hosts that are not infected by the simulated worms. The four lines




(a)

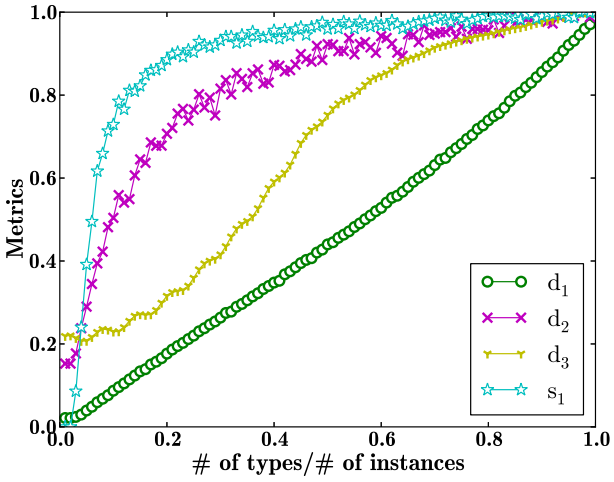



(b)

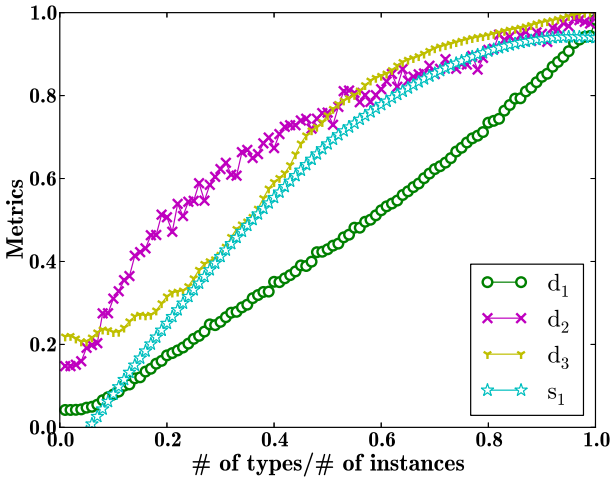
Figure 8: Comparison of Metrics (a) and the Effect of Increasing Diversity (b)

represent the three metrics (marked with $d_1$, $d_2$, and $d_3$) and the ratio of hosts uninfected by simulated worms (marked with $S_1$). Figure 8 (a) and (b) correspond to different percentage of first-level exploits (the exploits that only have initial conditions as their pre-conditions) among all exploits, which roughly depicts how well the network is safeguarded (e.g., 50% means a more vulnerable network than 10% since initially attackers can reach half, or five times more, exploits). For each configuration, we repeat 500 times to obtain the average

result of simulated worms.




(a)




(b)

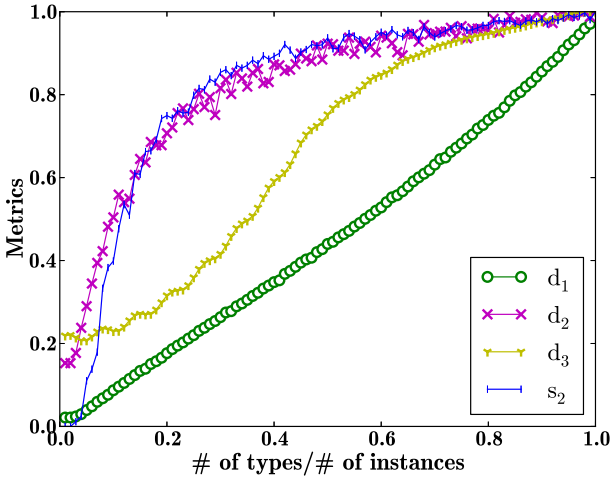
Figure 9: Worm Propagation (10% Initially Satisfied Exploits (a), 50% Initially Satisfied Exploits (b))

*Results and Implications:* In this simulation, we can make the following observations. First of all, all three metrics still exhibit similar trends and relationships as discussed above. The Figure 9 (a) shows that, when the network is better safeguarded (with only 10% of exploits initially reachable), the effect of increasing diversity on simulated worms shows a closer relationship with the $d_2$ metric than the other two, both of which indicate that

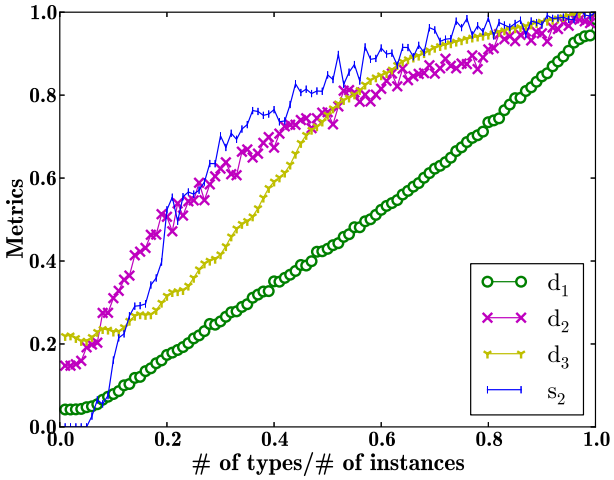
increasing diversity can significantly increase the percentage of hosts uninfected by worms. In comparison, the Figure 9 (b) shows a less promising result where both three metrics and the percentage of uninfected hosts all tend to follow a similar trend. Intuitively, in well guarded networks, many hosts cannot be reached until the worms have infected other adjacent hosts, so increasing diversity can more effectively mitigate worm propagation. In less guarded networks where half of the exploits may be reached initially, the effect of diversity on worms is almost proportional to the richness of resources ($d_1$), and all three metrics tend to yield similar results.

The second use case is targeted attacks (Section 3.2). We simulate attackers with different capabilities (sets of resources they can compromise) and the level of such capabilities (that is, the number of resources they can compromise) follows the Gamma distribution [72]. Similarly, we also repeat each experiment 500 times and we examine two different cases corresponding to different percentages of first-level exploits. In Figure 10, $S_2$ is the result of simulated attacker, which means the percentages of attackers who cannot reach the randomly selected goal condition.

*Results and Implications:* From the results we can observe similar results as with the simulated worms. Specifically, increasing diversity can more effectively mitigate the damage caused by simulated attackers for well guarded networks (the left figure) than for less guarded networks (the Figure 10 (a)). Also, in the left figure, the simulated attackers' results are closer to that of $d_2$ than the other two metrics, whereas it is closer to both $d_2$ and $d_3$ in the Figure 10 (b). In addition, by comparing the results in Figure 10 (targeted attack) to that in Figure 9 (worm), we can see that the same level of diversity can more effectively mitigate worm than it can do to simulated attackers. This can be explained by the fact that a worm is assumed to have much less capability (set of resources it can compromise) than a simulated attacker.



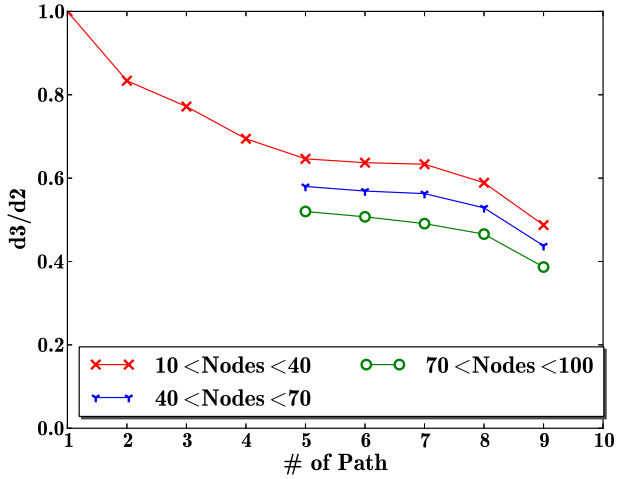

(a)



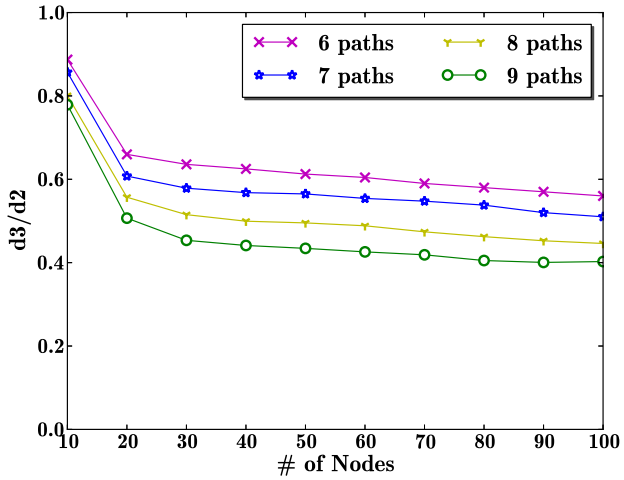

(b)

Figure 10: Targeted Attack (0% Initially Satisfied Vulnerabilities (a), 50% Initially Satisfied Vulnerabilities (b))

The next set of simulations examines the relationships between $d_2$ and $d_3$ in more details. Those two metrics both take into account the causal relationships between resources, but they focus on slightly different perspectives (the least and average efforts needed to compromise a network). At the same time, the two metrics are closely related. In previous simulations we already see that the values of $d_3$ are almost always smaller than $d_2$. The two metrics may also converge to similar values in certain cases (e.g., in an extreme case of




(a)




(b)

Figure 11: $d_3/d_2$ in the Number of Paths (a) and Nodes (b)

only one path in the resource graph, $d_2$ and $d_3$ will yield identical value). To see how those two metrics relate to each other, we simulate resource graphs of various sizes and shapes and the average results ($d_3/d_2$) of 500 simulations are shown in Figure 11 .

*Results and Implications:* Figure 11 (a) shows that, under fixed sizes of resource graphs, the difference between $d_2$ and $d_3$ increases (a ratio of 1 means they have identical values) with the number of paths in the resource graphs. This is expected since $d_2$ only represents

diversity in terms of one specific path while $d_3$ accounts all paths so with the number of paths increasing the ratios becomes smaller (meaning larger differences). The Figure 11 (b) shows that the differences increase (the ratio decreases) as graph sizes increase (all resource graphs have between 6 to 9 paths). This can be explained by the fact that the difference between different paths will become more significant as the size of graphs increases and with the number of paths fixed, and therefore the difference between the two metrics slowly increases (the ratio decreases). Both simulations imply that, although the choice of metrics mostly depends on the desired perspective, both metrics should be considered especially for larger and less well guarded networks (meaning more attack paths are present) since their values may be vastly different.

We now study the third use case, the Moving Target Defense (MTD). The MTD approach attempts to achieve better security by varying in time the configurations of networks, in which diversity plays a critical role. Our goal here is to study the effect of varying diversity on the effectiveness of MTD, and also on evaluating our metric when applied to MTD. To the best of our knowledge, this is among the first efforts on studying MTD using simulations (another similar effort by Zhuang et al. [141] also employs simulation results, but our goal is to evaluate the proposed network diversity metric under MTD applications, which is different from their study). Our simulations are based on following assumptions.

- We assume attackers do not initially have full details about the network but may gradually learn about such details as the configuration is changed over time. Specifically, attackers can learn about the change of a configuration (e.g., either through the failure of their attacks, or by observing special features of a configuration).

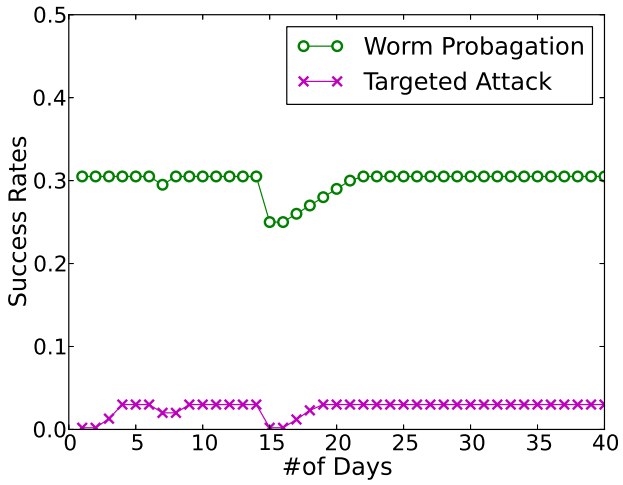
- We assume attackers' capabilities, which are sets of resources they can compromise, follow gamma distribution. And each resource has attack window, only those reach the duration of attack window may be compromised by attackers who have the capability. Moreover, having the capabilities does not mean the attacker can immediately

compromise the resource, since he/she may still need certain amount of time to actually implement the attack on specific instances of the resource. Therefore, in our simulations, we assign each resource an attack window, and only a resource whose duration of appearing inside a configuration is longer than the corresponding attack window may be compromised by attackers who have the capability.

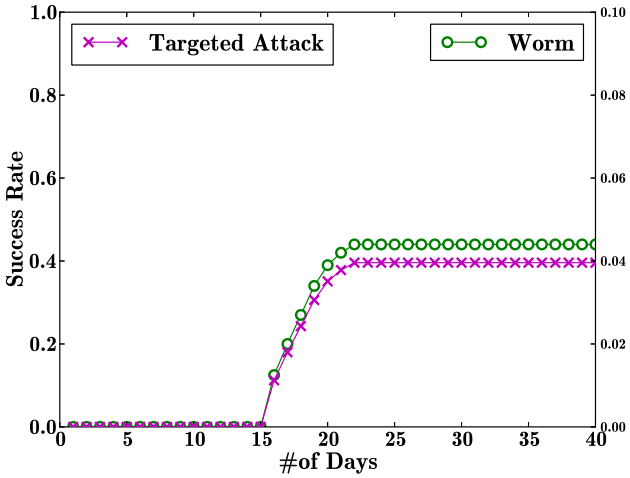
- We assume the MTD approach employs dynamically changing network configurations. But we use fixed frequency of changes in our simulations. Specifically, the network topology remains the same but resources of each host may change. Also, we assume a fixed frequency of changes in our simulations (e.g., in Figure 12, the left figure shows the frequency of configuration change is 1 configuration/per day). We leave other ways for changing network configurations, such as using a varying frequency, as future work.

Figure 12 (a) shows the average success rates of attackers after 40 days of exposure to the network in the number of days before a configuration changes (e.g., 5 day means there is one configuration change per five days). The Figure 12 (b) shows the attack success rates (the left $Y$-axis represents the success rate of worms and the right for targeted attacks) in the number of days since the network is exposed, with the frequency of changes set at one configuration change per day.

*Results and Implications:* From the Figure 12 (a), we can see that a more frequent change of network configurations does not necessarily equal to better security (lower success ratio), since too fast or too slow changes can both increase the exposure of a resource and hence increase an attacker's chance in compromising that resource. In fact, the left figure indicates no clear trend in the success rate as the number of days for a change increases. The small drops in both lines indicate that the lowest success rates coincident with the average size of attack windows (in this case we assume 10 or 15 days are required to compromise a resource), which may not be meaningful in reality since different resources



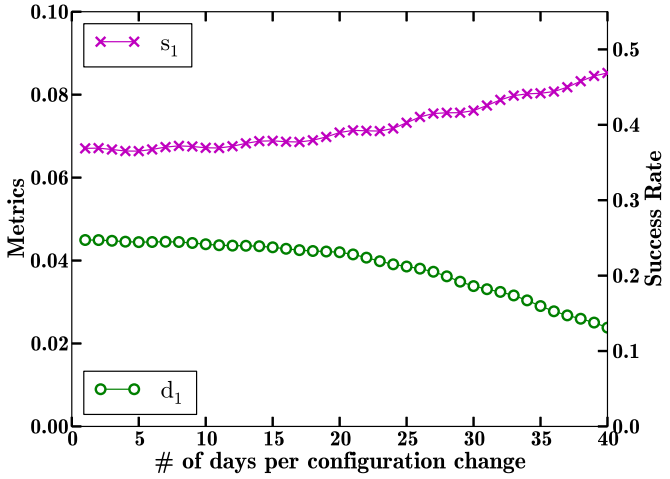

(a)



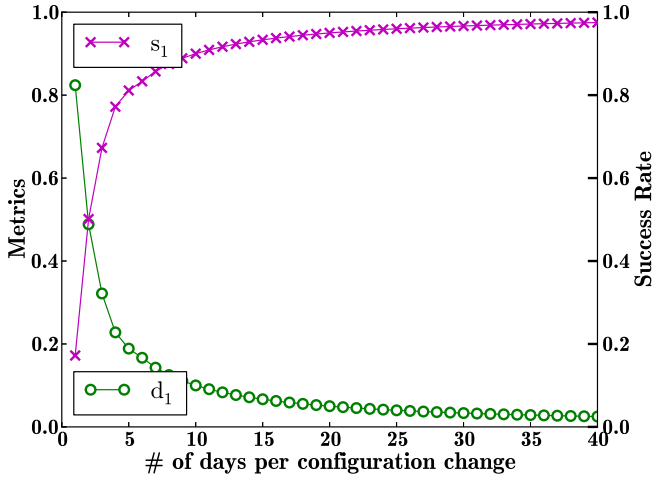

(b)

Figure 12: Success Rate of Attacks in Frequency of Changes (a) and in Time (b)

may have different attack windows. From Figure 12 (b), we can see that, before 15 days, there is zero success rate for both worm propagation and targeted attack, due to the assumed 10 to 15 days of attack windows. After 15 days, there is a jump in both lines, which means, with accumulated efforts, both worms and attackers will be able to compromise more and more resources, and the success rates do not change much after about 20 days since now they will depend more on the capability of worms (attackers).




(a)




(b)

Figure 13: $d_1$ in Frequency of Changes, under Less Resource Types (Left) and More Resource Types (Right)

In Figure 13, we only apply the $d_1$ metric to MTD since all three metrics will show similar trends as discussed above. In the Figure 13 (a), we can see that if the set of resources types remains at a relatively small size (e.g., $\frac{\#ofResources}{\#ofDays} < 20\%$), then our $d_1$ metric stays almost flat when the frequency of configuration changes is relatively high, and it then drops more dramatically when the frequency is lower (around one change per 20 days). This indicates that, with limited number of resource types, a higher frequency of configuration

changes does not provide much security gain, unless if the frequency is too low. Figure 13 (a) also indicates that, when the number of days per change increases over 20, diversity drops while success rate increases, which means our diversity metric effectively capture the effectiveness of MTD. Figure 13 (b) shows similar results under larger set of resources ($\frac{\#ofResources}{\#ofDays} > 80\%$). The successful rate and $d_1$ shows corresponding (reversed) trends. However, now with a large enough set of resources to choose from, less frequent changes in configurations mean lower diversity and hence less security.

In this last simulation, we study the relationship between cost and security in MTD, as shown in Figure 14. For worm propagation, we assign monetary values to all hosts, with critical assets (goal conditions) having higher values, whereas for targeted attack, we only assign a value to critical assets. The red and green dashed lines on top of the figure shows the total value for each scenario. Each time when worms or attackers compromise a resource, its assigned value is considered lost. Such lost value is the first part of overall cost. The other part is the cost of changing configurations in MTD (e.g., administrative cost of purchasing new software or performance cost of delaying a client's request).

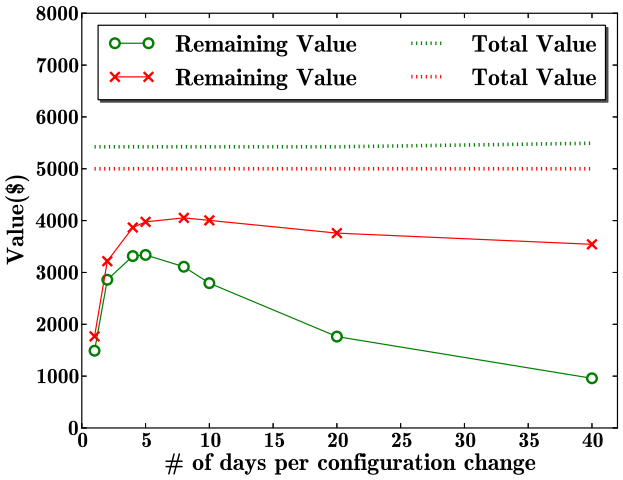



Figure 14: Total Cost in Frequency of Changes

*Results and Implications:* Figure 14 depicts the total cost (lost value due to compromised resources plus cost of configuration changes) in the number of days for a change of configuration. The results show that the overall cost will first decrease and then increase. The optimal setting of frequency is about one configuration change per 5 days, which best balances the cost of changing new configuration with lost values of compromised resources. Note that, according to our discussions above, the optimal frequency will also depend on the number of available resources.

## 3.8 Discussion

The similarity between resources is an important input to our metric models. In Section 3.6.2, we have taken a simplistic approach of regarding resources as either identical or completely different (the similarity score is either 1 or 0). Although such an approach may be acceptable in many cases, it certainly needs to be refined considering the fact that slight differences usually exist among different versions of the same software, whereas different software may share common code or libraries. Measuring such differences or similarity between resources will lead to more accurate results for the diversity metrics. This section demonstrates such a need and discusses potential solutions.

### 3.8.1 A Case Study

To demonstrate the need for studying software similarity, we conduct a case study on different versions of Chrome, and show how such differences when taken into account may affect the diversity metric results. Specifically, we examine 11 consecutive versions of the Chrome browser between versions 42.0.2283.5 (published Jan 22 06:24:31 2015) and 41.0.2272.16 (published Jan 21 02:44:41 2015) all of which are published along two development branches 41 and 42. The versions are labeled with index numbers between 0 to

10, from the latest to the oldest, in Figure 15. We use the latest version 0 (42.0.2283.5) as the baseline for comparison.

It might be expected that not much difference should exist between those versions with such small differences in version numbers and publish time. However, our study shows this is not really the case. Taking versions 0 (42.0.2283.5) and 1 (41.0.2272.28) as an example, while the total number of source files is quite similar (75136 and 73596, respectively), there are differences in totally 9338 files (about 12%) between the two versions. Moreover, such differences include 767,987 insertions and 190,943 deletions, which accounts for about 13% of lines (the total number of lines in those two versions are 14,750,264 and 15,330,677, respectively). Those numbers clearly indicate a significant difference between the two versions even though their publish time are only a few minutes apart.

In Figure 15, the two lines depict the number of differences in terms of files and modifications, respectively, for the ten versions. Clearly, there is a similar trend at those two different levels of differences, with smaller differences among versions in the same branch (e.g., 42) and more significant differences between different branches (the numbers can reach almost 10,000 at file level and 1,000,000 at modification level). Although the version numbers may seem to provide useful information in this particular case, such information about the relationships between multiple versions (e.g., development branches) may not always be provided by a software vendor. Therefore, it is not a reliable approach to rely on either the version numbers or publish time to determine the similarity between multiple versions.

### 3.8.2 Potential Solutions

In addition to slight differences between multiple versions of the same software, there may also exist similarity between completely different software. Most of today's complex

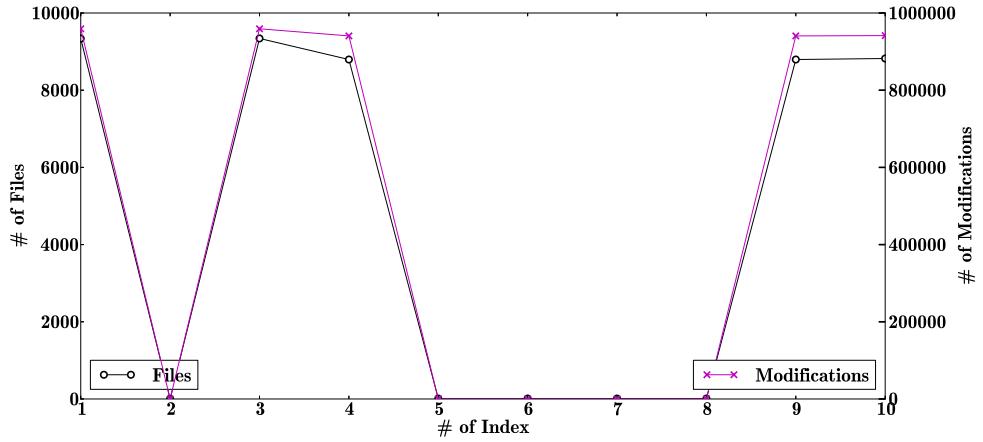



Figure 15: Differences at File and Modification Levels between Different Versions of Chrome

software are developed under established software engineering principles which encourage modularity, software reuse, the use of program generators, iterative development, and open-source packages. As a result, many software may share common code blocks or employ common libraries, both of which may result in significant similarity between those software. For example, Chrome and Opera have both been using Blink as their rendering engine since 2013 so both include common code blocks. A well known example of sharing common library functions in different software is the recent Heartbleed bug in which many popular Web servers, including Apache HTTP Server, IBM HTTP Server, and Nginx, all employ the common openssl library to establish SSL connections. To measure such similarity between different software, we believe existing efforts, such as clone detection at both source code and binary levels, should be leveraged and extended to develop practical solutions. Although this is not the main focus of this chapter, we provide a brief overview of existing approaches which we believe are promising in this regard.

At source code level, there exists a rich literature on clone detection, which attempts to match code fragments through both syntax and semantic features. For example, text-based approach extracts signatures of the lines and then match software based on substrings [53].

Such a technique provides basic matching results although it has many limitations, e.g., it cannot handle identifier renaming, since it does not transform source code into intermediate formats. Token-based approach parses the source code into a list of tokens [10] so that it can handle renaming, although it has its own limitations, e.g., it cannot deal with replacement of control flows. In a tree-based approach [28], an abstract syntax tree is generated from the source code for matching. Such technique provides more semantic features but it ignores data flows and therefore cannot deal with reordering statements. Apart from those matching-based approaches, a similarity distance-based approach [15] calculates a distance between two code segments and then compare to a given threshold. There exist many other approaches in this literature, all of which may be leveraged to identify and quantify similarity between open source software.

As to closed source software, identifying shared code or library functions is more challenging. Nonetheless, there are many existing efforts on assembly-level clone detection and library identification. The text-based approach regards the executable part of a binary as a sequence of bytes or lines of assembly and compares them to find identical sequences [50]. The token-based approach relies on feature vectors consisted of opcodes and operands and employs metrics to provide function-level clone detection [106]. The structural-based approach maps the code back to execution schema and compares their structural features [26]. Our recent work combines several existing concepts from classic program analysis, including control flow graph, register flow graph, and function call graph, to capture semantic similarity between two binaries [5]. Finally, the binary search engine provide an easy way for locating shared libraries inside a software binary [57]. Although more challenging than it is for open source software, We believe developing practical tools by leveraging such existing efforts to identify and estimate similarity for closed source software is still possible.

Finally, variations of software may also be caused by configuration differences (e.g., different settings of Sophos antivirus software), additional security hardening measures

(e.g., SELinux and Grsecurity), add-ons and plugins, etc., which may sometimes offer even more substantial impact than different versions of a software. Taking into account such factors in measuring software similarity can be a real challenge and the only tangible solution may still be relying on administrators' manual inspection and estimation.

## 3.9 Conclusion

In this topic, we have taken a first step towards formally modeling network diversity as a security metric for evaluating networks' robustness against zero day attacks. We first devised an effective richness-based metric based on the counterpart in ecology. We then proposed a least attacking effort-based metric to address causal relationships between resources and a probabilistic metric to reflect the average attacking effort. We provided guidelines for instantiating the proposed metrics and discussed how software diversity may be estimated. Finally, we evaluated our algorithms and metrics through simulations. Our study has shown that an intuitive notion of diversity could easily cause misleading results, and the proposed formal models provided better understanding of the effect of diversity on network security.

# Chapter 4

# *Network Attack Surface*: Lifting the Attack Surface Concept to Network Level for Evaluating the Resilience Against Zero-Day Attacks

In this chapter, we describe our efforts on the second proposed research topic, i.e., network attack surface metric.

## 4.1 Introduction

For a mission critical computer network (e.g., those that play the role of a nerve system in critical infrastructures, governmental or military organizations, and enterprises), the security administrators usually look beyond traditional security mechanisms, such as firewalls and IDSs. Their worry over the prospect of Advanced Persistent Threat (APT) and hidden malware usually drive them to understand the resilience of their networks against potential zero day attacks exploiting previously unknown vulnerabilities. However, while there

exist many standards and metrics for measuring the relative severity of known vulnerabilities (e.g., CVSS [73]), the task becomes far more challenging for unknown vulnerabilities, which are sometimes believed to be unmeasurable [70].

To that end, a promising solution is the *attack surface* concept [67], which is originally proposed for measuring a software's degree of security exposure along three dimensions, namely, entry and exit points (i.e., methods calling I/O functions), channels (e.g., TCP and UDP), and untrusted data items (e.g., registry entries or configuration files). Since attack surface relies on such intrinsic properties of a software independent of external factors, such as the disclosure of vulnerabilities or availability of exploits, it naturally covers both known and unknown vulnerabilities [67] and becomes a good candidate for understanding the threat of zero day attacks.

Evidently, in addition to software security, the concept of attack surface has also seen many applications in other emerging domains, e.g., cloud security [36], mobile device security [54], automotive security [17], Moving Target Defense (MTD) [49]. However, in contrast to the original attack surface metric, which is formally and quantitatively defined for a single software, most of the applications at higher abstraction levels (e.g., the network level) are limited to an intuitive and qualitative notion. Adopting such an imprecise notion unavoidably loses most of the original concept's power in formally and quantitatively reasoning about the likelihood of a system to contain vulnerabilities.

In this chapter, we address this issue by lifting the original attack surface concept to the network level as a formally defined security metric, namely, *network attack surface*, for evaluating the resilience of networks against potential zero day attacks. There are two main challenges in lifting attack surface to the network level. First, the attack surface model relies on addition for aggregating scores, which is incompatible with the causal relationships among different resources inside a network. Second, there exists a paradox that the only way to avoid the costly calculation of attack surface is to perform that calculation. We

devise models and heuristic algorithms to address those challenges, and we confirm the effectiveness of the proposed solutions through experiments (e.g., our algorithms has an error rate of 0.05 with only 20% of the resources calculated).

The main contribution of this work is twofold. First, to the best of our knowledge, this is the first effort on lifting the attack surface concept to the network level as a formally defined security metric. We believe such a metric may serve as the foundation of many useful analyses for quantitatively designing, evaluating, and improving network security. Second, our simulation results show that the proposed algorithms can produce relatively accurate results with a significant reduction in the costly calculation of attack surface, paving the way for practical applications.

**Mot**

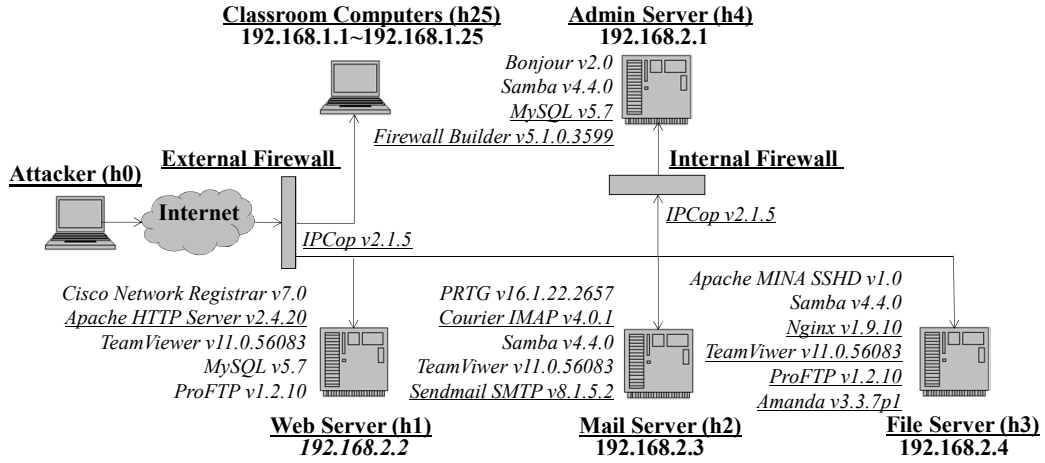



Figure 16: The Motivating Example

First, we illustrate the main challenges through a motivating example shown in Figure 16 ( the network topology depicts a fictitious campus network [98]). We assume the *External Firewall* allows all outbound connection requests but blocks all inbound requests to the *Mail Server* (h2) and *File Server* (h3), including those from the *Classroom Computers* (h25); the *Internal Firewall* allows all outbound requests from h4 but blocks all inbound requests except those from h2. We also assume our main concern is protecting the *Admin*

*Host* (h4) containing critical assets. Based on such assumptions, we can easily see that, an attacker at h0 can potentially follow an *attack path*, e.g., $h1 \rightarrow h2 \rightarrow h4$, to compromise h4. Keeping this in mind, we now consider the question: *How could we apply the attack surface concept, which is only defined for each individual resource [67] to such a network to measure its security (e.g., in terms of h4)*?

Two obvious solutions are to directly apply the metric either by regarding the whole network as a single software system, or by first applying it to each resource separately, and then adding the results together. Since the addition operation is associative, both solutions yield the same result, i.e., the total numbers of methods, channels, and untrusted data items, respectively (more details are given in Section 4.2). The main problem here is that such an addition operation is incompatible with the causal relationships between network resources, which can be either conjunctive or disjunctive. For example, in Figure 1, while it makes sense to add up the attack surface of all the Classroom Computers (i.e., a larger number of such computers means the network is more exposed to attacks). Applying this along an attack path, e.g., $h1 \rightarrow h2 \rightarrow h4$, is less meaningful. Because it means a longer attack path would yield a larger attack surface (less secure), but a longer path usually requires more effort from attackers (more secure), which is a contradiction. Therefore, our first challenge is *how to aggregate the attack surface of network resources while respecting their causal relationships*, which will be the main topic of Section 4.2.

The second major challenge lies in the calculation of attack surface, which is well known to be costly since identifying the source code that lies on the attack surface requires domain expertise and significant manual effort [67, 120]. Therefore, a natural question is whether we can reduce our effort by avoiding calculating attack surface for those resources that do not contribute to the final result. For example, in Figure 1, since our main concern is h4, we only need to calculate attack surface for the path $h1 \rightarrow h2 \rightarrow h4$, which significantly saves the effort by avoiding the calculation for the 25 *Classroom Computers*. However, the

problem is not so straightforward in general. In the above example, suppose we change the firewall rules such that requests from both h2 and h3 to h4 are allowed. We now have a paradox that, in order to know which path, $h1 \rightarrow h2 \rightarrow h4$ or $h1 \rightarrow h3 \rightarrow h4$, should be calculated (the criteria for selecting the path will be detailed in Section 4.2) such that we can avoid calculating the other path, we must first calculate and compare the attack surface of both h2 and h3, which defies the purpose because by then we would have calculated both attack paths. Therefore, our second challenge is *how to reduce the effort of calculating attack surface for network resources while keeping the final result sufficiently accurate*, which will be the main topic of Section 4.3.

## 4.2 The Network Attack Surface Model

In this section, we lift the attack surface concept to the network level in two steps. First, Section 4.2.1 and Section 4.2.2 convert the attack surface of a software application to its attack probability. Specifically, Section 4.2.1 obtains the attack probability based on the mapping between attack surface and the common vulnerability scoring system (CVSS), while Section 4.2.2 captures the relationships among resources and converts them into an overall attack probability. Second, Section 4.2.3 aggregates the attack probabilities of different network resources into a single measure of network attack surface.

### 4.2.1 CVSS-Based Attack Probability

This section addresses the challenge that the addition operation used in attack surface is incompatible with the causal relationships between network resources, as demonstrated in Section 4.1. Our main idea is to convert the attack surface of each software resource into an *attack probability* (the relative likelihood that the software contains at least one exploitable zero day vulnerability), which can then be aggregated for different resources

based on their causal relatinships [1]. Since attack surface provides an indication of both the severity (represented by the weights, i.e., the access rights and privileges) and the likelihood (represented by the counts, i.e., the total numbers of methods, channels, and untrusted data items) of potential vulnerabilities [67], the conversion will take two steps as follows.

- First, for each group of methods, we explore a mapping between the attack surface and the common vulnerability scoring system (CVSS) [73] to convert the access rights and privileges of attack surface to a CVSS base score.
- Second, at the software level, we aggregate the base scores of different groups of methods into a single attack probability for the entire software.

**Method Group-Level Conversion** First, we briefly review the concepts of attack surface and CVSS. As illustrated in the first column of Table 3, the CVSS defines six base metrics in two groups, the accessibility group including access vector (AV), access complexity (AC), and authentication (Au), and the impact group including confidentiality impact (C), integrity impact (I), and availability impact (A) (the possible values of each metric and their corresponding numerical scores are also shown in the table) [73]. The second column of Table 3 shows the different access rights and privileges and their numerical values used as weights in the attack surface metric (the underlined rows will be discussed later).

Since both the accessibility group of CVSS and the access rights of attack surface represent the pre-conditions for exploiting a vulnerability, their values may be mapped together. Similarly, the impact group of CVSS and the privileges of attack surface both represent the post-conditions of exploiting a vulnerability, and hence are mapped together. The exact mapping for those two IMAP daemons are shown in the last column of Table 3. Each CVSS vector maps to the corresponding access right or privilege shown in the same row in

[1] Note the attack probability here is only intended as a relative metric for comparison between different sofware applications, instead of the actual probability of attacks which is generally infeasible to obtain in practice.

| CVSS (Base Metric Group) | Attack Surface (Methods) | | | Vectors |
|---|---|---|---|---|
| AV:[L:0.395,A:0.646,N:1.0] | Access Rights | anoymous | 1 | [AV:N,AC:L,Au:N] |
| AC:[H:0.35,M:0.61,L:0.71] | | unauthenticated | 1 | [AV:N,AC:L,Au:N] |
| Au:[M:0.45,S:0.56,N:0.704] | | authenticated | 3 | [AV:N,AC:M,Au:S] |
| | | admin | 4 | [AV:A,AC:H,Au:M] |
| C:[N:0.0,P:0.275,C:0.66] | Privileges | authenticated | 3 | [C:P,I:P,A:C] |
| I:[N:0.0,P:0.275,C:0.66] | | cyrus | 4 | [C:C,I:C,A:C] |
| A:[N:0.0,P:0.275,C:0.66] | | root | 5 | [C:C,I:C,A:C] |

Table 3: Mapping Attack Surface to CVSS Base Metrics for Courier IMAP Server v4.1.0 and Cryus IMAP Server v2.2.10

the second column.

The mapping is established based on understanding the software, including its channels and untrusted data items (consequently, we will not count those again later when we convert base scores into attack probabilities). For example, in the third row, the authenticated access right is mapped to network for access vector (i.e., *AV:N*), because the UNIX socket in those software has the local authenticated access right, which means attackers may obtain the local authenticated access right over the network. Also, we assign access complexity to medium (i.e., *AC:M*), because the authenticated access right matches the description of the medium access complexity: "The affected configuration is non-default, and is not commonly configured (e.g., a vulnerability present when a server performs user account authentication via a specific scheme, but not present for another authentication scheme)" [73]. Finally, we assign Authentication to single (i.e., *Au:S*), because the access requires a single authenticated session in those software. Similarly, in the fifth row, the authenticated privilege is mapped to partial confidentiality impact, partial integrity impact, and complete availability impact (i.e., *C:P, I:P, A:C*), since the authenticated privilege implies accesses to 13 files in those software, allows modifying some system files or data, and may render the system unusable by deleting critical files.

Note that, since this mapping is based on the understanding of access rights, privileges, and the software, different administrators may end up assigning the mappings in different

and incomparable ways. However, since metrics are relative, and meant for comparing different configurations of the same network, the results would still be meaningful as long as the mapping is consistent across different configurations.

As shown in Table 3, we map all the methods of those two software to corresponding CVSS base metrics, and then calculate the overall base score according to the CVSS formula [73], as shown in Table 4. The methods are divided into groups (first column) according to similar privileges (second column) and access rights (third column). The fourth and fifth columns show the total numbers of entry and exit points in each group. The next two columns show the mapped CVSS vector and the calculated base score for each group.

| Method | Privilege | Access Rights | DEP | DExp | Vector | Base Score | Attack Probability |
|---|---|---|---|---|---|---|---|
| Courier | | | | | | | |
| M1 | root | unauthenticated | 28 | 17 | [AV:1.0,AC:0.71,Au:0.704,C:0.66,I:0.66,A:0.66] | 10 | 0.000315 |
| M2 | root | authenticated | 21 | 10 | [AV:1.0,AC:0.61,Au:0.56,C:0.66,I:0.66,A:0.66] | 8.5 | 0.000184 |
| M3 | authenticated | authenticated | 113 | 28 | [AV:1.0,AC:0.61,Au:0.56,C:0.275,I:0.275,A:0.66] | 7.5 | 0.000809 |
| Cyrus | | | | | | | |
| M1 | cyrus | unauthenticated | 16 | 17 | [AV:1.0,AC:0.71,Au:0.704,C:0.66,I:0.66,A:0.66] | 10 | 0.000132 |
| M2 | cyrus | authenticated | 12 | 21 | [AV:1.0,AC:0.61,Au:0.56,C:0.66,I:0.66,A:0.66] | 8.5 | 0.000112 |
| M3 | cyrus | admin | 13 | 22 | [AV:0.646,AC:0.35,Au:0.45,C:0.66,I:0.66,A:0.66] | 6.3 | 0.0000882 |
| M4 | cyrus | anonymous | 12 | 21 | [AV:1.0,AC:0.71,Au:0.704,C:0.66,I:0.66,A:0.66] | 10 | 0.000132 |

Table 4: Method Groups and Their Base Scores for Courier IMAP Server v4.1.0 and Cyrus IMAP Server v2.2.10

**Software-Level Conversion** Now that we have calculated the base score for each group of methods, we can convert the attack surface into an *attack probability* representing the relative likelihood of the software to be exploitable through at least one zero day vulnerability. Suppose there are totally $g$ groups of methods in the attack surface. Let $b_i$ and $s_i$ $(1 \leq i \leq g)$ denote the base score and the number of methods of each group, respectively. Assume on average there will exist one zero day vulnerability for every $n$ methods, and the probability for attackers to discover such a vulnerability is $p_0$ ($n$ and $p_0$ are both intended as normalizing constants; see below for more discussions). In Equation 1, the base score divided by its range 10 gives the probability that a vulnerability in this group is exploitable; multiplying this with $p_0$ gives the probability that the method can be both discovered and exploited; $s_i/n$ gives the number of vulnerabilities out of those $s_i$ methods in this group; the

equation therefore gives the probability $p$ that the software contains at least one exploitable zero day vulnerability.

$$p = 1 - \prod_{i=1}^{g} (1 - p_0 \frac{b_i}{10})^{\frac{s_i}{n}} \quad (1)$$

**Example 1.** *Assuming* $n = 30$ *and* $p_0 = 0.08$*, we can calculate p for both software as follows. For Courier,* $p = 1 - (1 - 0.08 * 10/10)^{45/30} * (1 - 0.08 * 8.5/10)^{31/30} * (1 - 0.08 * 7.5/10)^{141/30} = 0.384$*, and similarly for Cyrus,* $p = 0.273$.

Note that, the true values of parameters $n$ and $p_0$ are certainly impossible to obtain in practice, so those are only intended to be normalizing constants chosen to ensure a reasonable value for $p$. As long as those values stay constant between different software, the equation will yield a relative value sufficient for comparing the exploitability of different software based on both the severity (represented by the base scores $b_i$) and counts (represented by the number of methods $s_i$) of potential zero day vulnerabilities.

### 4.2.2 Graph-Based Attack Probability

In Section 4.2.1, the overall attack probability obtained using Formula 1 does not capture the relationships among different dimensions of attack surface, e.g. channels and untrusted data items are considered only as indirect inputs in the mapping process, which maps to the accessibility and impact of methods, respectively. In this section, we capture the relationships among different resources through a model that presents possible attacks among resources and then aggregate the overall attack probability with respect to a critical condition or asset, molded as a goal condition.

We combine different dimensions of attack surface by taking attackers' point of view where attack would typically require communication channels (the channel dimension) to access the methods and invoke methods to manipulate untrusted data items to fulfill their

goal. If a software application is isolated from the network and has no channels associated with it, attackers would not be able to launch attacks on the software regardless of the number of methods and untrusted data items. Likewise if the software application does not have any method, it would be impossible for attackers to access untrusted data items. This observation shows that a combination of thethree dimensions is usually needed to complete an attack. The conversion of the attack surface to graph-based attack probability requires two steps as follows.

- First, for each group of resources in the three dimensions of attack surface, we calculate the probability for the entire group of resources based on Equation 2.
- Second, at the software level, we aggregate the probabilities over each group into a single attack likelihood based on Bayesian inferences as follows.

**Method Group-level Conversion** First, we divide methods into groups based on the pair $\langle access\ right, privilege \rangle$, such that the methods in the same group require identical access right and lead to identical privilege. The first column of Table 4 gives the group name for each method group and we will simply use M1 in Courier to refer to the group of methods restricted by unauthenticated access right and lead to root privilege in Courier in the latter sections. In group M1 of Courier, attackers only need to exploit one method out of 45 to gain the corresponding privilege; however, without having the knowledge about the methods in a software application, attackers may exploit multiple methods in one group. Taking this into consideration, we define the attack likelihood of one group of methods as the probability of compromising at least one method out of the group. Suppose we have totally $s_i$ methods in one group, and let $b$ and $p_0$ denote the base score and the probability for attackers to discover one method, respectively. In Equation 2, the base score divided by its range 10 gives the probability of finding a method in a software application to be exploitable; multiplying this with $p_0$ gives the probability that the method can be

both discovered and exploited. As explained earlier, $p_0$ is only intended to be normalizing constants chosen to ensure a reasonable value for p in Section 4.2.1.

$$p = 1 - (1 - p_0 * \frac{b}{10})^{s_i} \tag{2}$$

**Example 2.** *To compare Courier and Cyrus, we take $p_0$ as the ratio of choosing one method per thousand line of the source code. The number of lines of source code for Courier and Cyrus are 138,283 and 236,321, respectively [84]. Therefore, it is easy to get $p_0 = 0.00723$ for Courier and $p_0 = 0.00423$ for Cyrus. We can calculate p for M2 for both software applications as follows. For Courier, $p = 1 - (1 - 0.00723 * \frac{8.5}{10})^{31} = 0.174$, and similarly M2 in Cyrus, $p = 0.112$*

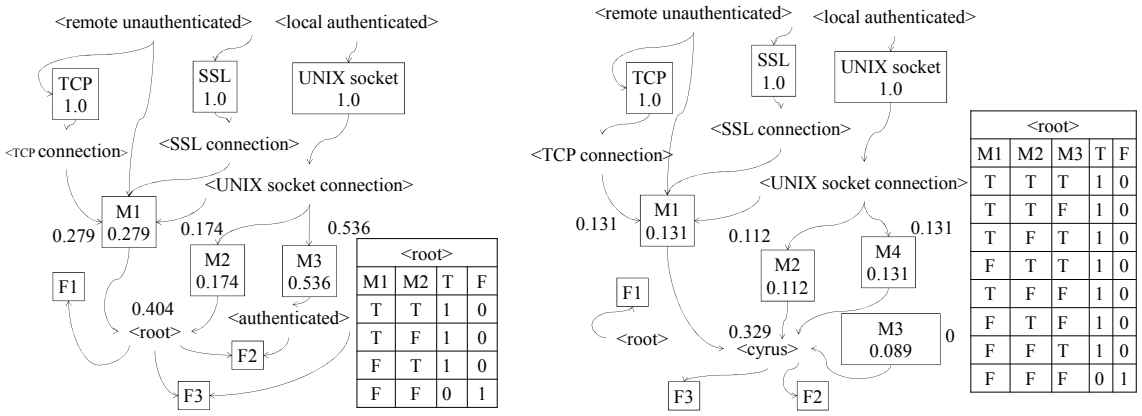



Figure 17: Attack Surface Graph for Courier (Left) and Cryus (Right)

**Software-Level Conversion** In order to model *attack probability* for a software application based on attack surface, we first need a model of the relationships among the three dimensions of attack surface. Our model is syntactically equivalent to an *attack graph* [110], [6], although our model focuses on resources inside software applications instead of known vulnerabilities.

**Definition 7** (Attack Surface Graph)**.** *Given a set of attack surface resources $R_A$ from*

$\langle M, C, D \rangle$ , *and a set of privileges Con, a* required *relation* $R_r \subseteq Con \times R_A$ , *and an* imply *relation* $R_i \subseteq R_A \times Con$. *An attack surface graph* $G_A$ *is the directed graph* $G_A(R_A \cup Con, R_r \cup R_i)$ *(*$R_A \cup Con$ *is the vertex set and* $R_r \cup R_i$ *is the edge set). resource.*$R_r$ *and resource.Ri denote as the relations request and imply by this resource, respectively.*

- *Resource C can be accessed by attackers only when* $C.R_r$ *are obtained by attackers*
- *Resource M can be invoked by C only when* $C.R_i \succ M.R_r$
- *Resource D can be accessed by M only when* $M.R_i \succ D.R_r$

Table 5: IMAP Daemon's Channels and Untrusted Data items [67]

| Courier Channels | | Untrusted Data items | | |
|---|---|---|---|---|
| Type | Access Rights | Group | Type | Access Rights |
| TCP | remote unauthenticated | F1 | file | root |
| SSL | remote unauthenticated | F2 | file | authenticated |
| UNIX socket | local authenticated | F3 | file | world |
| Cyrus Channels | | Untrusted Data Items | | |
| TCP | remote unauthenticated | F1 | file | root |
| SSL | remote unauthenticated | F2 | file | cyrus |
| UNIX socket | local authenticated | F3 | file | cyrus |

Figure 17 depicts corresponding *attack surface graph* for both Courier and Cryus (note the figure may look similar to an attack graph but here it indicates the aggregation of the three dimensions inside attack surface) Each square box in Figure 17 represents a resource in attack surface (e.g., TCP, SSL, and UNIX socket, which are channels in attack surface, are represented as the connectivity for the software applications); the edges point from the pre-conditions to the resources (e.g., $\langle TCP\ connection \rangle$ and $\langle remote\ unauthenticated \rangle$ to M1) or from the resources to the post-conditions (e.g., M1 to $\langle root \rangle$).

Specifically, channels, which are modeled as the resources connected by initial conditions in the attack surface graph, can be directly accessed by attackers since initial conditions are assumed to be already satisfied. Methods can be invoked by attackers only if the

corresponding channels are associated with an equivalent or higher privilege. For example, we consider that attackers passing from the channel UNIX socket is able to access M1 (UNIX socket has *local authenticated* privilege which is higher than the required access right of M1, *unauthenticated*). Similarly, when sending untrusted date items, the privileges gained from methods should be equivalent or higher than the access right of untrusted data items. In Table 5, *root* is required to send data to F1 in Courier, which means M3 with *authenticated* does not have sufficient access right to send data to F1.

In an attack surface graph, since attack paths lead to critical resources or escalated privileges the attack likelihood of a software application can be represented as the average attack effort by combining all attack paths. For this purpose, we define the overall attack surface as the conditional probability for an attacker to compromise a given critical asset in a software application. The goal condition can be used to model any resources or conditions in the attack surface graph. In this paper, we assume the highest privilege that attackers are able to obtain as the goal condition.

With a goal condition given in an attack surface graph, the overall *attack probability* can be calculated using Bayesian inference to combine all the possible attack paths. For example, the overall *attack probability* for courier is 0.404 ( the conditional probability table is shown on the side) assuming all initial conditions are satisfied, e.g., remote authenticated and local authenticated. Similarly, the overall *attack probability* for Cryus is 0.329. Both attack probability methods (CVSS-Based and Graph-Based attack probability) show that courier has more potential attack likelihood compared with Cryus.

### 4.2.3 Aggregating Attack Probabilities inside a Network

Now that we have converted the attack surface of a resource to its attack probability, we can easily aggregate the attack surface of all network resources into a single *network attack surface* value. We consider two different ways for aggregating the attack surface of resources

in the network, the shortest path-based approach [126] and the Bayesian network (BN)-based approach [140], which reflect the worst case scenario (i.e., with respect to attackers following the easiest attack path) and the average case scenario, respectively.

To illustrate the idea, Figure 18 shows a partial *resource graph* [126] for our example, and the dashed line can be ignored for now and will be needed later (note that both the resource graph demonstrated here and the attack surface graph discussed in the previous section are syntactically equivalent to an attack graph, but the resource graph models zero day attacks at the network level, whereas the attack surface graph models known vulnerabilities at the software level). Specifically, each pair in plaintext is a security-related condition, e.g., connection $\langle source, destination \rangle$ or privilege $\langle privilege, host \rangle$, and each triple inside a box is a zero day exploit $\langle resource, source, destination \rangle$. The probability inside each box is the attack probability of the corresponding resource.

**Example 3.** *In Figure 18, for the shortest path-based approach, we can calculate the attack probability for the shortest path indicated by the dashed line,* $\langle IPCop, 0, F \rangle \rightarrow \langle Courier, 0, 2 \rangle \rightarrow \langle FirewallBuilder, 2, 4 \rangle$*, the probability can be calculated as* $p = 0.48 * 0.384 * 0.04 = 0.0074$ *(*attack probabilities *are obtained by the method in Section 4.2.1, and Section 4.4 provides more detailed calculation).*

**Example 4.** *For the BN-based approach, we can simply regard Figure 18 as a Bayesian network, with the attack probability of each resource regarded as the conditional probability that the corresponding resource can be exploited given that its pre-conditions are all satisfied, and then perform Bayesian inference to calculate the overall attack probability [140]. In this example, we can calculate the probability for attackers to reach* $\langle user, 4 \rangle$ *as* $p_{goal} = 0.016$.

From above examples, it is clear that our models of attack surface-based attack probabilities can help to address a key limitation of the existing $k$-zero day safety metric ( which

also adopts a shortest path-based approach) [126], i.e., it cannot discriminate different resources based on their relative attack probabilities. More formally, the following formally defines the concept of network attack surface.

**Definition 8** (Network Attack Surface)**.** *Given a network with the set of resources R, the attack probability $p(r)$ as defined in Equation 1 or Equation 2 for each $r \in R$, the resource graph G and a given condition $c_g \in G$,*

- *let AP denote the collection of all attack paths in G ending at $c_g$, and for each $ap \in AP$, let $R(ap)$ denote the set of resources involved in $ap$ and denote $p(ap) = \prod_{r \in R(ap)} p(r)$. We call $max(\{p(ap) : ap \in AP\})$ (where $max(.)$ returns the maximum value of a set) the worst case network attack surface w.r.t. $c_g$.*

- *let $B = (G', \theta)$ be a BN, where $G'$ is G annotated with the attack probabilities and $\theta$ is the set of parameters of the BN (the BN is more precisely defined in [140] and details are omitted here), and let $C_I$ be the set of conditions without parents in $G'$, we call $p = P(c_g \mid \forall_{c \in C_I} c = True)$ the average case network attack surface w.r.t. $c_g$.*

We note that, although the network attack surface above is defined as probabilities, those can potentially be converted into other forms for different interpretations. For example, given the network attack surface $p$ as a probability, we can easily convert $p$ into the equivalent number of methods $s$ with a given base score $b$, by inverting Equation 1 as: $s = n \log_{1-p_0}(1-p)$. We can therefore evaluate the network as a single software system with an attack surface composed of $s$ methods with the base score $b$ (which can also be mapped back to access rights and privileges if necessary). Also, we can convert $p$ back into an equivalent number of zero day vulnerabilities as $\log_{0.08} p$ (here 0.08 is a nominal probability for zero day vulnerabilities based on CVSS base metrics as described in [140]), which is a simple count-based metric helpful for interpretation and comparison purposes (we will use this method in our algorithms and simulations).

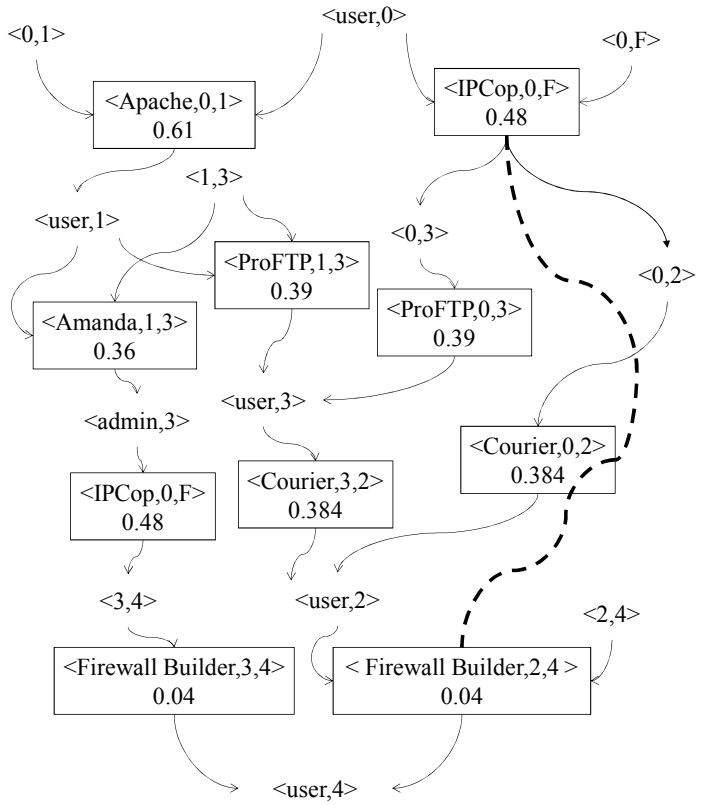



Figure 18: The Network Resource Graph with Attack Probability for the Network in Figure 1

## 4.3 Heuristic Algorithms for Calculating Network Attack Surface

In this section, we propose heuristic algorithms to reduce the effort in evaluating the network attack surface. We first state the problem in Section 4.3, and then introduce several simple heuristics in Section 4.3.1 and design algorithms based on such heuristics.

### The Problem statement

The calculation of attack surface is becoming more practical due to ongoing efforts on automating or approximating the calculation [120]. However, calculating the attack surface of a software can still be costly [67, 120] mostly due to the manual effort and expertise

**Procedure** *Mpath-Topo_Heuristic*
**Input:** Resource graph $G$, parameter $M$, and budget $N$
**Output:** a sequence of resources $P$
**Method:**
1. **Let** $P = \phi$ be a sequence of resources
2. **Let** $MS$ be the sequence of $M$ paths with the least numbers of exploits in $G$, with the paths sorted ascendingly based on such numbers, and the resources inside each path topologically sorted
3. **Let** $T = G \setminus MS$, topologically sorted
4. **While** $N > 0$
5. **If**$|\ MS\ | > 0$
6. **Append** the first resource $r$ in $MS$ to $P$
7. **Remove** $r$ from $MS$
8. **Else If** $|\ T\ | > 0$
9. **Append** the first resource $r$ in $T$ to $P$
10. **Remove** $r$ from $T$
11. **Let** N= N-1
12. **Return** $P$

**Procedure** *Keynode_Heuristic*
**Input:** Resource graph $G$, $p_0, p_1 \in [0,1]$, and budget $N$
**Output:** a sequence of resources $P$
**Method:**
1. **Let** $P = \phi$ be a sequence of resources
2. **Let** $KN = \phi$ be a sequence of resources
3. **Let** $p$ be the network attack surface calculated based on assigning $p_0$ to all the resources in $G$
4. **For** each resource $r$ in $G$
5. **Calculate** $p$ again on $G$ with $p_1$ assigned to $r$
6. **If** $p$ changes
7. **Add** $r$ to $KN$
8. **Sort** $KN$ based on topological order
9. **While** $N > 0$
10. **If** $|\ KN\ | > 0$
11. **Append** the first resource $r$ in $KN$ to $P$
12. **Remove** $r$ from $KN$
13. **Else If** $|\ G \setminus KN\ | > 0$
14. **Append** the first resource $r$ to $P$
15. **Remove** $r$ from $G$
16. **Let** N= N-1
17. **Return** $P$

Figure 19: Mpath-Topo (Left) and Keynode (Right) Heuristic Algorithms

required for analyzing the source code of the software in order to extract both the counts (e.g., the total number of methods calling I/O functions) and weights (e.g., the access rights and privileges). Moreover, even with automated techniques, the calculation will likely remain a costly process due to the ever increasing size of modern software [2].

Therefore, we investigate the problem of evaluating the network attack surface while reducing the effort of calculating the attack surface of individual resources. Clearly, there will be a tradeoff between the cost (i.e., the percentage of network resources whose attack surface is calculated, with the attack surface of the other resources estimated), and the error in the calculated network attack surface result. Specifically, given a network with the set of resources $R$ and suppose the true value of the network attack surface is $p_{true}$ and the calculated value is $p_{cal}$ (we assume all the values described in this section are count-based, as described at the end of Section 4.2.3), we would like to minimize the error $\frac{|p_{true} - p_{cal}|}{p_{true}}$ while calculating the attack surface for no more than a given percentage of resources (the budget).

[2]For example, the number of lines of software mentioned in our running example in Figure 1 are as follows: Nginx (171,567), IPCop (271,645), Apache(1,800,402), MySql (2,731,107), Linux Kernel (18,766,825), and Google Chrome (14,137,145).

Note that, although the above may seem to be a standard optimization problem, this is not really the case, because the objective function $\frac{|p_{true}-p_{cal}|}{p_{true}}$ contains an unknown value $p_{true}$ whose calculation would imply calculating the attack surface for all resources and defy the very purpose of reducing the cost. Also, since the problem of finding the shortest path is already NP-hard [126], which is a special case of our problem with unlimited budget, the latter is also intractable. Therefore, we study heuristic algorithms in the coming section.

### 4.3.1 The Heuristic Algorithms

The main observation is that, since we can only calculate a certain percentage of resources under a given budget, what determines the error is the order of calculation among all resources. Therefore, this section first considers a few straightforward heuristics for choosing the resources in the right order, e.g., by exploring the structural properties of a resource graph. We will then combine those heuristics into better algorithms in the coming section and evaluate their performance later in Section 4.5. We will focus on the worst case network attack surface, as given in Definition 8, while leaving the average case network attack surface to future work.

**Random Choose** The most obvious solution is probably to simply choose resources in a completely random fashion, namely, the *random choose* heuristic. Although the random choose algorithm is likely far from optimal, it provides a baseline for comparison with other heuristic algorithms we will propose. For example, in Figure 18, if our budget is to calculate the attack surface of at most two resources, then among the $\binom{6}{2} = 15$ possible choices, the worst result is $p = 0.46$ with an error rate of 0.76, whereas the best result is $p = 1.73$ with error rate 0.109. Clearly, this heuristic may lead to a solution that is far from optimal.

**Frequency Choose** The idea of this heuristic is that, since the same resource may appear on multiple hosts inside a network, calculating the attack surface for the most frequently seen resources will provide the most information with the same cost. For example, in Figure 18, we can see both *IPCop*, *Firewall Builder*, *Courier* and *ProFTP* appear twice among totally 10 exploits. Therefore, if our budget is two, then calculating any two of them will unveil $4/10$ of the exploits (the best result is $p = 1.73$ with an error rate of $0.109$, by choosing to calculate *Firewall Builder* and *Courier*. And the worst result is $p = 0.60$ with an error rate $0.69$, by choosing to calculate *IPCop* and *ProFTP*).

**Topological Order** The idea here is that, since the nodes closer to the first and last nodes of a resource graph (in the sense of a topological sorting) tend to be shared among more attack paths (e.g., the last two exploits are shared by all paths in Figure 18), it may help to choose resources based on a topological order among the exploits. We consider both the *topological order* and the *reversed topological order* heuristics, which choose resources in the same, and opposite order as topological sorting, respectively. For example, in Figure 18, suppose our budget is two, the topological order heuristic may choose *Apache* and *IPCop* (the result would be $p = 0.60$ with error rate $0.69$) while the reversed topological order may choose *Firewall Builder* and *Courier* (the result would be $p = 1.73$ with error rate $0.109$).

**Shortest Path** This heuristic starts the calculation with resources on the path with the least number of exploits (e.g., the path depicted in dashed line in Figure 18), which, although not always the right path in terms of the final result, may serve as a good starting point. For example, in Figure 18, if our budget is two, then the shortest path heuristic will choose *Courier* and *Firewall Builder* on the dashed line path (the result is $p = 1.73$ with error rate $0.109$). In this particular example, this path happens to be the right path for calculating the final result, so a larger budget will potentially produce more accurate result.

The above heuristics may not produce good results when each of them is used alone, but

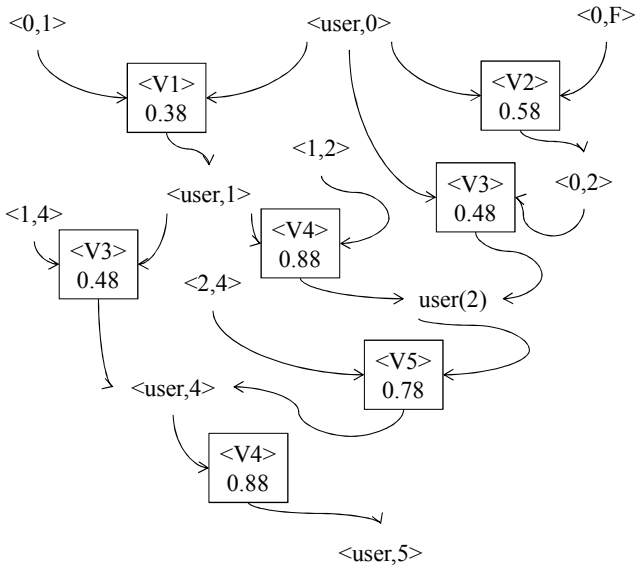



Figure 20: An Example of Applying Mpath-Topo and Keynode Heuristic Algorithms

combining them leads to algorithms with good performance, as will be confirmed through experiments in Section 4.5. The following presents two such algorithms.

**Mpath-Topo Heuristic Algorithm** This algorithm combines the above topological order and shortest path heuristics as follows. First, we apply the shortest path heuristic to choose $M$ (an integer parameter) shortest paths, which are ranked based on the number of unique exploits, as the starting points. Since there is no order between resources along each such path, we next apply the topological order heuristic to sort all paths, as well as those not on such paths. The algorithm is more clearly depicted on the left-hand side of Figure 19. Line 1, Line 2 and Line 3 define the sequence of resources P set, the resource set MS from least numbers of exploits in G of M paths (the least number of exploit paths can be found through the heuristic algorithm in [140]) and T the resource set regardless of the resources in MS with topologically order. The main loop cycle is from line 4 to line 11, which presents the resource choosing process. Constrained by the budget N, resources are first chosen from MS set and continue to be chosen from T set. The final resource sequence is presented in

P set.

**Example 5.** *In Figure 20, we have three paths with 5 distinct exploits, assuming* $M = 2$*, we have* $MS = \{V1, V4, V3, V5\}$*. If our budget* $N = 2$*, then* $P = \{V1, V4\}$*, and the final result is* $p = 0.51$*, with error rate* $0.04$

**Keynode Heuristic Algorithm** This heuristic algorithm is based on the idea that a resource is more important in determining the final network attack surface value $p$, if changing its value may result in significant changes, e.g., a change in the optimal path (the path selected for calculating the final result), or a change in the currently calculated result of $p$. We then combine this heuristic with the topological order heuristic to form the algorithm depicted on the right-hand side of Figure 19 (here we only show the change in $p$, which can be replaced with the change in the optimal path, and we will evaluate both variations in the coming section).

**Example 6.** *Here we choose* $p_0 = 0.08$ *and* $p_1 = 1$*. In Figure 20, we initially calculate* $p = 5.12 * 10^{-4}$*. We then calculate p again by assigning* $p_1$ *to each resource. For example, with* V1 *changed from* $p_0$ *to* $p_1$*, we have* $p = 0.0064$*, so* V1 *is a key node. Similarly, we can obtain the key nodes sequence as* $KN = \{V1, V4, V3, V5\}$*. If our budget* $N = 2$*, then* $V1$ *and* $V4$ *will be chosen and the result* $p = 0.51$ *with error rate* $0.04$*.*

If we apply all the heuristic algorithms to Figure 20, we can have the error rates with the corresponding algorithms: Frequency Choose (0.33), Topological Order (0.27), Reversed Topological Oder (0.33), Shortest Path (0.29), Mpath-Topo (0.04), Keynode (0.04). Clearly, the mpath-topo and keynode algorithms are significantly more accurate than other algorithms. We will evaluate the performance of those heuristics and algorithms, including both the accuracy and running time, in the coming section.

## 4.4 Instantiating the Network Attack Surface Metric

### 4.4.1 Case Study

To demonstrate how to apply the guidelines discussed in Section 4.4.1, we now present a case study using our motivating example shown in Figure 1

**The CVSS-Based Attack Probability** The information for instantiating the CVSS-Based attack probability have been listed in Table 7 are collected as follows.

- Attack surface: All the 34 software applications we have tested are based on C or C++ language. The functions which call I/O methods (from standard C library [46]) are defined as methods (entry/exit points) in attack surface [67]. We have implemented a script to automatically identify methods from the call graphs which are generated from *cflow* [96] started from *main* function. The information about channels can be gathered from the application documentation and verified. Specifically, the connection functions and methods sometimes can be found in the developing documentations. For example, Amanda can be connected to through four different ways, namely UDP, TCP, RSH and SSH [123]. For simplicity, we only consider one type of untrusted data item in our case study, which is file, so all the exit points related to files are captured as the modification to files.

- Access right and privileges: We annotate source code to identify privilege-related functions. For example, in Amanda, the function *access_init* is used to authenticate user access right from *unauthenticated* to *authenticated*, therefore the methods appearing before this function has *unauthenticated* access right and those appearing afterwards have *authenticated*. The function *set_root_privs* is used to escalate the privileges, which means the methods invoked afterward has root privilege. Default privilege-related functions [18], such as setreuid, seteuid, setuid, setfsuid and suid,

are also annotated in source code.

- Mapping table: With the information collected from previous steps, it is easy to map attack surface to CVSS base metrics. A detailed example is already provided in Section 4.2.1.

Table 6: Amanda Channels and Untrusted Data items

| Amanda Channels | | | Untrusted Data items | | |
|---|---|---|---|---|---|
| Type | Access Rights | Count | Type | Access Rights | Count |
| TCP | remote unauthenticated | 2 | file | root | 27 |
| SSL | remote unauthenticated | 2 | file | authenticated | 6 |
| RSH | remote authenticated | 1 | file | unauthenticated | 27 |
| SSH | remote unauthenticated | 1 | | | |
| TCP | authenticated | 2 | | | |
| UDP | authenticated | 2 | | | |
| Firewall Builder Channels | | | Untrusted Data Items | | |
| TCP | remote unauthenticated | 2 | file | authenticated | 22 |
| UDP | remote unauthenticated | 2 | | | |
| IP | local authenticated | 1 | | | |

**The Graph-Based Attack Probability** To instantiate the probabilistic-based attack probability, we collect the following information.

- $p_0$: Different measurements can be used to measure the size of the software applications, for example, the lines of source code and the number of files. In our case study, we take the number of functions to represent the size of software applications. The total number of functions in a software application can be simply obtained from call graph. For example, In our study, Firewall Builder has 552 functions and Amanda has 34768 functions.

- Goal condition: We use root privilege as goal conditions for all our experiments (use maximum privilege if root can not applicable), in practice, other goals may be defined

by administrators based on the most critical resources in a given network. Table 7 lists the attack probabilities for both software applications.

**Network Attack Surface Metric** To instantiate network attack surface metric, we need to collect the following additional information to aggregate the attack probability we have gathered into a network resource graph.

- Connectivity: From the network topology (as shown in Figure 1), it is easy to obtain the connectivity between hosts.
- Security conditions: The access rights for each applications are used as pre-conditions, and the privileges are used as post-conditions. For example, Amanda could leads to root privilege [123], whereas Firewall Builder only lead to authenticated privilege [80]. Studying the existing vulnerabilities of applications leads security-related conditions. For example, Apache has a vulnerability (CVE-2016-1240) allowing local users to gain root privilege.
- Critical assets: In this case, we consider $\langle user, 4 \rangle$ as the critical asset (system administrators can choose critical asset based on their priority).

Figure 18 shows partial network resource graph for our motivating example.

| Method Group | Privilege | Access Rights | Count | Vector | Base Score | Attack Probability |
|---|---|---|---|---|---|---|
| Amanda | | | | | | |
| M1 | unauthenticated | unauthenticated | 834 | [AV:1.0,AC:0.71,Au:0.704,C:0,I:0,A:0] | 0 | 0 |
| M2 | root | unauthenticated | 672 | [AV:1.0,AC:0.71,Au:0.704,C:0.66,I:0.66,A:0.66] | 10 | 0.0191 |
| M3 | authenticated | authenticated | 1953 | [AV:1.0,AC:0.61,Au:0.56,C:0.275,I:0.275,A:0.66] | 7.5 | 0.0415 |
| M4 | root | authenticated | 297 | [AV:1.0,AC:0.61,Au:0.56,C:0.66,I:0.66,A:0.66] | 8.5 | 0.00723 |
| Firewall Builder | | | | | | |
| M1 | unauthenticated | unauthenticated | 46 | [AV:1.0,AC:0.71,Au:0.704,C:0,I:0,A:0] | 10 | 0.082 |
| M2 | authenticated | authenticated | 28 | [AV:1.0,AC:0.61,Au:0.56,C:0.275,I:0.275,A:0.66] | 7.5 | 0.037 |

Table 7: Method Groups and Their Base Scores for Amanda and Firewall Builder

To instantiate the network attack surface, necessary input information corresponding to this metric needs to be collected. This section discusses various practical issues in instantiating network attack surface metric and provides a case study based on our running example.

### Guidelines for Instantiating the Network Attack Surface Models

**The guidelines for instantiate the network attack surface metric** The general guidelines for collecting necessary inputs for network attack surface from a given network are described and discussed as follows:

**The CVSS-Based Attack Probability** To instantiate CVSS-Based attack probability, we need to collect information about

- attack surface of each dimension(channels, methods, and untrusted data items),
- the access right and privileges for the three dimensions in attack surface, and
- the mapping table between attack surface and CVSS base metrics, e.g. Table 4, and
- the normalizing constants $p_0$ and $n$,

Calculating the attack surface of a software application is well known to require significant manual effort and knowledge from domain experts [67]. Source code analysis is usually mandatory in enumerating methods for an application. Constructing call graphs to study the relationships among all the functions in the source code is usually the first step to analyze attack surface. For example, *cflow* [96] can be used to generate call graphs for source code in C language. Methods considered as direct entry/exit points are the functions that call input/output functions in C library [67]. Channels and untrusted data items can be observed at runtime from the application [67].

The privileges for the three dimensions can be checked based on a set of *uid-setting* system calls [18] which associate with change of privileges. The access right requires the study of authentication functions in source code, e.g., the methods can be invoked only after user authentications are considered as *authenticated* access rights [67].

Establishing the mapping table with CVSS vectors requires domain experts to assign numeric values based on the relationships between attack surface results and CVSS base

metrics. The aggregated attack surface base score can be calculated using the CVSS base score calculator [78].

Since $n$, the average number of methods among which there exists one zero day vulnerability, and $p_0$, the probability for attackers to discover this vulnerability, are the the normalizing constants in Equation 1, we only need to choose the values of both parameters that yield comparable results among software applications. The aggregated attack surface which represent as CVSS-based attack probability can be calculated using Equation 1 with the aforementioned inputs.

**Graph-Based Attack Probability** To instantiate the graph-based attack probability, we need to collect the following, in addition to what is already required by CVSS-based attack probability,

- $p_0$, and
- goal condition.

In Equation 2, $p_0$ represents the probability of finding a method in a software application. The probability intended as a normalizing factor based on the size of the software application, this probability can be defined by the users in a way that yields comparable results among software applications. In our examples and case studies, we use both the thousand lines of source code and the number of functions to represent the size of the software applications, respectively. It is easy to find such information about open source projects (e.g., through the Open Hub [84]) We will discuss how to estimate the attack surface of closed source software applications and the impact of non-calculable software applications in Section 4.5.

The goal condition represents the most critical assets in the organizations which can be predefined based on the needs and priority of the users. The attack surface graph can

be constructed based on the attack surface and pre- and post-conditions (access rights and privileges) which are already obtained during CVSS-based attack probability.

**Network Attack Surface Metric** To instantiate network attack surface metric, we need to collect the following, in addition to attack probabilities,

- connectivity between hosts,
- security conditions either required for, or implied by, the resources (e.g., privileges), and
- critical assets.

The connectivity information can be obtained from the network topology or using network scanners (e.g., Nessus [119]). To obtain the security conditions required for accessing hosts or resources, the configuration of hosts (e.g., the rules for firewalls) and the setting for applications (e.g., the policies for authentication) may needed to be closely examined.

The security conditions associated with the resources can be derived from the attack surface which are collected during the instantiating of *attack probability* for software applications. Finally, critical assets are able to be assigned based on the needs and priority for the network.

## 4.5 Experimental Results

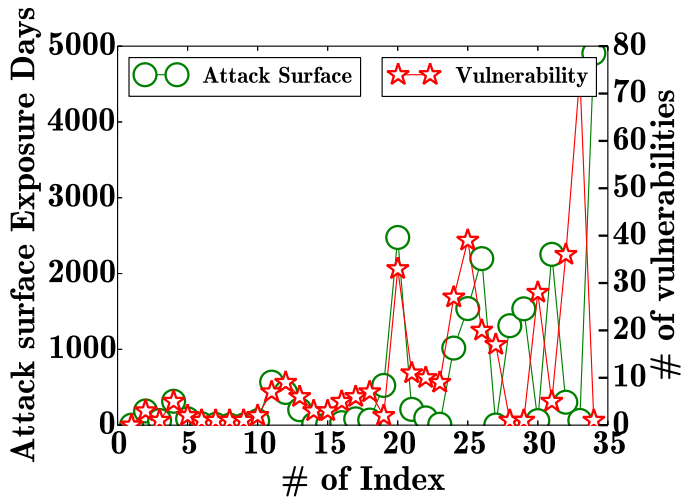
In this section, we first support our model for converting attack surface to attack probability with experimental results on the correlation between attack surface and vulnerabilities based on real software. We then conduct simulations to evaluate the performance of our heuristic algorithms proposed in Section 4.3.

### 4.5.1 Correlation Between Attack Surface and Vulnerabilities

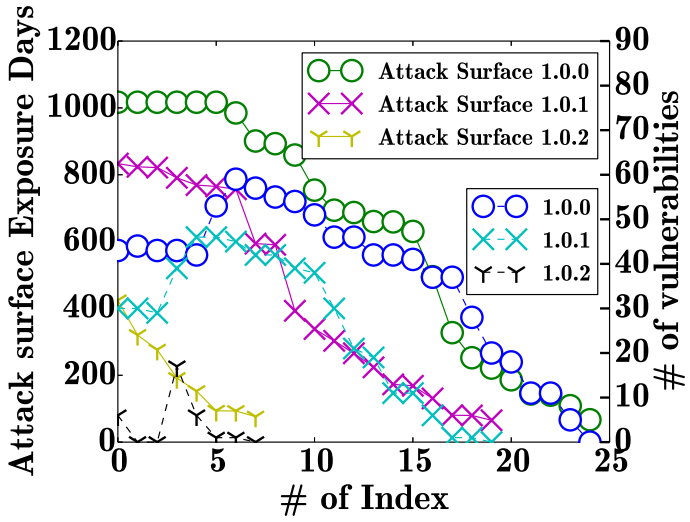
Since our model for converting attack surface to attack probability (presented in Section 4.2.1) is based on the hypothesis that attack surface reflects a software's likelihood of having vulnerabilities, we investigate this correlation by conducting experiments with real software. We examine the correlation both for different software and for different versions of the same software.

First, we examine 34 popular software and their correlation results are presented in Figure 21(a). The name of each software can be found in the Appendix based on its index number. We manually study the source code of each software in order to calculate the attack surface, and subsequently convert the result into attack probability using the method mentioned in Section 4.2.1. In Figure 21(a), the left $y$-axis and the green line show the attack surface (converted to attack probability) multiplied by the days of exposure of each software (since vulnerabilities take time to be discovered even though the attack surface of the software remains the same over time). The right $y$-axis and the red line show the number of vulnerabilities found for the same software in NVD [81].

From the results, we can see that there is a positive correlation between the number of vulnerabilities and attack surface multiplied by exposure days for most of the software (specifically, 25 out of 34). The correlation is unclear for the last few software (after index number 25). We believe the reason lies in other related factors affecting vulnerability discovery, e.g., the market share of a software, popularity of a software among attackers, and the security expertise level of typical users of a software. For example, the index number 33 is *freetype*, a popular software development library used for rendering font-related operations, which is widely used by modern video games, Opera for Wii, and many other projects [122]. Such a widely used software is usually more attractive for attackers to discover vulnerabilities, and hence becomes an outlier in our results. As another example,




(a)




(b)

Figure 21: Correlation between Attack Surface and the Number of Vulnerabilities for Different Software (a) and Different Versions of OpenSSL (b)

the index number 34 is *Amanda*, a network-based backup system, which has only one vulnerability, even though its attack surface multiplied by exposure days is relative large. We believe the reason could be that such a backup system is usually hosted in enterprise networks and operated by administrators with more security expertise and awareness, which may make the software less attractive to attackers.

Second, we examine 53 different versions of OpenSSL along 3 version branches, 1.1.0, 1.0.1, and 1.0.2, respectively, and the results are presented in Figure 21(b). The study of

different versions of the same software reduces the influence of aforementioned unrelated factors in discovering the vulnerabilities (e.g., market share). The index indicates the version numbers in chronologically order. From the results, we can see that the number of vulnerabilities has a similar trend with the attack surface multiplied by exposure days for all three branches. The branch with large value for attack surface multiplied by exposure days also has more vulnerabilities. The new versions inside each branch always have less vulnerabilities while attack surface multiplied by exposure days are also smaller. For all three branches, we can see the maximum number of vulnerabilities always appears somewhere in the middle of the branch, likely because, with a major change of version branch, it takes time for user adoption and also for attackers to change the focus. The version branch 1.0.2 is newly released since January 2015, so the attack surface multiplied by exposure days is not sufficient to create visible trends.

The above experiments, although are still of a limited scale, show a promising result supporting our hypothesis that there is a positive correlation between the attack surface and the number of vulnerabilities. Our future work will expand the scope and scale of the experiments.

**Performance of Heuristic Algorithms**

In this section, we study the performance of our proposed heuristic algorithms via simulations. All simulation results are collected using a computer equipped with a 3.0 GHz CPU and 8GB RAM in the Python environment under Ubuntu 14.04 LTS. All the resource graphs are created from small seed graphs based on realistic networks (e.g., the one shown in Figure 1), by increasing the number of hosts and resources in a random but realistic fashion.

The objective of the first two simulations is to evaluate the error rate of our heuristics (presented in Section 4.3.1). The error rate is defined in the same way as in the previous

section ($\frac{|p_{true}-p_{cal}|}{p_{true}}$ where both $p_{true}$ and $p_{cal}$ are count-based values, as described at the end of Section 4.2.3). The cost is defined as the percentage of resources whose attack surface is calculated, and denoted as $\alpha$. The reason we choose the percentage of resources instead of the absolute numbers, is that evaluating a larger network naturally implies a larger budget will be required so a relative value will be more meaningful.

Figure 22(a) shows the error vs. the percentage of calculated resources ($\alpha$) for simple heuristics and Figure 22(b) shows the same for the heuristic algorithms. The *y*-axis is shown in reversed scale in both figures in order to show the increasing accuracy of those algorithms for a larger $\alpha$. Figure 22(c) depicts the processing time of the algorithms. In all simulations, for each configuration, we repeat 500 times to obtain the average results.

*Results and Implications:* From Figure 22 (a), we have following observations. First of all, with the increase of $\alpha$, the error generally decreases, and when $\alpha$ increases to 1, which means we calculate all the resources in the network, the error of all the heuristics reaches 0 as expected. The green line with round markers is the baseline for comparison, which represents the results of the random choose heuristic and the error of this heuristic reduces almost linearly in both simulations. The frequency choose heuristic represented by the red line with vertical markers has the worst error among all the heuristics. The reason is that, the repetition of a resource does not necessarily mean the importance of this resource in determining the final result. The blue line with square and purple line with star represents the reversed topological order heuristic and the topological order heuristic, respectively. Both heuristics start worse than the random heuristic, and the reverse topological order stays worse than the random heuristic, but the topological order heuristic reduces and later becomes better than random. The reason is that, the reversed topological order tends to choose resources equally among all the paths, since the paths converge towards the end of the graph. On the other hand, the topological order heuristic chooses from beginning nodes, which might converge into one path and give better results. The most accurate one

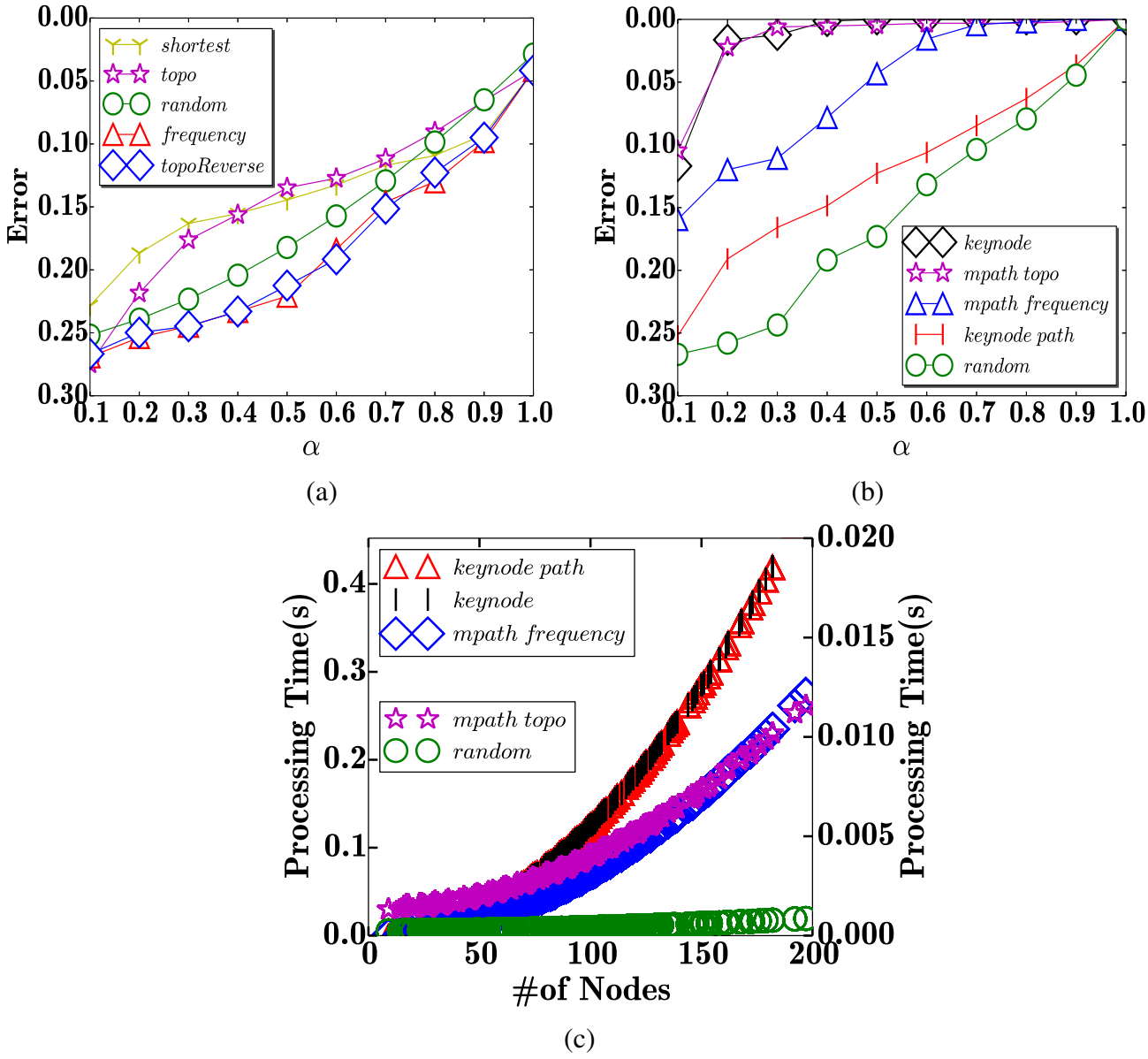



Figure 22: The Cost vs. Error for Simple Heuristics (a) and for the Heuristic Algorithms (b), and the Processing Time (c)

in Figure 22(a) is the shortest path heuristic algorithm, which combines the topological order and shortest path heuristics together. The error rate of this algorithm becomes flat when it finishes calculating the shortest path and starts to calculate other resources.

Figure 22(b) depicts the error rate of the heuristic algorithms combining multiple heuristics. We can see that the keynode and the mpath topo algorithms produce very good results, e.g., less than 0.05 error rate with only 20% of resources calculated. Such results show a

promising solution for obtaining relatively accurate network attack surface results without incurring too much cost for calculation. Here the mpath frequency and mpath topo algorithms are the combination of m-shortest path heuristic with the frequency choose heuristic and the topological order heuristic, respectively. From the results we can see that the mpath topo algorithm has less error than mpath frequency. For the keynode heuristic algorithm, we tested two different variations, one based on the change of shortest path and the other based on the change of the calculated result. From the results, we can see that those two have very different error rate, because the result-based keynode algorithm tends to gather the resources in the shortest path, whereas the path-based algorithm tends to avoid such resources.

Figure 22 (c) depicts the processing time. From the results, we can see that the keynode path and keynode result algorithms have almost the same processing time, because the majority of processing is used to preselect the keynode set. The processing time for mpath frequency is higher than mpath topo, because each iteration generates new m-shortest paths and we need to reorder frequency. But for mpah topo, we only gather m-shortest paths once and order them by the topological order. The random choose heuristic has the lowest processing time as expected. Overall, we can conclude that the mpath topo algorithm is the best choice in terms of both error rate and processing time.

### 4.5.2 The Impact of Non-Calculatable Resources

Instead of calculating attack surface for only a subset of resources, approximately calculating for all resources is another technique to reduce the total cost. For example, the Microsoft research team has proposed attack surface approximation method based on stack trace analysis [120]. In a trial on Windows 8, the authors discover that the approximation selects 48.6% of software but includes 94.6% of the known vulnerabilities. In this section, we would like to evaluate the impact of this idea through simulations, and show

the relationship between $\alpha$ and approximation rate of attack surface, denoted as $\beta$, in our simulations.

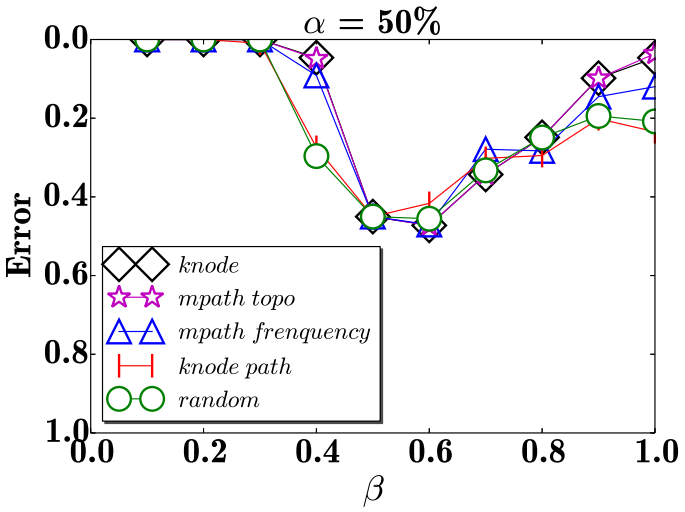
Furthermore to calculate attack surface of applications, we generally need to have direct access to the source code of the applications. However, some hosts may have closed source applications. For example, according to the statistical result [115], 84.34% of desktop operating system is Windows. For these type of hosts, we will not able to calculate attack surface. On the other hand, it maybe difficult to fully calculate the attack surface for some open source software applications due to their sheer size. Recall that we use source lines of code (SLOC) to measure the size of software applications. With the evolution of software applications, the size of the source code increases dramatically. For example, operating system Debian's SLOC increases from 55-59 SLOC million (Debian 2.2 in 2000) to 419 SLOC million (Debian 7.0 in 2012). Therefore, we have divided software applications into three different categories in terms of the feasibility of calculating attack surface as follows.

- Non-calculable software resources: The resources, provided by software applications with inaccessible source code, called *non-calculable* attack surfaces.

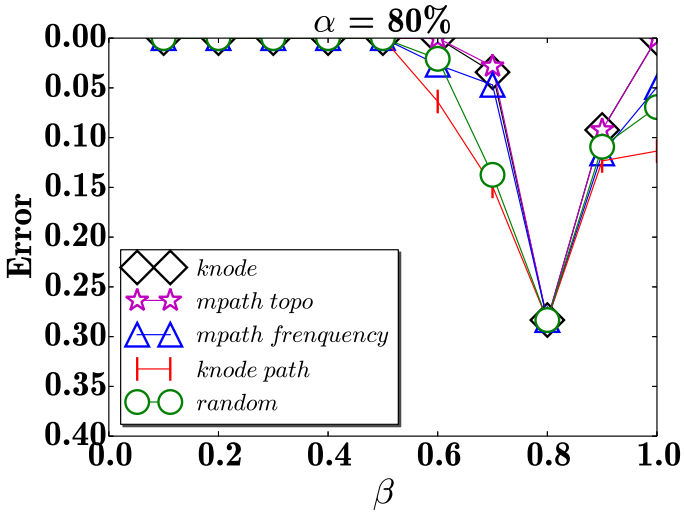
- Partially calculable resources: Software applications with large SLOC, e.g., Debian which is open source but with 419 SLOC million size of source code, may be too costly to generate call graph and fully calculate the attack surface. A more feasible solution in such a case may be to calculate attack surface only for some critical components of the software applications to balance between accuracy and cost. Therefore, the resources provide provided by software applications with very large SLOC are called *partially* calculable resources.

- Fully calculable resources: For open source software applications with small or medium SLOC, it is generally feasible to generate call graph and fully calculate attack surface. The services provide by such software applications with small SLOC

are called *fully* calculable resources.




(a)




(b)

Figure 23: (a) The Error of the Algorithms with $\alpha = 50\%$ (b) The Error of the Algorithms with $\alpha = 80\%$

The first simulation studies the impact of partially-calculable resources, and the impact of non-calculable resource is also examined through this simulation. The error rate is

defined in the same way as in the previous simulations. The budget is defined as the percentage of effort spend to calculate attack surface in one network, denoted as $\alpha$. Partially-calculable rate is defined as the percentage of attack surface of each resource which is calculated, and denoted as $\beta$. Calculating $\beta$ percent of a software application may result in approximated value of the exact attack surface. However, it is impossible to obtain such an approximated value. Compared to the real attack probability (modeled in Section 4.2) derived from attack surface, a lower value of attack probability may be obtained because of the less I/O methods in the chosen percentage of the source code, on the contrast, a higher value may appear because more I/O methods are included in less size of the source code. Therefore, in this simulation, we set an estimation range for the approximated attack probability value. Assuming the true attack probability in a software application is $p$, the estimation range is defined as $[(1-\beta)*p, min((2-\beta)*p, 1)]$, $(1-\beta)*p$ lower than the true $p$ value and $(1-\beta)*p$ higher than the exact p (because the probability needs to stay in the range from 0 to 1, we use $min((2-\beta)*p, 1)$ as the upper bound). An approximated attack probability value is randomly generated from the estimation range.

Unlike in the previous simulation, $\alpha$ only represent the percentage of effort to calculate attack surface in this simulation. The percentage of resources whose attack surface are calculated will depend on both $\alpha$ and $\beta$. After calculating $\alpha$ percent of the resources in one network, we are still able to calculate $\alpha - \alpha*\beta$ percent of attack surface since we still have budget left. After calculating $\alpha - \alpha*\beta$ percent of the resources, the remaining budget for attack surface is $\alpha - \alpha*\beta - (\alpha - \alpha*\beta)*\beta$. It is easy to calculate the overall percentage of calculated resources $\frac{\alpha}{\beta}$, see the detailed calculation as followed. Notice that when $\alpha > \beta$, we have extra budget to calculate attack surface. In this case, we apply the extra budget to fully calculate the remaining resources according to the algorithms' order of calculation.

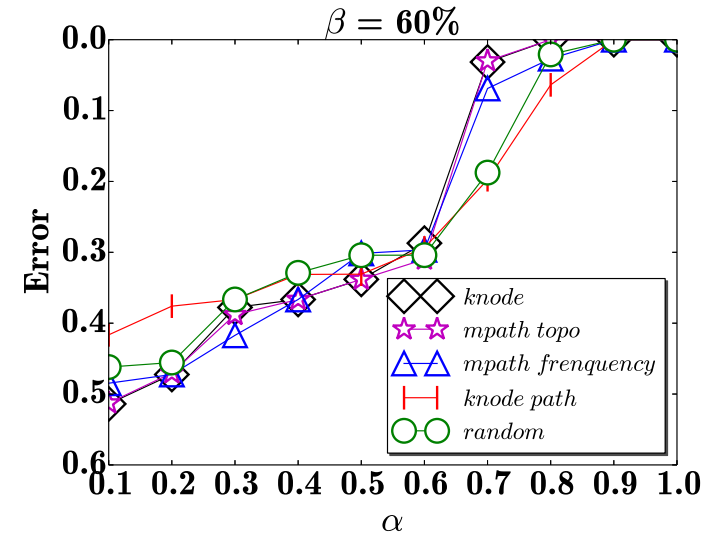
Assume $\alpha$ is the percent of the budget effort to calculate the attack surface in one

network, and $\beta$ is the percentage to calculate in each resource. Let $n$ be the number of the resources can be calculated, and $N$ be the total number of resources in one network. The formula to calculate the percentage of resources for which the attack surface can be calculated is $\frac{n}{N}$=$\sum_{i=0}^{i=inf} \alpha * (1-\beta)^i$. This is a geometric series with the constant ratio (1-$\beta$), therefore we have $\frac{n}{N} = \frac{\alpha}{\beta}$.

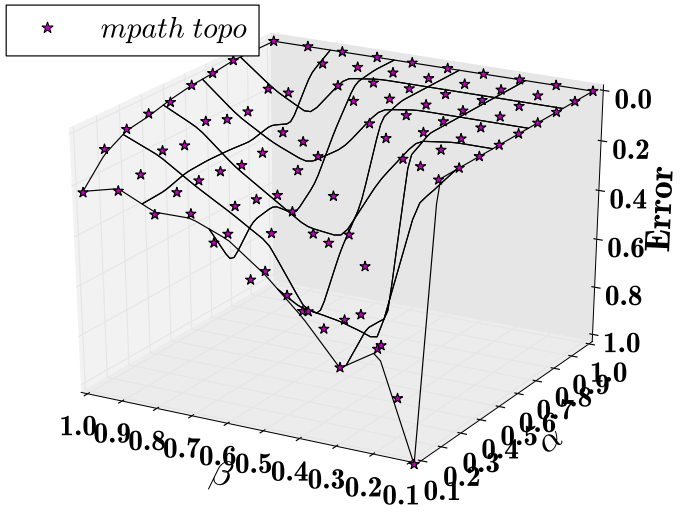
*Results and Implications*: In Figure 23a, $\alpha = 50\%$, which means the effort budget is 50% of the overall effort to fully calculate every attack surface in the network. Therefore when $\beta < 50\%$ ($\frac{\alpha}{\beta} > 1$), we have extra budget to apply back to calculate resources according to the algorithms' order. The smaller value of $\beta$, the more extra budget left to apply fully calculation after first partially calculating every service. Comparing with $\alpha = 50\%$ in Figure 22a and Figure 22b, the error rates are smaller when $\beta$ is smaller than 30%, mostly because the effort spent on partially calculating the services provides a rough ranking of the resources, and the remaining efforts are used to fully calculate the services that contribute the most to the final result of network attack surface. When $\beta$ increases to 40%, the error rates of algorithms become worse than $\alpha = 50\%$ in Figure 22a and Figure 22b, mainly because the remaining efforts are not sufficient to fully calculate the services on the optimal path.

The error rate is increasing till $\alpha = \beta$, which is the worst case in Figure 23a, simply because when $\alpha = \beta$, the attack probability used to calculate the final result are all with partially calculated values (the attack probability value for each resource falls in the estimation range we mentioned earlier). Similar trend can be observed in Figure 23b when $\beta \leq \alpha = 80\%$. We can see that the error rate is 0.5 when $\beta = 50\%$, because we set the approximation range as 50% lower to 50% higher than the exact attack surface (1 is the maximum value for the upper bound). And the error rate is 0.28 when $\alpha = \beta = 80\%$ which is still close to our approximation range as 20% lower to 20% higher than the exact attack surface.

Unlike the increasing trend of the error rates when $\beta \leq 50\%$, the error rates are decreasing when $\beta > \alpha$ for all the algorithms in both Figure 23a and Figure 23b, mainly because the approximated attack probability get closer to the true attack probability when more source code are used in calculating attack surface. When $\beta = 100\%$, this simulation becomes the same as the previous simulation.




(a)




(b)

Figure 24: (a) The Error of the Algorithms with $\alpha = 50\%$ (b) The Error in $\alpha$ and $\beta$

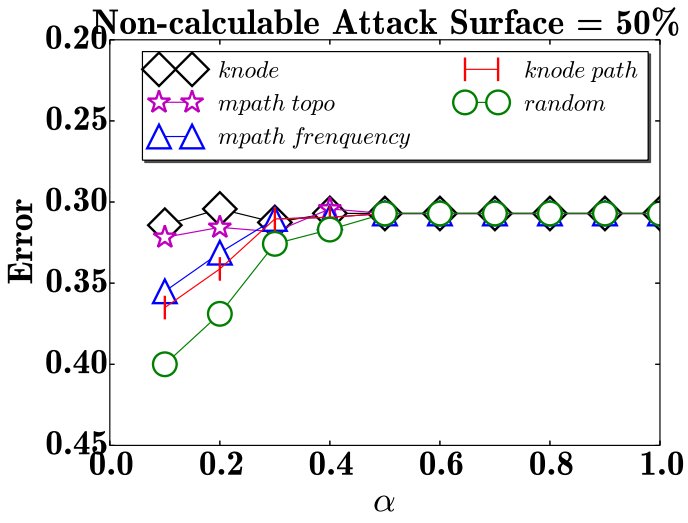
Figure 23 presents the relationships of error rate and $\beta$ with fixed $\alpha$ for the algorithms,

whereas Figure 24a presents the relationships of error rate and $\alpha$ with fixed $\beta = 60\%$. When $\alpha \leq \beta$, only $\alpha$ percent of the services can be calculated partially, and the error rate decreases with the percentage of calculated services. When $\alpha > \beta$, extra budget can be applied back to fully calculate services along the algorithms' choosing sequences, and the error rate thus distinguishes the different performance of algorithms (knode and mpath topo have the best choosing sequences). Figure 24b presents relationship among $\alpha$, $\beta$ and the error. The concave upward part from the 3D graph shows the special case when $\alpha = \beta$, which has been discussed in the previous paragraph.

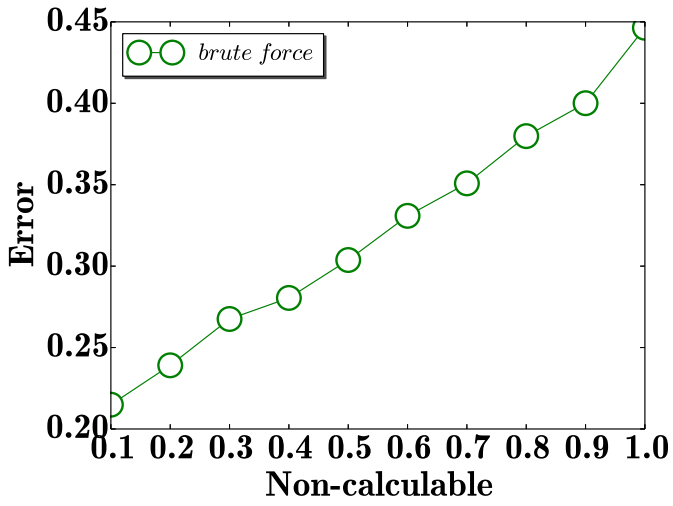
Finally, this next simulation focuses on the impact of *non-calculable* resources. The definition of $\alpha$ is the same as in the first simulation, i.e., the percentage of resources whose attack surface is calculated. We assign 0.68 as the probability for non-calculable resources (which is the attack probability from the average value of all CVSS scores in NVD [81]) in this simulation.

*Results and Implications:*

Figure 25a presents the impact of our algorithms when 50% of the resources are *non-calculable* in a network. The error rate decreases till $\alpha = 50\%$, while error rate remains the same when $\alpha > 50\%$. Our algorithms help to reduce the error rate and the knode and mpath topo algorithms give the best performance in all the simulations. Figure 25b studies the impact of *non-calculable* resources by using the brute force algorithm (i.e., regardless of the budget, calculate 100% for every calculable attack surface). The error rate increases linearly with the increasing of non-calculable resources in one network. When non-calculable resources reach 100%, our metric essentially becomes the same as *k*-zero day safety metric [126], which still can be considered as a useful measurement even though the metric no longer distinguishes the resources.




(a)




(b)

Figure 25: The Error vs. $\alpha$ of Algorithms with 50% *Non-Calculable* Resources (a) and the Percentage of *Non-Calculable* Resources vs. Error (b)

## 4.6 Conclusion

An intuitive notion of attack surface at the network level has prevented applications from inheriting the precise and quantitative reasoning power of the original attack surface metric. In this topic, we have designed methods for lifting this concept to the network level as a formal security metric for measuring networks' resilience against zero day attacks. The

correlation between attack surface and vulnerabilities was validated through our preliminary experimental results. We have also shown through algorithm design and simulations that the cost of calculating attack surface for network resources could be saved without losing too much accuracy.

# Chapter 5

# Learning-Based Model for Software Vulnerability Prediction

In this chapter, we describe the efforts we made in the third proposed research topic; the learning based model for software vulnerability prediction.

## 5.1 Introduction

Software vulnerabilities are considered a key threat to critical networks, such as power plants, governmental or military organizations, and data centers. Predicting and studying software vulnerabilities helps administrators in improving security deployment for their organizations, as well as aids them in choosing the right applications among those with similar functionality, and also provides assistance for software vendors in estimating the security level of their applications. The VDMs we have discussed in Section 2.5 either predict vulnerable components inside one software application, or establish specific prediction model for one software application based on large amounts of historical data. Although the prediction of vulnerable components helps in discovering or fixing security issues, it only provides this information in one software application. It fails to provide overall security

interpretations of the software applications. Although statistical VDMs can predict the future number of CVE vulnerabilities, they require large amounts of historical vulnerability data as the inputs to their models. The prediction capability on newly released software applications is currently lacking in the literature.

In this topic, we perform a large-scale empirical study to investigate the relationship between software features and the number of vulnerabilities in the software. Our study focuses on 780 open-source applications collected from GitHub, and the entire version history of Google Chrome, which consists of 11,454 releases as of November 2017.

First, we describe three use cases to motivate our study.

**Use Case 1: Estimating the Number of Vulnerabilities in Newly Released Applications.** Vulnerabilities in existing software applications are published as CVE entries in the NVD database, and are often used as an indicator to demonstrate the security level of an application. However, new applications lack this official information since any vulnerability has yet to be discovered or reported. Estimating the number of vulnerabilities based on certain intrinsic features, e.g., the size or the number of I/O functions, would help potential users to compare products, as well as help software vendors to promote their products. The concept of attack surface fits this purpose, as it captures the intrinsic properties of an application. However, in reality, a large attack surface is not necessarily equivalent to a large number of the vulnerabilities, since multiple other facts could have affected the vulnerability discovery process. Unknown vulnerabilities may remain unknown for unpopular software applications. In contrast, popular libraries, e.g., OpenSSL, contain a large number of vulnerabilities, as they are more widely used in many software applications and therefore more attracting to attackers.

Therefore, it is insufficient to compare attack surface among software applications. A comprehensive study about the relationships between vulnerabilities in the software and

software features is needed for both software vendors and purchasers. With sufficient information from software metrics, vendors could assume the popularity of their software to obtain the future possible number of vulnerabilities. This quantitative result helps to better position new release software applications in the market in terms of revealing an important decision factor to potential purchasers.

**Use Case 2: Software Deployment Decision Making:** As mentioned before, it is insufficient to only use the attack surface to compare software applications when the administrators want to deploy one application from applications with similar functionalities. It is also not always possible to use only the NVD database to make this decision, since some applications might be newly released (which do not have enough entries in the database and, some of which are not reported in NVD database). The method mentioned above, which predicts the number of CVE vulnerabilities based on the features, could be used to solve this issue, providing fairness comparison among software applications.

**Use Case 3: Completing the Existing VDM:** The existing VDMs that rely on large amounts of historical vulnerability data are mainly time-based models. Closest to our study, Rahimi et al. [97] apply two code metrics, i.e., code quality and code complexity, in their model to lower the dependency on the historical data. The authors obtain better precision and less divergence compared to the statistical VDMs by using stochastic model with the code metrics in their study. Our study provides a more comprehensive look at the relationship between the features and the number of vulnerabilities, which could be used as the input in improving the statistical VDMs.

## 5.2 Background

This section provides the background knowledge used in this topic.

### 5.2.1 Statistical Analysis of data

To improve the accuracy of the testing, *Min-Max standardization* is applied to the features.

$$z_i = \frac{x_i - min(x)}{max(x) - min(x)} \tag{3}$$

where $x = (x_1, \ldots, x_n)$ are the original data set and $z_i$ is the $i^{th}$ normalized data corresponding to the original data set. *Min-Max standardization* function maps the dataset into the range (0,1).

### 5.2.2 Data Visualization

**Direct Visualization Methods:** *Matrix of scatter plots* displays all the possible pairwise in selected plane $R^2$ or space $R^3$, which is the most common geometric visualization method in data visualization. However, with the increasing number of features, it is difficult to interpret the result of this method.

*Parallel coordinates* [45] visualizes the features as parallel lines on the graph and map the values of the features as coordinates on the parallel lines. Feature coordinates for the same type of observation are connected through parallel lines. This method fails to capture the data structure when the density of the coordinates is high.

**Dimensionality Reduction Methods:** In the dimensionality reduction methods, also called the projection methods, normally the multidimensional datasets are transformed into 2D or 3D data-points. The challenge for these methods is to represent the maximum properties of datasets into a lower-dimensional space.

*Principal component analysis* (PCA) [91] was first proposed in 1901, applies linear transformations on the datasets and keeps the independent combinations of features as the components. Most of the information in the original dataset has been stored in the

first few components, which allows it to map out the multidimensionality data into lower-dimensional space. A more recent algorithm *t-SNE* [66] was first introduced in 2008, is a nonlinear dimensionality reduction technique. We use both PCA and t-SNE in our study: PCA to combine the features and t-SNE for visualization.

### 5.2.3 Feature Selection and Evaluation Methods

A commonly known effect in machine learning, *curse of dimensionality*, points out that an increasing feature space dimensionality weakens the reliability of trained analysis systems [95] by *overfitting* the data. An efficient solution is to apply *feature selection* to find feature subsets, with lower-dimensional space, which leads to more reliable learning results. Feature selection is also known to enhance the classification performance, lower the computational costs, simplify the classifiers and provides better understanding in the classification problems [103]. Three types of feature selection methods, namely, the filter methods, the wrapper methods and the embedded methods, are applied in our study to achieve feature subsets.

Filter methods evaluate the score of each feature according to certain criteria and the experts then choose the subsets based on the scores, e.g., correlation-based feature selection (CFS), and mutual information (MI). However, filter methods only consider the relationship between the pairs of features; the relationships between multiple features are ignored in this type of feature selection methodology.

Wrapper methods involve the learning algorithms to evaluate the relevance of the feature sets, e.g., k-nearest neighbors algorithm ($k$-NN), which classifies object based on the k-nearest neighbors. Ideally, wrapper methods test all the possible permutations for the feature subsets and output the ones with the best results in terms of accuracy. The computation time in searching for the best feature subsets from the possible permutations of the feature subsets grows exponentially with the number of features; heuristic algorithms,

such as sequential forward selection (SFS), are proposed to tackle this search problem. SFS starts from an empty set and adds the feature in order to obtain the maximal score in the iteration to the feature set.

Embedded methods build the feature selection process inside the learning algorithm, e.g., decision trees [75], random forests [14]. The feature selection process is using the entire dataset as the input to generate the feature.

### Evaluation Methods

In our study, we build on GitHub project and Chrome project as regression models, of which the target variables are the continuous numeric values. We implemented six selected regression classifiers for the GitHub project: logistic regression (LR), SVM for regression, Random Forest (RF), Decision Tree (DT), Boosted Tree (BT), Artificial Neural Network (ANN); Neural Network for time series is applied in Chrome project.

**Validation of Regression Models:** In the regression model, we use tenfold cross-validation to separate the data into training set and testing set. The accuracy of the model is described through mean absolute error (MAE), which measures the difference between two variables, mean-squared error (MSE) and root mean square (RMS), widely used functions for analyzing the performance of linear regression. The mean absolute percentage error (MAPE) is used for the dispersion measure.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n} \tag{4}$$

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2 \tag{5}$$

$$RMS = \sqrt{MSE} \tag{6}$$

$$MAPE = \frac{\sum_{i=1}^{n} \frac{|y_i - x_i|}{y_i} * 100}{n} \tag{7}$$

where $y_i$, $x_i$ are the predicted value and the actual number of the target, respectively, and n is the total number of the observations.

**Evaluation of Classification Models:** To complete the background knowledge, we also present the evaluation of classification models. Commonly, the confusion matrix is a table that displays the performance of the classification algorithm. True positive (TP) is the number of true observations that classify to the true label, and false positive (FP) is the number of true observations that classify to the false label; false negative (FN) is the number of false observations that classify to the true label and true negative (TN) is the false observations that classify to the false label.

| | | Actual class | |
|---|---|---|---|
| | | True | False |
| Predicted class | True | True Positive | False Positive |
| | False | False Negative | True Negative |

Table 8: An Example of Confusion Matrix

Based on the elements from the confusion matrix, we can further define the indicators to evaluate the classification models.

*Accuracy:*

$$Acc = \frac{TP + TN}{TP + TN + FP + FN} \tag{8}$$

*True negative rate:*

$$TNR = \frac{TN}{TN + FP} \tag{9}$$

*False negative rate:*

$$FNR = \frac{FN}{FN + TN} \tag{10}$$

*False positive rate:*

$$FPR = \frac{FP}{TP + FP} \tag{11}$$

*Recall:* The true positive rate.

$$Recall = \frac{TP}{TP + FN} \tag{12}$$

*Precision:*

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

*F1-measure:* F1-measure is harmonic mean of Precision and Recall.

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{14}$$

## 5.3 Dataset Collection and Preparation

In this section we present the datasets that we collect in our study and the methodologies we apply to prepare the datasets.

### 5.3.1 Datasets

In this study, we obtain two datasets to perform the empirical study about the correlation between the features and the software application vulnerabilities. The first dataset, which we obtain from GitHub (a version control and source code management hosting service), contains features of 780 open source projects and 6498 vulnerabilities from NVD database[1]. This dataset provides the most complete mapping from open source projects on

[1] https://nvd.nist.gov/vuln/data-feeds

GitHub to CVEs. In contrast, the closest work to our study, conducts the first large-scale mapping of CVEs to GitHub commits for only 66 open source applications [92]. The second dataset consists of the features and vulnerabilities for 11,454 versions of the Google browser Chrome from Chromium repository [2]. To the best of our knowledge, this is the first large-scale dataset, spanning over 9 years, that includes multiple features about Chrome and the corresponding vulnerability information. In contrast, the other related work, Stuckman et al. [116], collects dataset from three open-source web applications, 100 phpMyAdmin, 95 Moodle, and 23 Drupal 6.0 versions in their experiments.

### GitHub Project Dataset

This dataset contains the software metrics, developer metrics, software property metrics, security metrics, popularity metrics and the number of vulnerabilities for 780 open-source projects from GitHub. To the best of our knowledge, this dataset contains the most complete Git repository which contains at least one published vulnerability.

| Metrics | Features |
|---|---|
| Software Metrics | size<br>#files<br>#program-files<br>blank<br>comment<br>code |
| Developer Metrics | #contributors<br>#commits |
| Software Property Metrics | age<br>#labels<br>last push date<br>language distribution |
| Popularity Metrics | #stars<br>#watches<br>#forks |
| Security Metrics | #issues<br>#functions<br>flawfinder outputs |

Table 9: GitHub Features Collection

**Software Metrics:** The software metrics we collected from open-source projects on GitHub contain 4 different granularities: the size, the number of files, the number of code files and

[2] https://chromium.googlesource.com/chromium/src.git/

the SLOC in order to represent the *cyclomatic complexity* of the software. To measure a project's size, one option is to rely on GitHub API, which provides a *size* attribute for each repository. However, the reported size reflects the server-side storage requirement for all revisions and after some level of storage optimization. Thus, we resort to cloning all projects locally and we measure only the size of the HEAD tree, i.e., the view of the latest revision's files tree. All projects occupy a total 106GB on disk, while the API's size attribute reported only 73.5GB. The total size representing all latest revisions is 31GB.

The number of files is obtained through the *git-ls*. This command is also used in collecting the number of files in Chrome data. Cloc [23] is an open source tool used to count the blank lines, comment lines, and physical lines of source code in multiple programming languages. The number of code files is the output from cloc, which only takes the code files into consideration. SLOC for each repository has been obtained through cloc same as the way to obtain SLOC for Chrome versions.

**Developer Metrics:** In this study, we focus on comparing the software level changes among the software applications. Therefore, the developer metrics we collected are the number of commits in the entire life of the open source projects, the time-stamps of the commits and the frequency of the commit for each repository, as well as the number of contributors. The number of commits represents the number of changes/patches that have been made in the life cycle of each project. In addition, the frequency of the commit reveals the update speed of the project. The time stamp of the commit may associate with the error made by the developers according to Sliwerski et al. [114], e.g., the commits submitted on Fridays have more chance to contain bugs. Therefore, time stamps for the commits are also considered in the later cases. The number of contributors reveals the maintenance or developing speed for each project, which is also obtained through the meta-data from GitHub repositories.

**Software Property Metrics:** In this study, we collect the intrinsic properties of software, e.g., the release date, the last push date, the languages distributions and the labels. Each of the properties correspond with a certain factor that related to the discovery of vulnerabilities. The release date reveals the life span of the open source project that is related to the exposure period. After the release date of that project, everyone could fork and have access to the project and discover its bugs. The last push date is related to the active change done to the project, which is corresponding with the actions made by the developers, e.g., the fix of the bug, and the addition of new components. It demonstrates the recent maintenance of the project. The language distributions correlate the project to language specific fixes and narrow down the comparison scope. Finally, the labels, which are assigned by the owners of the projects, illustrate the functionalities of the project.

**Popularity Metrics:** The popularity metrics reveal attack probability for each project; the higher the popularity, the more attention to the project. In this study, the popularity metrics are collected from GitHub metadata, e.g., the number of *fork*, *star*, and *watch*. The number of forks is the number of the copies of the projects. Contributors could work on the fork and make the fix of the project. Once the requests are pulled from the fork contributors, the original owners could decide whether to include the fixes from the fork into the original repository. Therefore, the number of the forks reveals the popularity of the repository among other developers. The number of stars and watches of a project show the attention given to the repository among the GitHub registers. *Star* gives the track of the project to users. With the star to certain project, it is easier to find the project. *Watch* enables users to receive notifications about project. The number of *star* and *watch* demonstrates different level of attentions from Git users.

**Security Metrics:** The security metrics we consider in this study cover three aspects: the potential attack likelihood, the existing attack likelihood, and the history attack likelihood.

We use the flaws that are discovered by Flawfinder [132] as the potential attack likelihood for one application. Flawfinder identifies the dangerous functions, such as *memcpy*, and present the flaw points with multiple severity levels. Furthermore, in attack surface concept [67], the entry/exit points (the methods which directly/indirectly invoke the I/O function) of a software application contains the possible explorable points from one software application. Ideally, the attack surface from one software application would be a better metric to indicate the potential attack likelihood. However, to obtain the attack surface, we need to construct the call graph from the *main* function in one software application, which does not always exist in the project from GitHub. In this study, we use the number of functions, which is upper bound of the attack surface, as the second potential attack likelihood metric for one software application. We obtain the number of open security issues that have been released to each project as the existing attack likelihood. The open issues contain the software bugs reported from the users and the tasks for the project maintainers. We only collect the security-related issues and ignore the task-oriented issues. The history attack likelihood is represented as the number of closed security issues.

**Number of Vulnerabilities:** The data source for the number of the vulnerabilities is the NVD database; the same as the one we used for Chrome dataset. We first use the NVD database [81] in the reverse search to obtain the GitHub project, which contains at least one vulnerability. Then, with the name of the project, we obtain the number of vulnerabilities for each repository. In this study, we obtain the total number of vulnerabilities for the repository, which are not separated by versions. In this way, we don't have the mislabeling version issue, which we will explain in the Chrome dataset, however, the mishandling of the links and the vulnerable products increase the difficulties in obtaining an accurate number of vulnerabilities for each project. For example, CVE-2017-15041 contains the GitHub link in it's $\langle vuln : source \rangle$ which fits the first reverse search rule we apply in our study. However, in the CVE entry, no specific product is listed, but only the language is mentioned, which is

Go language. 472 CVE entries in 2017 (more than 5%) are not associating with any affected products in the NVD database. CVE-2017-13570 is the CVE for the product *blackcat_cms* contains GitHub link in the $\langle vuln : references \rangle$, however, the link only points to the page discuss about this vulnerability not the page for the product. Furthermore, CVE-2017-9609 corresponds with *blackcat_cms* in $\langle vuln : product \rangle$ but connects to different git link in $\langle vuln : reference \rangle$ than the CVE-2017-13570. By simply taking the link, it would be difficult to distinguish the correct repertory for the product.

**The Chrome Dataset**

This dataset includes the software metrics, code churn, release information, and the vulnerability numbers for 12000 versions of Chrome back to 2008. Originally, we try to obtain the data from Open Hub [3], a website to track and compare open source projects, to obtain the software metrics and the release for the Chromium project. We discover the list of vulnerabilities corresponds to various Chrome versions under the security link [4]. It seems trivial to collect the version information corresponding to the number of vulnerabilities, however, the inconsistency of data has been discovered when we compare the release data gathered from this website to the release tag from official Chrome repository. The same inconsistency is discovered when we compare the number of vulnerabilities between Open-Hub and NVD database. In order to obtain the correct features for our study, we decide to collect the data from the official websites.

**Software Metrics:** The software metrics, considered in this study are based on the software level, which is different from other studies that are based on predicting the vulnerable components or files in one software applications. Our study focuses on the trend of vulnerabilities among versions. Recall that Chrome is a huge open source project and it is not easy

[3] https://www.openhub.net/p/chrome
[4] https://www.openhub.net/p/chrome/security

to obtain any of the software metric. Besides, Graylin et al. [52] have shown empirically that the *cyclomatic complexity* and line of code is near-linear. Adding files often associates with introducing new features into one program. Therefore, we choose the line number of code to represent the complexity, and the number of files to represent the components of the source code.

We apply cloc to 11,454 versions of Chrome since 2008. Given the increasing code size, cloc took up to five minutes to complete, resulting in a full week of computation using multiple parallel instances. The number of files is counted as the result of *git -ls*, which shows the number of files under the version checked out, while the number of the code files is counted by cloc, which only takes the programming files into account.

**Release Information:** The release time for Chrome versions is tagged on the Chromium repository by authors; each version associates with one time tag in the repository. However, Chromium project has migrated from SVN to git in 2014 April, which means the versions released before April 2014 are all tag only for the migration time and not the release time. Requesting the time tag from git repository only provides the migration dates for the versions before 35. The problem for this migration time tags is that we gather the version release time later than the vulnerability in that version. To solve this in accurate data, we use the first release vulnerability date as the approximate date for the versions before 35. However, this method is not compatible with the problem we discovered in the gathering for the number of vulnerabilities. See the details in the gathering for the number of vulnerabilities.

In order to get accurate release time for all the versions, we merge the old SVN used by Chromium project [5], which contains the proper release time for the versions before 35, to the time tags from new Chromium git repository.

[5]https://src.chromium.org/viewvc/chrome/releases/

| CVE | Description | Affected Versions |
|---|---|---|
| CVE-2011-3034 | Google Chrome before 17.0.963.56 allows remote attackers to cause a denial of service (application crash) via an empty X.509 certificate. | since version 0.1.38.1 |
| CVE-2012-5143 | Integer overflow in Google Chrome before 23.0.1271.97 allows remote attackers to cause a denial of service or possibly have unspecified other impact ... | since 23.0.1271.0 |
| CVE-2016-5136 | Use-after-free vulnerability in extensions/renderer/user_script_injector.cc in the Extensions subsystem in Google Chrome before 52.0.2743.82... | 51.0.2704.106 |

Table 10: An Example of CVE Interpretation

**Number of Vulnerabilities:** We download the CVE database from NVD to obtain the number of vulnerabilities for each version of the Chrome for the later studies. In the CVE database, the tag, $\langle vuln : product \rangle$, which connects the CVE to the affected programs. For example, in CVE-2016-5136, the vulnerable program is cited as Chrome version 51.0.2704.106. However, this citation for the versions has been interpreted in a very different ways in the history of Chrome vulnerabilities. In early versions, for example the versions released before 2011, the keyword *before* has been interpreted as the versions since the first version of Chrome (which is the first entry in Table 10). The keyword *before* in vulnerable version, which was released after 2012 and before 2014 has been interpreted as the versions since the first version of the vulnerable major version (See the example in second row of Table 10). And the recent vulnerabilities only cite the exact one version before the vulnerable version (See the example in last row of Table 10).

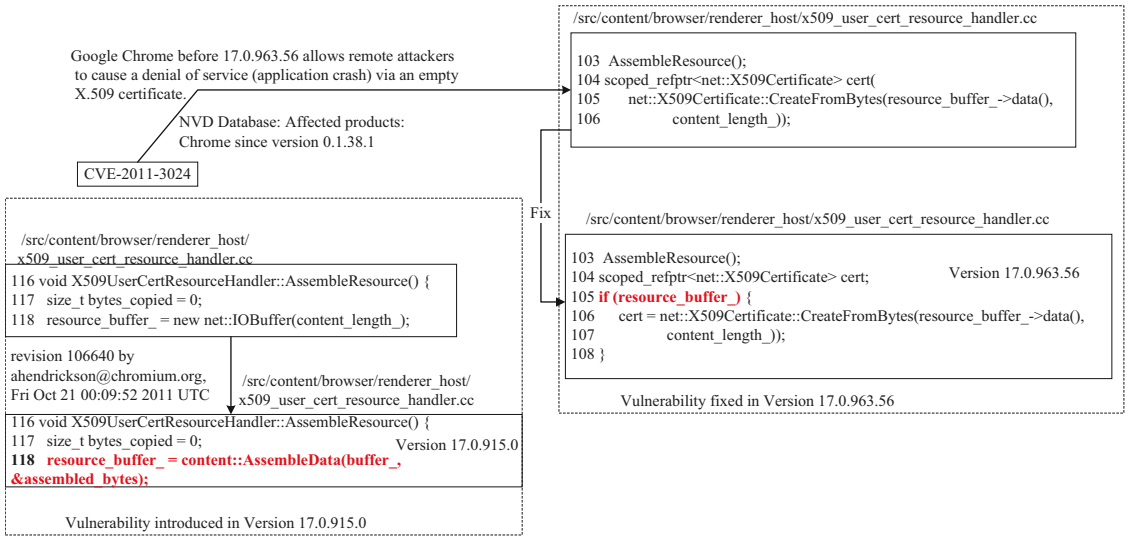



Figure 26: Inaccuracies in NVD Database CVE-2011-3034

The methods used in interpreting the keyword *before*, *prior to* and *and earlier* in vulnerabilities are all inaccurate. Figure 26 shows the detailed information about CVE-2011-3024, and how the vulnerability introduced in Chrome. The vulnerable code has been fixed in version 17.0.963.56 by adding *if* condition as boundary for variable *resource_buffer_* (which caused the empty X.509 certificate in the previous versions). By using the *git blame* command on the vulnerable lines, we could know the change dates for the vulnerable lines. We checkout the one version of the fixed version to *git blame* the fixed lines to get the date for the last change of those lines. However, in reality the *fixed* lines are not equivalent to the code, which introduced vulnerabilities. For example, in Figure 26, the vulnerability is caused by the fact that no boundary check is applied on the variable *resource_buffer_* in CVE-2011-3024, but the vulnerability was not introduced by this variable. The vulnerability was introduced by the version 17.0.915.0, when they changed the method to generate this variable, which allows the null value in this variable (the original method does not generate null value, so no boundary check was needed). The result is different than the first interpretation of the keyword *before* in CVE entries.

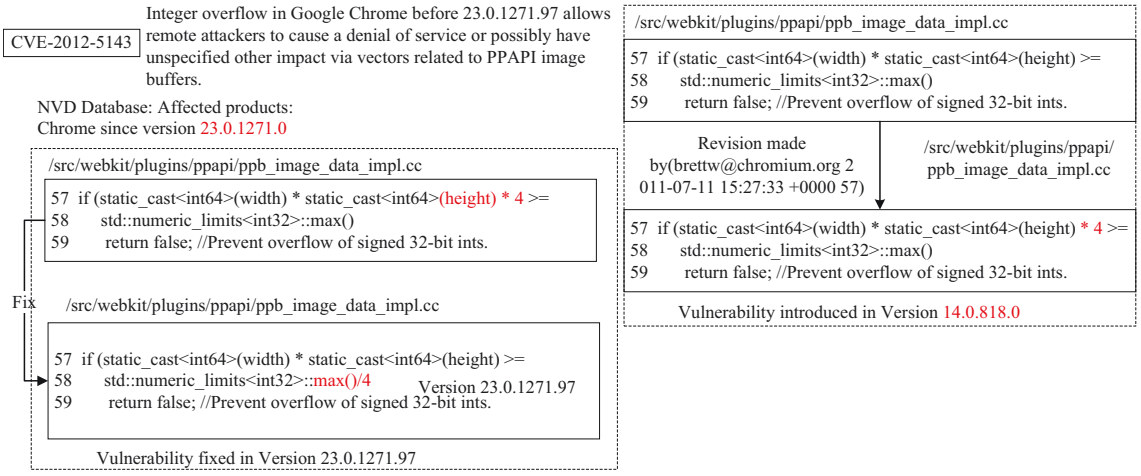



Figure 27: Inaccuracies in NVD Database CVE-2012-5143

Figure 27 shows that the actual version introduced the vulnerability is in version 14.0.818.0, which is way earlier than the first version in the same major version 23. And Figure 28

demonstrates the *git blame* result around the fixed lines, line 107 and line 108 in 2016-07-16, which the entire function was introduced in 2014-06-27. We further check for this commit and discover that the entire file was introduced in version 37.0.2062.7, which is significantly earlier than the version 51.0.2704.106 mentioned in NVD database.

CVE-2016-5136: Use-after-free vulnerability in extensions/renderer/user_script_injector.cc in the Extensions subsystem in Google Chrome before 52.0.2743.82 allows remote attackers to cause a denial of service or possibly have unspecified other impact via vectors related to script deletion.

NVD Database: Affected products:
Chrome version 51.0.2704.106

| /src/extensions/renderer/user_script_injector.cc | **Vulnerability introduced in Version 37.0.2062.7** |
|---|---|
| (rdevlin.cronin@chromium.org 2014-06-27 17:07:34 +0000) | 102 void 37.0.2062.7::OnUserScriptsUpdated( |
| (hanxi2015-03-11 16:40:06 -0700) | 103 const std::set<HostID>& changed_hosts, |
| (rdevlin.cronin@chromium.org 2014-06-23 21:44:25 +0000) | 104 const std::vector<UserScript*>& scripts) { |
| (hanxi2015-03-11 16:40:06 -0700) | 105 // If the host causing this injection changed, then this injection |
| (rdevlin.cronin@chromium.org 2014-06-23 21:44:25 +0000) | 106 // will be removed, and there's no guarantee the backing script still exists. |
| (Rob Wu 2016-07-16 02:00:27 -0700) | 107 if (changed_hosts.count(host_id_) > 0) { |
| (Rob Wu 2016-07-16 02:00:27 -0700) | 108 script_ = nullptr; |
| (rdevlin.cronin@chromium.org 2014-06-23 21:44:25 +0000) | 109 return; |
| (Rob Wu 2016-07-16 02:00:27 -0700) | 110 } |
| **Vulnerability fixed in Version 52.0.2743.82** | |

Figure 28: Inaccuracies in NVD Database CVE-2016-5136

The labeling for NVD database is not accurate according to the three examples we mentioned before. The number of vulnerabilities we could have for the Chrome versions is an imperfect data set. And the inconsistent interpretation of the data introduced more noise in the data set. To reduce the maximum of the noise in our data set, we reinterpret the keyword *before* or *prior to* in the same way across all the versions. According to the Google Chrome version history Wikipedia page [6], Google discontinues the versions three months after their release date. We assume the discovered vulnerability only affects the versions released within a year. We adjust the affect window to half a year, one year and one year and half in our study.

[6] https://en.wikipedia.org/wiki/Google_Chrome_version_history

### 5.3.2 Data Preparation and Feature Extraction

During the data collection procedure, we collect the datasets for Chrome and GitHub projects. However, the entries in the datasets are not all validated. In this section, we introduce the data processing techniques we applied before the feature selection and the learning models.

#### Incompleteness Data

In the Chrome dataset, the earliest version we could obtain is the version *3.0.195.25*. We obtain the date for the versions before 2014 April 07 from the SVN repository since the merge to the Git repository loses the release date for those versions, but we were not able to check out the versions before 3.0.195.25, since these repositories were not applied the merge to the Git repository. There are 22 versions we could not download since the data are not available in the repository anymore, which is 0.19% in our Chrome dataset. In the GitHub dataset, 5 projects are empty after check out; slocs for these projects are 0 that are eliminated from the dataset. Flawfinder crashes in 16 projects during the data collection. Those projects are eliminated as long as the Flawfinder results are considered as the selected features in the later study. Since the Flawfinder only works on the project relates with C/C++ languages, the projects generated with other language is marked as 0 in the original entries. We adjust the entry to -1 to ensure the difference between the projects without any discovered flaws and the project without C/C++ language.

#### Noise Data

In the original Chrome dataset we obtain in this study, we manually analyze the abnormal data entries. Some versions of Chrome had been mis-committed twice in the updating history of the Chrome git repository, which lead to the unexpected jump of SLOC among the rest of the versions. We check all the unusual versions, and verify the mishandling in the

repository. See the detailed version numbers in Section . Two versions of the Chrome has mis-labeled release dates and we discover that release date on the Google blog is different than the one mentioned on the SVN repository (*4.0.249.76* and *6.0.472.47*). We also adjust the release date to the date listing on the Google blog. In GitHub dataset, the expert selection contains the noise, which we can not eliminate in this study.

### Inconsistency Data

The inconsistency data from the Chrome dataset is the number of CVEs and the way in which they interpret the keywords. We adjust this issue by assuming the affected windows for versions as 0.5 year, 1 year and 1.5 year. In GitHub project, we apply expert selection to reduce the inconsistency link and citations of the projects.

### Feature Extraction

Table 22 in Appendix demonstrates the sample data from the Chrome data set; we only give the number of CVEs based on the 0.5 year model as an example here. Table 25 shows the meaning for the corresponding features. Formally, we define the mapping function $F$ from features to numeric values. X is the set of all the features and x $\in$ X is one of the features in the features set X. The function $F$ returns numeric value to be used in the later feature selection and classification.

$$F(X) = \begin{cases} x & x \in numeric\ features \\ days & x \in time\ features \\ numbers & x \in label\ features \end{cases}$$

The time features include the release time and the first published CVE date in Chrome data set. We use the date when we obtained the dataset (15-11-2017) as the current time to calculate the age for the versions of the Chrome. The age represents the attack period

for the attackers for this version. Label feature is the release day for each version. We create the mapping set [Monday=1,Tuesday=2, Wednesday = 3, Thursday = 4, Friday = 5, Saturday = 6, Sunday = 7] to create numbers for this feature. The name of CVE has been deleted in this study since it does not provide any intrinsic information for the software application. We still show this feature in Table 22 as the completeness of the data set.

In the GitHub project, the time features are the release date and the last push time. The release date is transformed as the age of the Git project by using the same method as mentioned in Chrome dataset. The last push time is transformed as the last history active date by calculating the date between current time to the last push time. We delete two features: *is fork* and the *number of flaw in severity level 0*, which are all 0 for the projects we collected. The features contain the same value, for all the data entries do not make any difference to distinct the data.

## 5.4 Feature Selection of Software Vulnerability Model

In this section we use machine learning technologies to evaluate the effectiveness of the features we collected to the defined target (the number of CVE vulnerabilities) in GitHub dataset. Before we continue, we uniform the terms in the latter sections. We refer the target variable in regression models as *response* and as *class* in classification models. The features in regression models are referred as *predictors* and as *predictive variables* in classification models. The data entries are referred as instances in both models.

### 5.4.1 Data Visualization

To understand the full picture of our dataset, we choose the t-Distributed Stochastic Neighbor Embedding (t-SNE) [66] technique in order to reduce the dimensions of our datasets and to visualize the data into 2D graph. t-SNE represents the similarities of high-dimensional

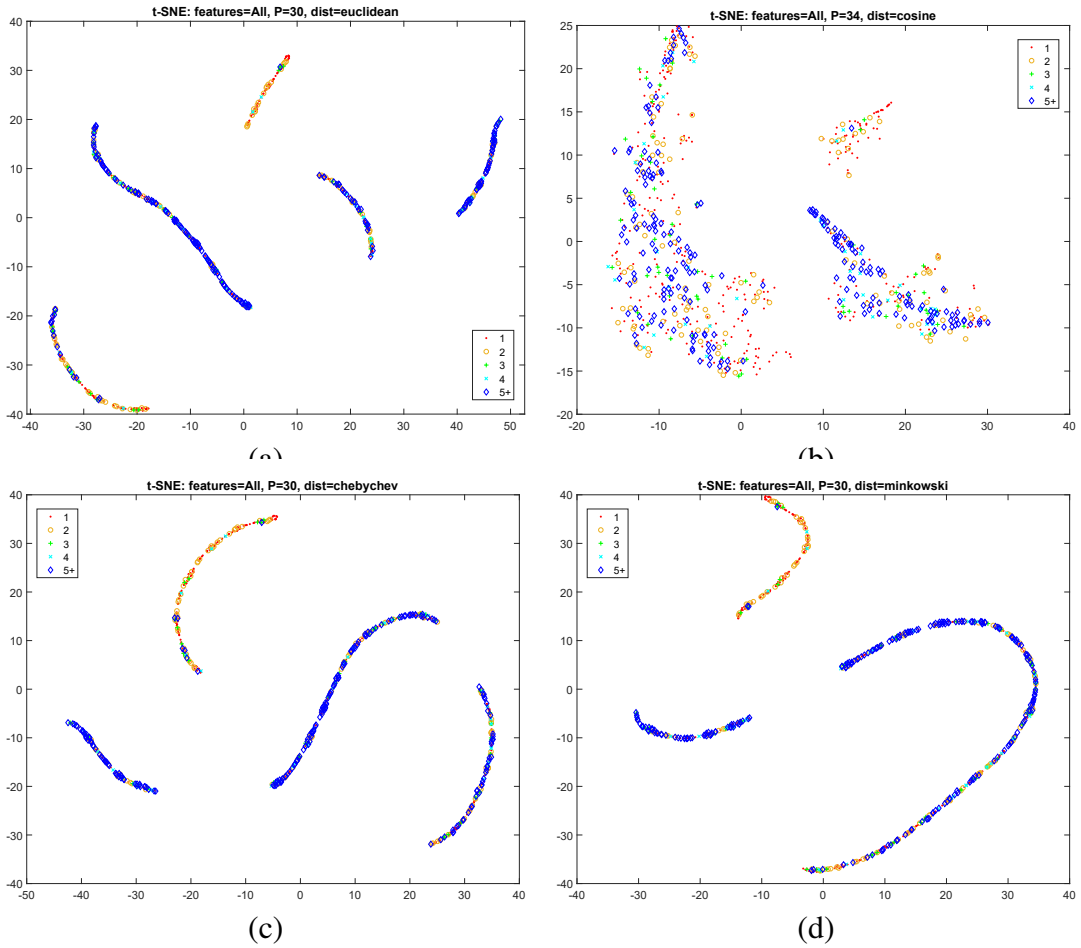



Figure 29: The Data Visualization for GitHub with t-SNE (a)(perplexity = 30, Algorithm = Euclidean),(b)(perplexity = 34, Algorithm = Cosine),(c)(perplexity = 30, Algorithm = Chebychev),(d)(perplexity = 30, Algorithm = Minkowski)

data-points as the conditional probabilities which are calculated by the algorithms, e.g., the default algorithm is Euclidean distance algorithm. Compared to the traditional dimensionality reduction techniques, e.g., Principal Components Analysis (PCA) [2], which uses linear techniques, t-SNE performs better in keeping the similar datapoints close together with nonlinear dimensionality reduction techniques, which is not easily achieved by a linear mapping. Compared with other nonlinear dimensionality reduction techniques, such as Sammon mapping [109], Stochastic Neighbor Embedding (SNE) [40], t-SNE is capable of capturing both the local structure and the global structure of the datasets.

In matlab software, 11 distance algorithms can be chosen in the t-SNE function (the default the distance algorithm is Euclidean distance). The parameter ”Perplexity” controls the effective number of local neighbors at each point (the default value is 30). Figure 29 is the visualization of GitHub dataset with the default distance algorithm and the default perplexity value. t-SNE maps the 34 predictors into 2 dimensional coordinate, then we use the number of CVEs to color the 2 dimension datapoints. The blue dots are defined as the instances that correspond to 5 or more CVEs. The other colors correspond with the exact number of the CVEs. In Figure 29 (a), the datasets are divided into five clusters; two of the clusters are majority with the lower number of CVEs and three of them are majority with higher CVEs.

We apply all the distance algorithms with various perplexity values in order to obtain the best visualization results. Figure 29 (b),(c),(d) are the best visualization results from GitHub dataset. In the visualization process, we remove the response, the number of the CVE vulnerabilities, from the dataset; only the predictors are clustered with t-SNE algorithm. After the unsupervised clustering, we use the number of the CVEs to color the datapoints on the lower-dimensional graph. In total, we have 55 distinct numbers of CVEs in our dataset, which makes it impossible to put them all on the graph. The entire dataset has been separated into 5 groups based on the number of instances. The group 1,2,3,4 means the instance contains corresponding number of CVE(s); group 5 contains the instances with 5 or higher number of CVEs.

The shapes of the visualization results change with the distance algorithms and the ”Perplexity” values; the common observation from the visualizations is the instances with lower number CVEs are different from the instances where higher number of CVEs (the blue color on the graph).

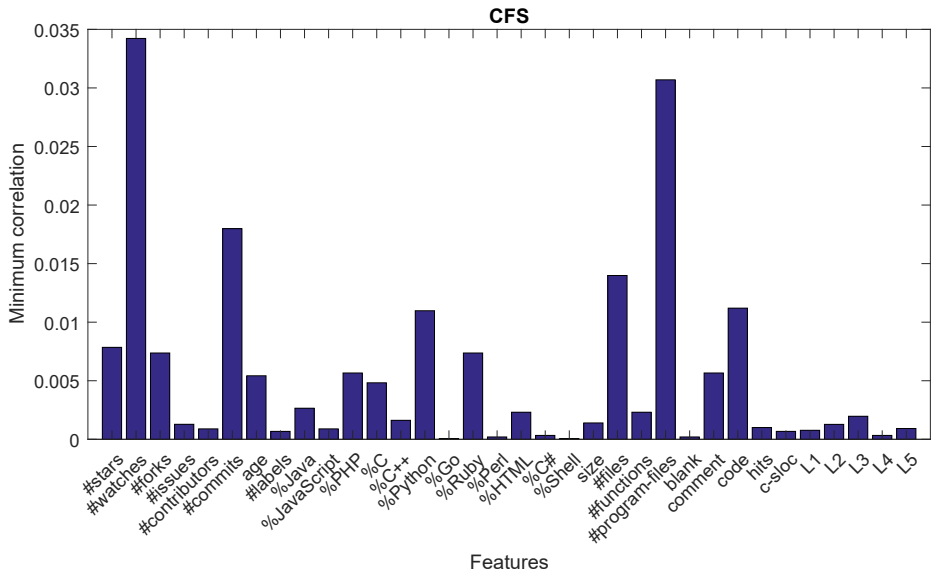



Figure 30: The Feature Selection from CFS

### 5.4.2 Feature Selection

We apply filter, wrapper and embedded feature selection methods on our GitHub dataset respectively, in this section.

#### Filter Methods

In filter method, we choose correlation-based feature selection (CFS) [38], mutual information (MI) [11] and RELIEF [58], due to the fact that these methods are wildly used in the literatures.

**Correlation based Feature Selection** CFS calculates the correlations between any pair of features, and uses the lowest correlation in one feature as the weight of this feature. CFS method selects the features that highly correlate with the response and do not correlate with other features. Figure 30 is the feature selection scores from CFS. In this study, we first set 0.01 as the threshold for the CFS method, which means the weight lower than 0.01 is preselected in this step. Then we joint the results from CFS with the Table 12 correlation

column to build the feature set.

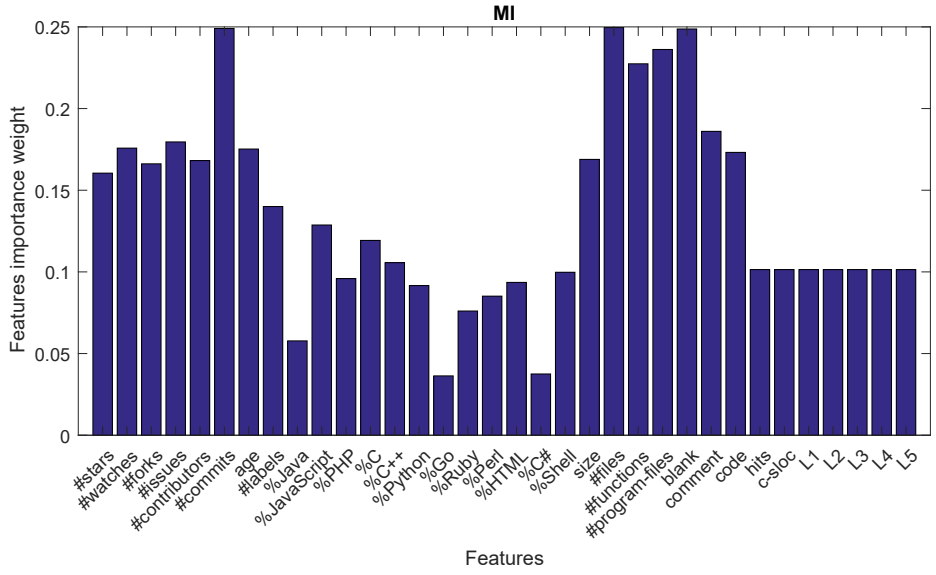



Figure 31: The Feature Selection from MI

**Mutual Information** Mutual information (MI) measures the mutual dependences between the response and the predictors. Same as the correlation, MI only produce pairwise results. Contrasted with the correlation, MI captures the nonlinear dependency through the joint probabilities. In this study, we output the MI score as the importance weight for features. Figure 31 demonstrates the importance for features with the features as x-axis and MI score as y-axis. The threshold to select the feature subset is 0.15 in this study.

**RELIEF** RELIEF algorithm calculates the Euclidean distance from each predictor to response. $k$ is the number of closest predictors, which is taken into consideration for the majority vote. Figure 32 shows the feature selection results from different number of $k$. In this study, we focus on selecting the features from the $k = 15$, threshold = 0.02.

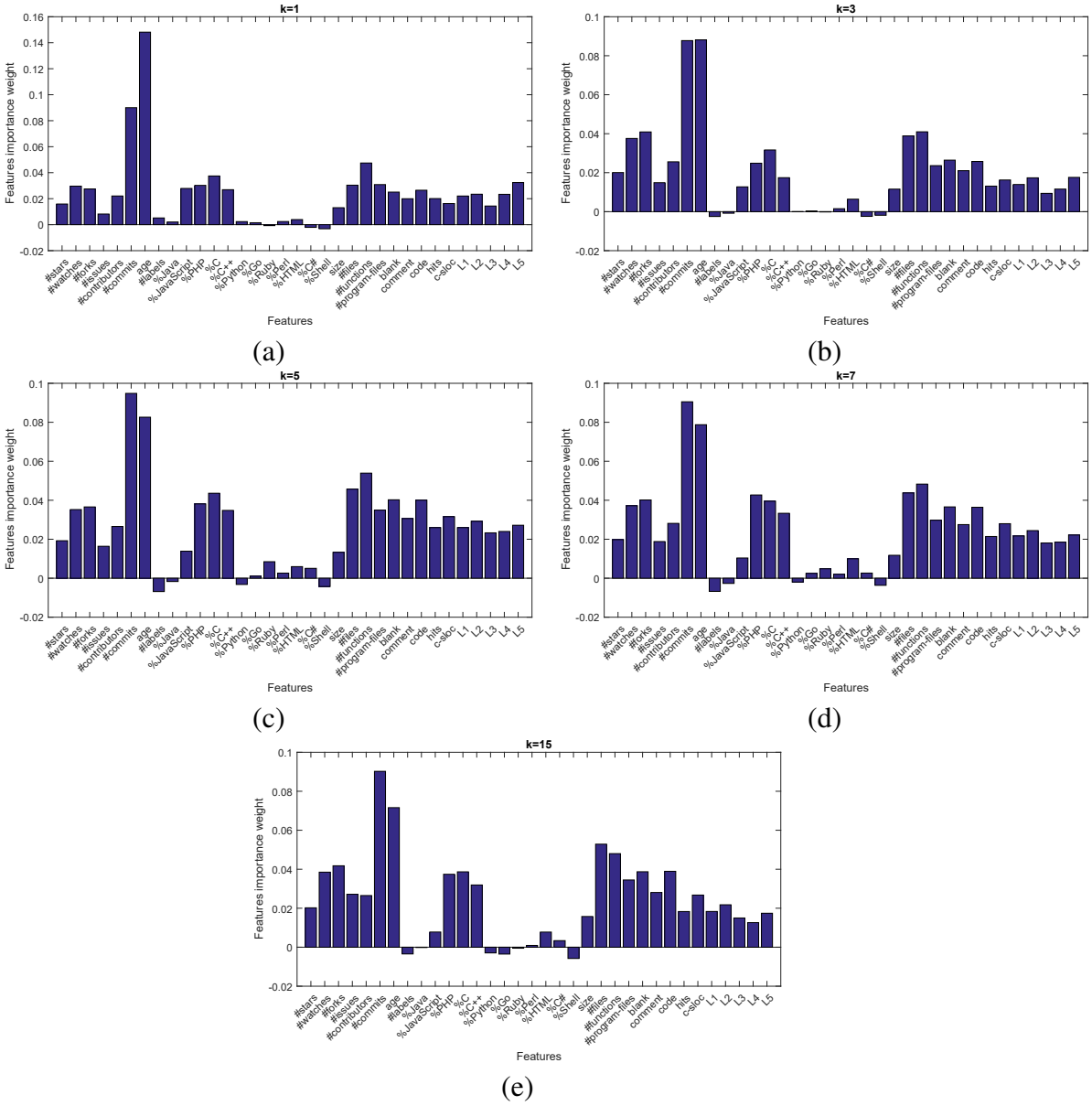



Figure 32: The Feature Selection from RELIEF with Different $k$

## Wrapper Methods

Multiple heuristic algorithms are available to build the best set in wrapper methods; in this study we apply the SFS and sequential forward floating selection (SFFS); both are the family of greedy search algorithms, with $k$-NN classifier to select the feature set. Since SFFS is an extension of the SFS, in our study, both algorithms give the same output feature set.

The SFS algorithm starts with an empty set and adds one feature, which gives maximum accuracy within each iteration. 6 features are selected in the wrapper methods.

### Embedded Methods

Three are embedded methods that are applied to our dataset: Decision tree (DT), Boosted tree (BT), and Random forest (RF). Figure 33 demonstrates the feature selection results from the DT algorithm; the results are plotted as the importance bars. We select 0.05 as our threshold to select features from the feature sets.

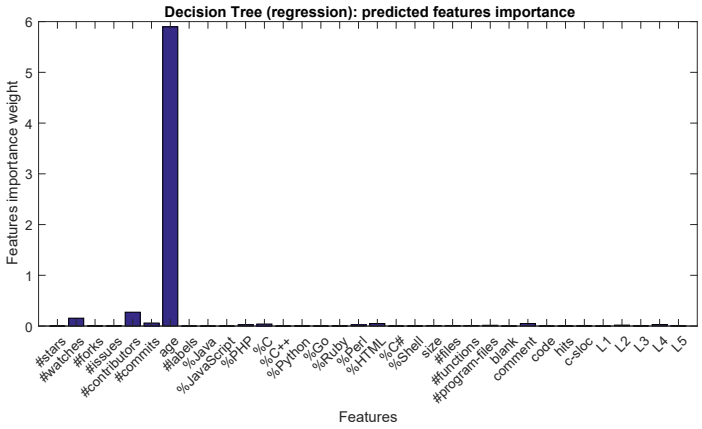



Figure 33: The Feature Selection from Decision Tree

Then we apply the boosted tree with 200, 500 and 1000 trees in our study. The results are almost identical with the different number of trees, therefore, we only show the result from 200 trees as an example here. Figure 34 shows the feature selection result from BT. In this study, we set 0.018 as the threshold to select the feature subset.

Figure 35 shows the feature selection from RF with 200, 500 and 1000 trees. The results for the important weight are slightly different with the number of trees. In this study, we select the feature subset based on the result from 1000 trees with 0.1 as the threshold.

Table 11 demonstrates the feature selection results from all the methods we mentioned in the previous section. In the latter section, the classifiers would be built on the full feature

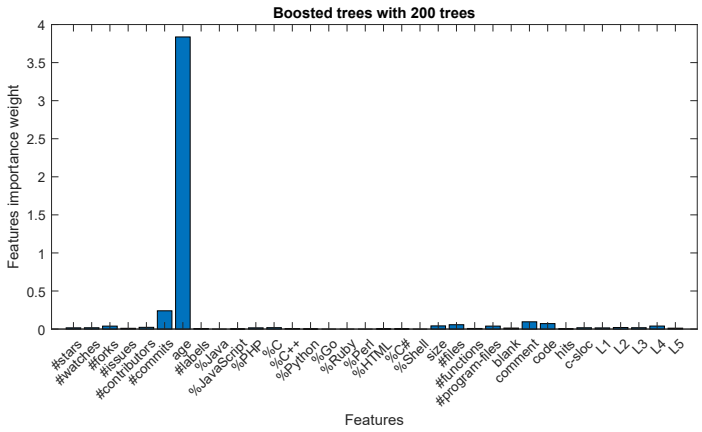



Figure 34: The Feature Selection from Boosted Tree

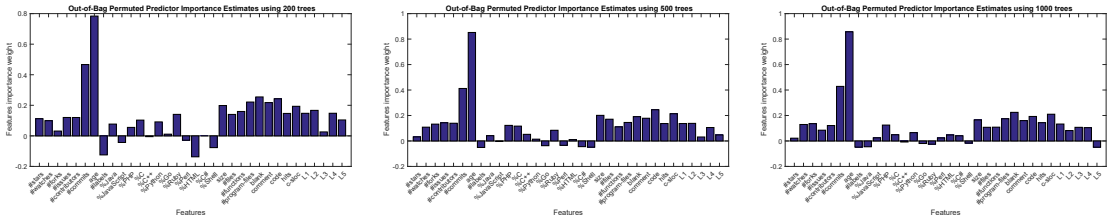



Figure 35: The Feature Selection from Random Forest

sets and the selected feature sets from various feature selection methods.

## 5.5 Analysis of Software Vulnerability Model

In this section, we first describe the hypotheses that we researched on this topic, and then we analyze the hypotheses with multiple methods.

### 5.5.1 Hypotheses

A large body of research studied the vulnerability prediction models based on the software metrics, however, most of the them tried to identify the relationships between software metrics and the vulnerabilities in the components of one software application. In our study, besides the software metrics, we gather software property metrics, security metrics and

| | | | Filter Method | | | Wrapper Method | Embedded Method | | |
|---|---|---|---|---|---|---|---|---|---|
| | | Correlation | CFS | MI | RELIEF | SFS | DT | BT | RF |
| Popularity Metrics | #stars | 0.0816 | | ✓ | ✓ | | | | |
| | #watches | 0.1420 | | ✓ | ✓ | | ✓ | | |
| | #forks | 0.1425 | ✓ | ✓ | ✓ | ✓ | | ✓ | ✓ |
| Developer Metrics | #contributors | 0.1307 | ✓ | ✓ | ✓ | | ✓ | ✓ | ✓ |
| | #commits | 0.4360 | | ✓ | ✓ | | ✓ | ✓ | ✓ |
| Software Property Metrics | age | 0.3363 | ✓ | ✓ | ✓ | | ✓ | ✓ | |
| | #labels | -0.0054 | | | | | | | |
| | %Java | -0.0358 | | | | ✓ | | | |
| | %JavaScript | -0.0202 | | | | | | | |
| | %PHP | 0.0499 | | | ✓ | | | | ✓ |
| | %C | 0.0553 | | | ✓ | | | ✓ | |
| | %C++ | 0.0319 | | | ✓ | | | | |
| | %Python | -0.0479 | | | | | | | ✓ |
| | %Go | -0.0204 | | | | | | | |
| | %Ruby | -0.0421 | | | | | | | |
| | %Perl | -0.0047 | | | | ✓ | | | |
| | %HTML | 0.0072 | | | | ✓ | | | |
| | %C# | -0.0119 | | | | | | | |
| | %Shell | -0.0279 | | | | ✓ | | | |
| Software Metrics | size | 0.1266 | ✓ | ✓ | | | | ✓ | |
| | #files | 0.2600 | | ✓ | ✓ | | | ✓ | ✓ |
| | #program-files | 0.1620 | | ✓ | ✓ | | | ✓ | ✓ |
| | blank | 0.1956 | ✓ | ✓ | ✓ | | | | ✓ |
| | comment | 0.1694 | ✓ | ✓ | ✓ | | | ✓ | ✓ |
| | code | 0.2036 | | ✓ | ✓ | | | ✓ | ✓ |
| Security Metrics | #issues | 0.0805 | | ✓ | ✓ | | | | ✓ |
| | #functions | 0.2658 | ✓ | ✓ | ✓ | | | | ✓ |
| | hits | 0.1246 | ✓ | | | | | | ✓ |
| | c-sloc | 0.1966 | ✓ | | ✓ | | | | ✓ |
| | L1 | 0.1209 | ✓ | | | | | | ✓ |
| | L2 | 0.1532 | ✓ | | ✓ | | | ✓ | ✓ |
| | L3 | 0.1001 | ✓ | | | | | | ✓ |
| | L4 | 0.0521 | | | | | | ✓ | ✓ |
| | L5 | 0.1005 | ✓ | | | ✓ | | | |

Table 11: Feature Selection with Different Algorithms

popularity metrics to study the vulnerability discovery model.

It has been accepted that the complexity of the software is the cause of the vulnerabilities [112]. However, this argument is not always valid in the vulnerability discovery process. For example, in our GitHub dataset, project *opencms-core*[7], 780638 line number of codes, which is considered to have less complexity than the project *SiberianCMS*[8], 1863840 line number of codes, has a higher number of vulnerabilities. This is mainly because project *opencms-core* was released 2331 days ago and project *SiberianCMS* was only released 601 days ago. The latter project contains lower number of vulnerabilities mainly due to having less attack windows for the attackers to access. In this case, the complexity

[7]https://github.com/alkacon/opencms-core
[8]https://github.com/Xtraball/SiberianCMS

is not the key feature, which indicates the number of CVEs in one application. Similar examples could be listed in comparing the age (the days counted from the release time until now) of the software application and the popularity (the number of stars, watches, and forks) of the applications. Based on these observations, we set up the following research hypotheses:

**H1:** *There is one feature that significantly related to the number of vulnerabilities in software applications.*

**H2:** *There is a combination of features that significantly related to the number of vulnerabilities in software applications.*

**H3:** *The features can predict the number of vulnerabilities in software applications.*

### 5.5.2 Statistical Analysis of Data

In this section, we apply two statistical methods to evaluate our features against the number of CVE vulnerabilities. The first one is Spearman's rank correlation coefficient, which illustrates the linear relationships between the response and the predictor, e.g., in our case, it is the number of CVE vulnerabilities and other features. The output result for the correlation coefficient is between -1 to 1, which corresponds to fully opposite, or fully linear correlations. The value 0 means that two sets of the data have no correlation. The second method is two-sample Kolmogorov-Smirnov test (K-S test)[9], which returns the decision, and the *p-value*, whether the response and the predictor are from the same continuous distribution. Bonferroni correction is used to deal with multiple hypothesis testings, which means our stricter significance level is 0.00029 (corresponding to a non-corrected $p \leq 0.01$ for each test).

The top three correlations are the number of the commits, the number of functions and

[9]This test calculate the distance between the distribution of two samples; the null hypothesis is that the two samples are from the same distribution

| CVE | | | | |
|---|---|---|---|---|
| | | Correlation | p-value | K-S test |
| Popularity Metrics | #stars | 0.0816 | 6.0484E-225 | Reject |
| | #watches | 0.1420 | 1.251E-157 | Reject |
| | #forks | 0.1425 | 1.6426E-187 | Reject |
| Developer Metrics | #contributors | 0.1307 | 8.7476E-121 | Reject |
| | #commits | **0.4360** | 1.0614E-288 | Reject |
| Software Property Metrics | age | **0.3363** | 0 | Reject |
| | #labels | -0.0054 | 5.6425E-132 | Reject |
| | %Java | -0.0358 | 2.7586E-282 | Reject |
| | %JavaScript | -0.0202 | 2.3643E-122 | Reject |
| | %PHP | 0.0499 | 1.9834E-162 | Reject |
| | %C | 0.0553 | 8.407E-145 | Reject |
| | %C++ | 0.0319 | 1.6595E-207 | Reject |
| | %Python | -0.0479 | 4.6039E-194 | Reject |
| | %Go | -0.0204 | 0 | Reject |
| | %Ruby | -0.0421 | 1.0515E-262 | Reject |
| | %Perl | -0.0047 | 8.6337E-266 | Reject |
| | %HTML | 0.0072 | 1.2195E-163 | Reject |
| | %C# | -0.0119 | 0 | Reject |
| | %Shell | -0.0279 | 1.2681E-171 | Reject |
| Software Metrics | size | 0.1266 | 0 | Reject |
| | #files | 0.2600 | 1.8791E-270 | Reject |
| | #program-files | 0.1620 | 4.3496E-243 | Reject |
| | blank | 0.1956 | 0 | Reject |
| | comment | 0.1694 | 1.0302E-306 | Reject |
| | code | 0.2036 | 0 | Reject |
| Security Metrics | #issues | 0.0805 | 1.2384E-183 | Reject |
| | #functions | **0.2658** | 2.3783E-182 | Reject |
| | hits | 0.1246 | 3.6434E-117 | Reject |
| | c-sloc | 0.1966 | 4.3303E-112 | Reject |
| | L1 | 0.1209 | 4.0575E-127 | Reject |
| | L2 | 0.1532 | 7.8988E-122 | Reject |
| | L3 | 0.1001 | 4.9222E-157 | Reject |
| | L4 | 0.0521 | 1.0349E-139 | Reject |
| | L5 | 0.1005 | 2.1061E-239 | Reject |

Table 12: Results of the Statistical Analysis for GitHub Dataset based on Spearman's rank correlation coefficient and K-S test. K-S test significant if $p\text{-}value \leq 0.00029$

the number of files corresponding to the developer metric, software metric and security metric. However, in K-S test, all the features are not in the same distribution of the number of CVEs. As the conclusion for the **H1**, the number of commits is the best feature that correlates to the number of CVEs, however, none of the collected features are in the same distribution as the number of CVEs. Software property metrics have the overall lowest correlation with the response; other categories of features share some correlations with it.

### 5.5.3 Learning Based Model

In this section, we apply multiple classifiers to evaluate the correlation between feature sets and number of CVEs. To address the **H2**, we evaluate the performance of the classifiers in order to assess the predictive power of the features.

The first classifier sets we use in our experiment are BT and BT-opt (B-opt chooses to calculate parameters with the inbuilt algorithms). MSE, RMS, MAE, and MAPE are calculated to evaluate the performance of the classifiers. However, the 4 performance evaluators fail to demonstrate the general picture of the predicted results; correlation has been applied to indicate the relationships between the predicted values and the original results.

| Model | Performance Measures | All Features | Filter Method | | | Wrapper Method | Embedded Method | | | PCA |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | CFS | MI | RELIEF | SFS | **DT** | BT | RF | |
| BT | MSE | 274.63 | 296.81 | 274.63 | 274.63 | 1006.1 | **250.5** | 274.63 | 967.14 | 663.85 |
| | RMS | 16.57 | 17.23 | 16.57 | 16.57 | 31.72 | **15.83** | 16.57 | 31.1 | 25.77 |
| | MAE | 6.57 | 6.99 | 6.57 | 6.57 | 9.07 | **6.37** | 6.57 | 9.87 | 9.35 |
| | MAPE (%) | 206.91 | 218.71 | 206.91 | 206.91 | 263.69 | **198.18** | 206.91 | 291.24 | 291.51 |
| BT-opt | MSE | 338.48 | 485.76 | 419.9 | 366.65 | 985.68 | 387.01 | 387.3 | 850.53 | 801.39 |
| | RMS | 18.4 | 22.04 | 20.49 | 19.15 | 31.4 | 19.67 | 19.68 | 29.16 | 28.31 |
| | MAE | 6.92 | 7.76 | 7.61 | 6.78 | 9.19 | 6.96 | 6.86 | 8.47 | 9.81 |
| | MAPE (%) | 188.99 | 219.41 | 209.62 | 208.08 | 280.12 | 211.17 | 195.17 | 219.33 | 347.25 |

Table 13: Result of Predictive Power Test for BT and BT-opt Classifier

Although 4 indicators could not reflect the individual prediction results, the values could compare among the classifiers with various feature sets. Usually, a lower number of the MSE, RMS, MAE, and MAPE correlate to the better performance of the classifier/feature set. Table 37 shows that the best result from DT feature set with BT algorithm.

Figure 36 demonstrates the predicted value from this combination. In our dataset, multiple observations correspond to the same number of CVEs; therefore, we group the observations based on the number of CVEs. X-axis in the graph gives the unique number of the real CVEs from our dataset. The red squares in the upper graph plot the real CVEs for each unique CVE, which are the same as the labels on the x-axis. The green crosses are the average predicated CVEs for each unique CVE and the blue pluses are the predication CVEs for each observation.

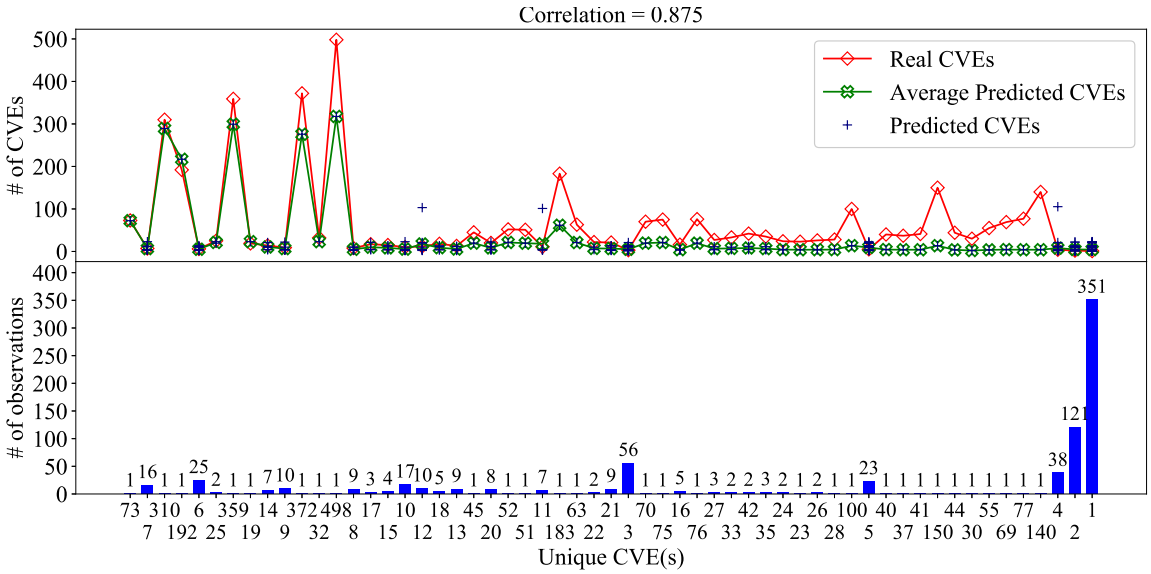



Figure 36: The Prediction Results from BT with DT Feature Set

*Results and Implications:* The results of Figure 36 are ordered by the relative percentage error ($\frac{|average\ predicted\ CVEs - real\ CVEs|}{real\ CVEs}$), which means the observation with CVEs equal to 73 has the smallest relative percentage error and the observation with CVEs equal to 1 has the largest relative percentage error. The correlation value between the real CVE(s) and the predicted CVE(s) is 0.875 in this experiment, which indicates that predicted values have similar trend with the real CVEs. This conclusion can be observed in the upper figure of Figure 36. The predicted values do not show good trend when CVEs equal to 40, 37, 41, 150, 44, 30, 55, 69, 77, 140 (the data-points between the CVEs equal to 23 and CVEs equal to 4), mainly because we lack of observations in those CVEs (all of them contain only one observation).

Visually, it is impossible to evaluate the results for CVEs equal to 1, 2, 4 (the last three values in Figure 36); we generate the accuracy for those CVEs in Figure 37. Relative percentage error is exaggerated within the small number observations since the small difference already generates a large percentage error. For example, the relative percentage is 100 when the predicted number is 2 and the real number is 1. To better understand the

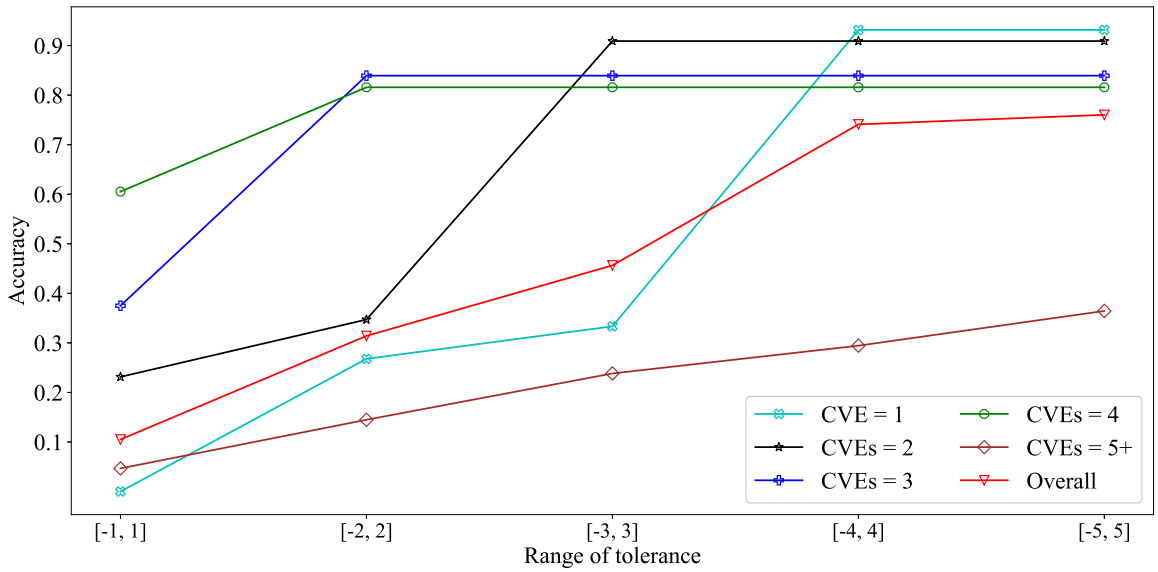



Figure 37: The Accuracy for Each CVEs from BT with DT Feature Set with Different Range of Tolerance

small number of CVEs, which are the majority of our dataset, we use the absolute range of the tolerance to calculate the accuracy. The x-axis of Figure 37 is the range of tolerance, for example, [-1,1] means that when the absolute difference between the predicted value and the real CVEs is less or equal to 1, then we consider the prediction could be accepted. The lines on the figure shows the change of the accuracy with the increase of the tolerance ranges. When the range of tolerance is [-4,4], accuracy reaches more than 75% except the line for CVEs higher or equal to 5. Notice that the accuracy calculated in this way is too strict with the high number of CVEs. For example, the prediction of the observation with CVEs equal to 310 is 289; this result is 6% away from the real CVEs but it does not fall into any tolerance range we define in Figure 37. Therefore, it is normal to have relatively low accuracy within high number of CVEs.

Table 14 demonstrates the performance measures for the DT regression (DTr) classifier with feature sets. BT feature set from the embedded method has best MSE, RMS, MAE and MAPE among all the other results. We present the predicted results and the accuracy

| Model | Performance Measures | All Features | Filter Method | | | Wrapper Method | Embedded Method | | | PCA |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | CFS | MI | RELIEF | SFS | DT | **BT** | RF | |
| DTr | MSE | 426.15 | 379.83 | 300.96 | 295.69 | 981.9 | 425.51 | **284.96** | 838.25 | 896.14 |
| | RMS | 20.64 | 19.49 | 17.35 | 17.2 | 31.34 | 20.63 | **16.88** | 28.95 | 29.94 |
| | MAE | 7.66 | 7.11 | 6.38 | 6.62 | 10.05 | 7.65 | **6.55** | 9.51 | 9.49 |
| | MAPE (%) | 284.32 | 183.17 | 182.35 | 180.72 | 363.36 | 303.12 | **179.28** | 331.89 | 239.52 |

Table 14: Result of Predictive Power Test for DT Regression Classifier

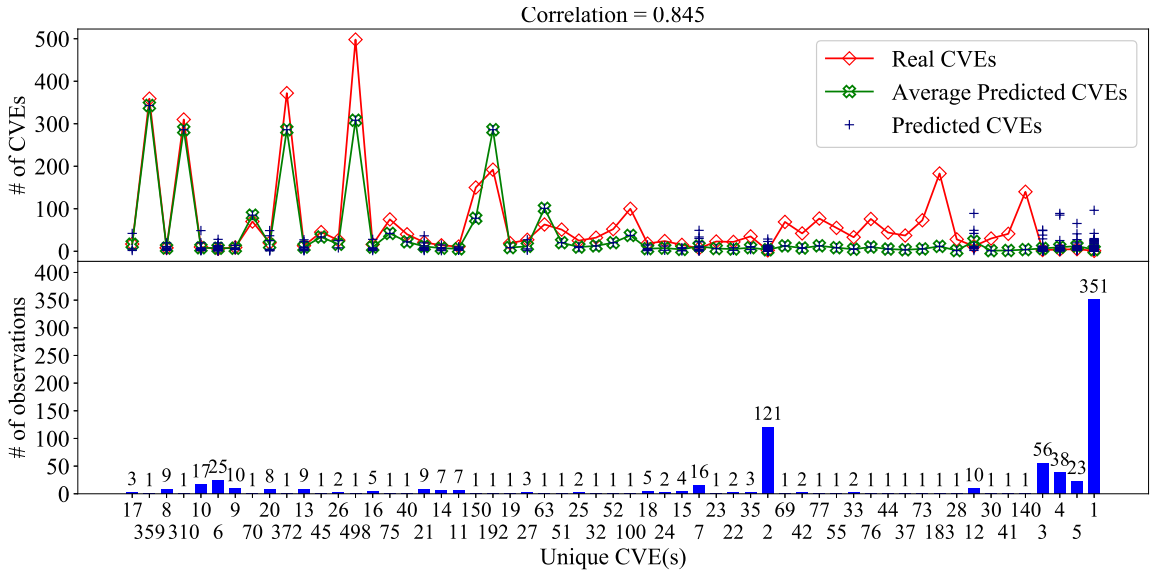



Figure 38: The Prediction Results from DT with BT Feature Set

for this combination in Figure 36 and 37.

*Results and Implications:* The performances for the DTr classifier with the BT feature set is similar to the BT classifier with the DT feature set. However, the prediction results vary between Figure 36 and Figure 38. The performance measure from DT classifier has better MAPE compare with the BT classifier which indicates the classifiers may have better prediction results. This conclusion could be verified both in accuracy Figure 39 and Figure 38. CVEs equal to 2 and CVE equal to 1 already achieve 60% of the accuracy with tolerance range of [-1,1]. We could also observe that CVEs equal to 2 has better ranking in relative percentage error from the prediction figure.

Table 15 demonstrates the performance measures for the LR classifier with feature sets. DT feature set from the embedded method has the best MSE and RMS; SFS feature set

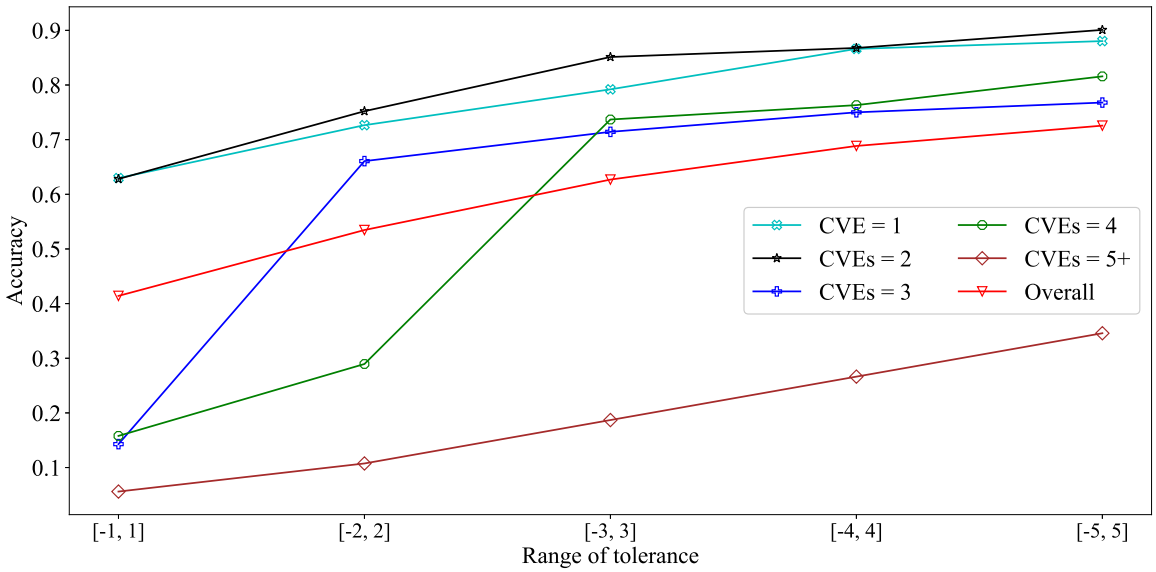



Figure 39: The Accuracy for Each CVEs from DT with BT Feature Set with Different Range of Tolerance

| Model | Performance Measures | All Features | Filter Method | | | Wrapper Method | Embedded Method | | | PCA |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | CFS | MI | RELIEF | SFS | **DT** | BT | RF | |
| LR | MSE | 966.39 | 965.53 | 896.86 | 951.48 | 1841.9 | **798.88** | 934.43 | 1007.18 | 966.39 |
| | RMS | 28.68 | 28.91 | 26.99 | 28.07 | 33.21 | **25.33** | 28.04 | 29.66 | 28.68 |
| | MAE | 13.34 | 12.71 | 12.33 | 13.07 | 10.86 | **11.41** | 12.35 | 11.46 | 13.34 |
| | MAPE (%) | 489.99 | 492.33 | 441.72 | 491.45 | 397.57 | **434.75** | 450.44 | 353.89 | 489.99 |

Table 15: Result of Predictive Power Test for LR Classifier

from wrapper method has the best MAE and MAPE. The correlations between predicted values and the real CVE are 0.451 and 0.434, respectively, we only show the predicted results and the accuracy for DT feature set in Figure 40 and 41.

*Results and Implications:* Generally speaking, the performance measures from LR classifier are worse than the previous classifiers; the best MSE is around 800 with DT feature set. Usually, a large MSE indicates the bad predictions among the large number of CVEs. This conclusion can be verified visually in Figure 40, where all the high number CVEs are moved to the middle of the figure, which means the large number of CVEs have a worse relative percentage error compared with the previous classifiers. The large number of the MAPE could be due to the bad performance in the small number of CVEs. From Figure 40,

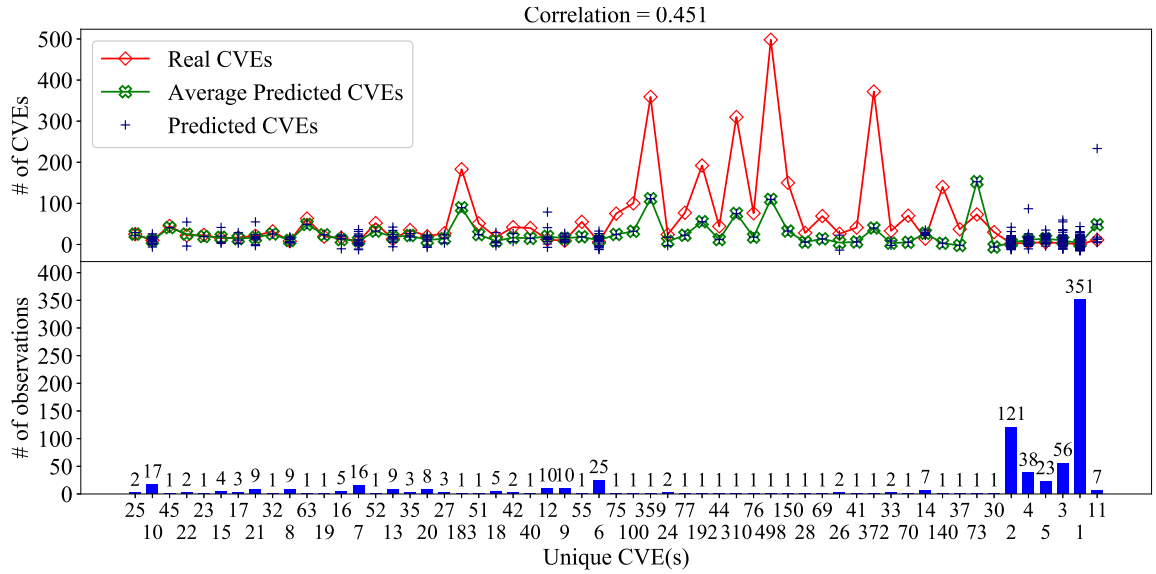



Figure 40: The Prediction Results from LR with DT Feature Set

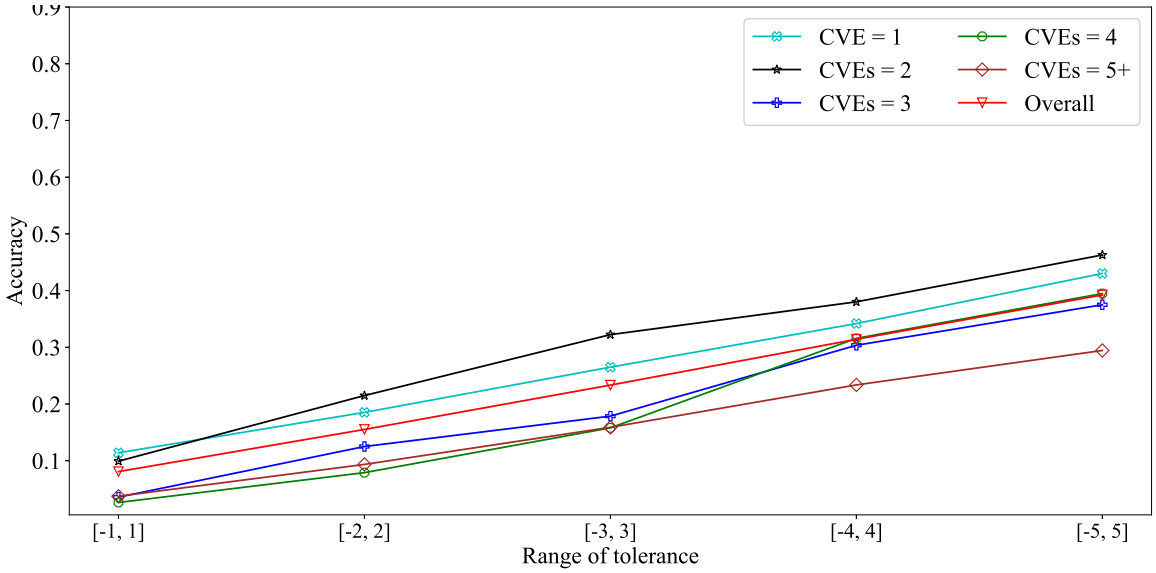



Figure 41: The Accuracy for Each CVEs from LR with DT Feature Set with Different Range of Tolerance

CVEs equal to 1, 2 ,3, 4 and 5 are in the tail of the figure, which indicates the worst relative percentage error among all the predication values. Unlike the large increase of accuracy with the increase of the tolerance range, the accuracies in Figure 41 stay relatively flatter. The best accuracy, in CVEs equal to 2 with tolerance range [-5,5], is still lower than 50%,

which is significantly worse than the previous classifiers.

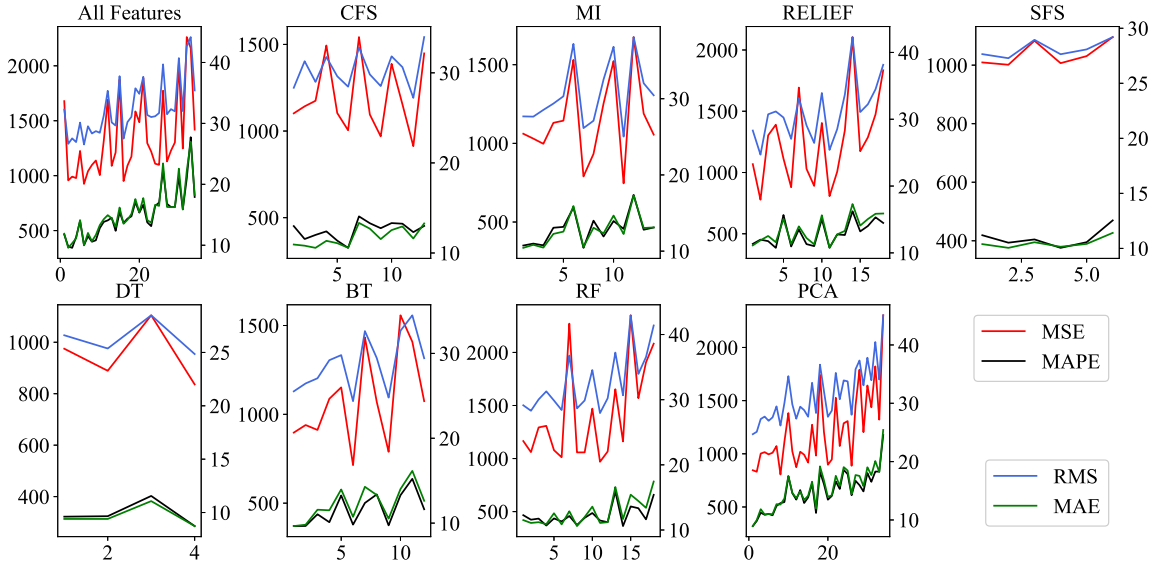



Figure 42: Function Fitting Neural Network with Different Parameters

We use two Neural Network (NN) classifiers in this study, function fitting NN and generalized regression NN, however, it is difficult to know which value to choose to obtain the best results from both classifiers. In order to get the best prediction result, we choose multiple values to feed to the parameters in the classifiers. Figure 42 and Figure 43 demonstrate the MSE, RMSE, MAE and MAPE for different feature sets with function fitting NN and generalized regression NN, respectively.

| Model | Performance Measures | All Features | Filter Method | | | Wrapper Method | Embedded Method | | | PCA |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | CFS | MI | RELIEF | SFS | **DT** | BT | RF | |
| FFNN | MSE | 956.83 | 1004.36 | 789.6 | 890.55 | 1001.31 | **836.05** | 713.99 | 969.48 | 845.47 |
| | RMS | 26.64 | 28.46 | 26.14 | 26.45 | 27.28 | **24.85** | 24.37 | 27.94 | 24.7 |
| | MAE | 9.57 | 10.58 | 10.55 | 11.28 | 10.05 | **8.73** | 10.74 | 11.08 | 8.95 |
| | MAPE (%) | 355.98 | 325.79 | 335.8 | 399.3 | 393.48 | **285.45** | 379.02 | 416.12 | 320.29 |
| GRNN | MSE | 1009.05 | 1009.05 | 1179.78 | 997.86 | 987.15 | 790.01 | 1009.05 | 986.01 | 1479.17 |
| | RMS | 28.09 | 28.09 | 31.48 | 27.23 | 27.38 | 25.94 | 28.09 | 26.97 | 32.79 |
| | MAE | 9.64 | 9.64 | 9.71 | 10.09 | 9.81 | 8.88 | 9.64 | 9.41 | 9.14 |
| | MAPE (%) | 352.83 | 352.83 | 285.15 | 403.23 | 345.11 | 290.43 | 352.83 | 341.81 | 171.48 |

Table 16: Result of Predictive Power Test for NN Classifiers

Table 16 demonstrates the performance measures (only the best results are chosen) for the NN classifiers with feature sets. BT feature set from the embedded method has best

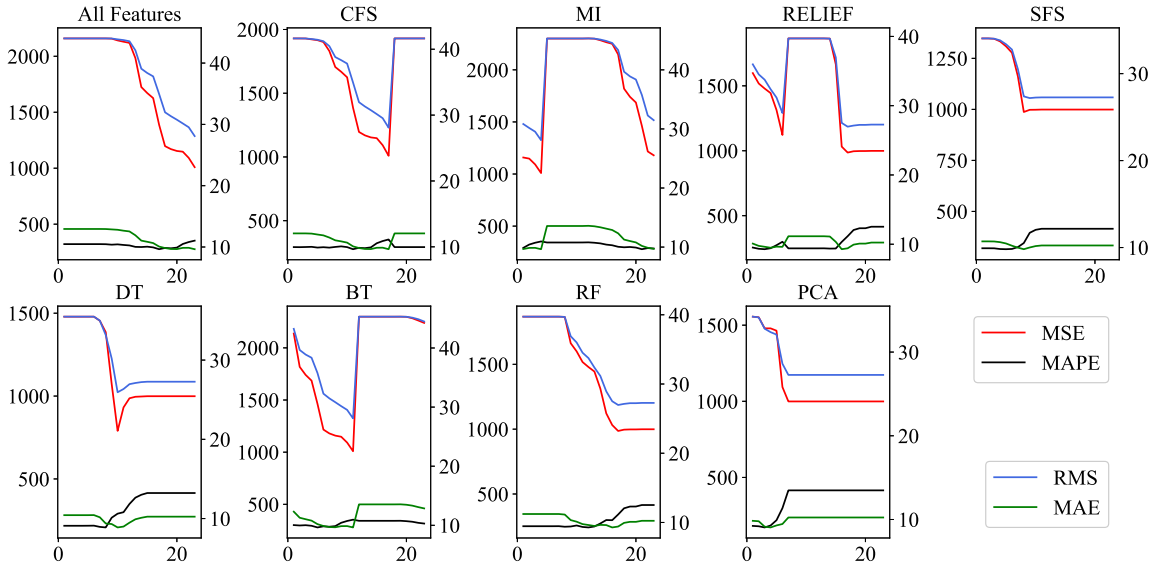



Figure 43: Generalized Regression Neural Network with Different Parameters

MSE and RMS; DT feature set from the wrapper method has the best MAE and MAPE. The correlations between predicted values and the real CVE are 0.533 and 0.406, respectively, we only show the predicted results and the accuracy for BT feature set in Figure 40 and 41.

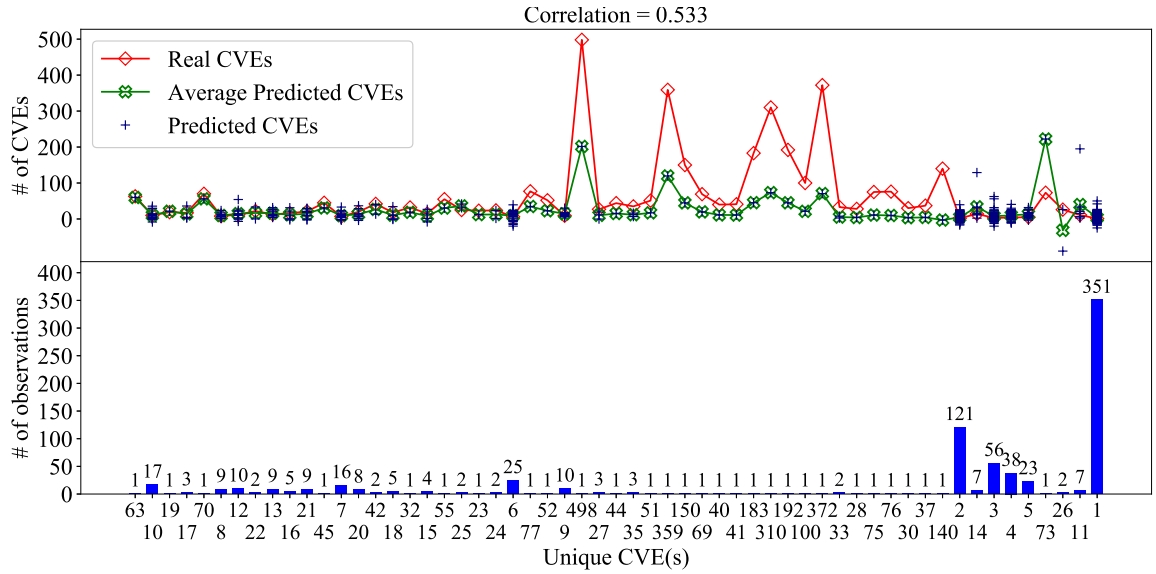



Figure 44: The Prediction Results from NN with BT Feature Set

*Results and Implications:* The general trend from NN with a BT feature set is very

similar to the result we have from LR DT feature set. Large MSE is due to the fact that the large number of CVEs are predicted with relatively worse results than the first two classifiers. The correlation in NN is better than the correlation in LR because CVEs equal to 2 has smaller relative percentage error. This conclusion could be verified in Figure 45 where the accuracy depicted by black line with the star markers is increasing faster than the result in Figure 41.

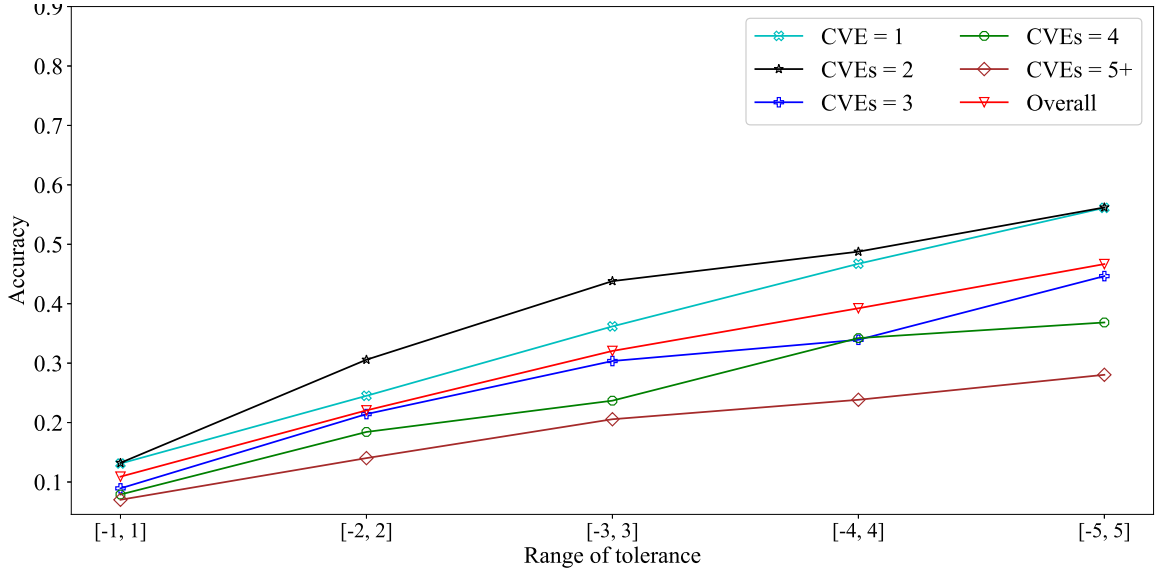



Figure 45: The Accuracy for Each CVEs from NN with BT Feature Set with Different Range of Tolerance

Table 15 demonstrates the performance measures for the RF classifier with feature sets. We choose a different number of trees for this classifier, 10, 100, 200, 500, 1000. All the results have been presented in the table; the best performance is from the DT feature set with 500 trees. As in the previous result, we show the predicted results and the accuracy in Figure 46 and 47.

*Results and Implications:* This classifier yields the prediction value as a 0.783 correlation with the real CVEs. Compared to the results from the DT and BT classifiers, Figure 46 has worse relative percentage error for large number of CVEs, which explains the large

| Model | Performance Measures | All Features | Filter Method | | | Wrapper Method | Embedded Method | | | PCA |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | CFS | MI | RELIEF | SFS | **DT** | BT | RF | |
| RF-10 | MSE | 497.96 | 535.72 | 607.58 | 597.11 | 988.1 | 435.33 | 627 | 934.21 | 826.54 |
| | RMS | 20.32 | 20.2 | 22.17 | 22.62 | 27.79 | 18.9 | 22.6 | 27.46 | 25.95 |
| | MAE | 7.37 | 7.33 | 7.97 | 7.97 | 9.94 | 7 | 7.81 | 9.31 | 8.87 |
| | MAPE (%) | 192.54 | 192.85 | 209.6 | 223.77 | 302.83 | 203.85 | 205.44 | 245.21 | 231.19 |
| RF-100 | MSE | 527.99 | 533.15 | 504.36 | 542.64 | 969.86 | 448.33 | 519.22 | 876.99 | 782.34 |
| | RMS | 20.67 | 20.7 | 20.3 | 20.67 | 27.37 | 18.8 | 20.58 | 26.87 | 25 |
| | MAE | 7.45 | 7.48 | 7.45 | 7.43 | 9.83 | 6.93 | 7.4 | 9.29 | 8.81 |
| | MAPE (%) | 198.93 | 202.63 | 211.43 | 204.7 | 305.97 | 200.6 | 195.35 | 248.45 | 258.97 |
| RF-200 | MSE | 531.61 | 537.92 | 523.94 | 534.16 | 973.33 | 442.91 | 540.11 | 868.72 | 791.32 |
| | RMS | 20.55 | 20.75 | 20.5 | 20.82 | 27.45 | 18.81 | 20.67 | 26.54 | 25.09 |
| | MAE | 7.44 | 7.48 | 7.49 | 7.49 | 9.93 | 6.93 | 7.41 | 9.14 | 8.82 |
| | MAPE (%) | 201.97 | 199.43 | 206.57 | 202.6 | 312.26 | 201.8 | 200.39 | 247.04 | 255.83 |
| RF-500 | MSE | 531.05 | 534.53 | 517.4 | 536.14 | 974.82 | **448.64** | 520.08 | 872.57 | 781.68 |
| | RMS | 20.61 | 20.5 | 20.39 | 20.68 | 27.45 | **18.71** | 20.48 | 26.71 | 25.05 |
| | MAE | 7.43 | 7.44 | 7.43 | 7.38 | 9.92 | **6.86** | 7.42 | 9.24 | 8.88 |
| | MAPE (%) | 200.47 | 201.53 | 207.14 | 200.72 | 309.44 | **199.27** | 199.45 | 249.21 | 259.93 |
| RF-1000 | MSE | 529.92 | 543.71 | 511.33 | 531.86 | 972.91 | 440.7 | 524.93 | 874.63 | 781.68 |
| | RMS | 20.56 | 20.68 | 20.23 | 20.61 | 27.41 | 18.67 | 20.53 | 26.71 | 25.05 |
| | MAE | 7.38 | 7.47 | 7.34 | 7.43 | 9.93 | 6.88 | 7.39 | 9.26 | 8.88 |
| | MAPE (%) | 197.9 | 201.38 | 204.11 | 201.58 | 311.71 | 199.01 | 199.73 | 248.31 | 259.93 |

Table 17: Result of Predictive Power Test for RF Classifiers

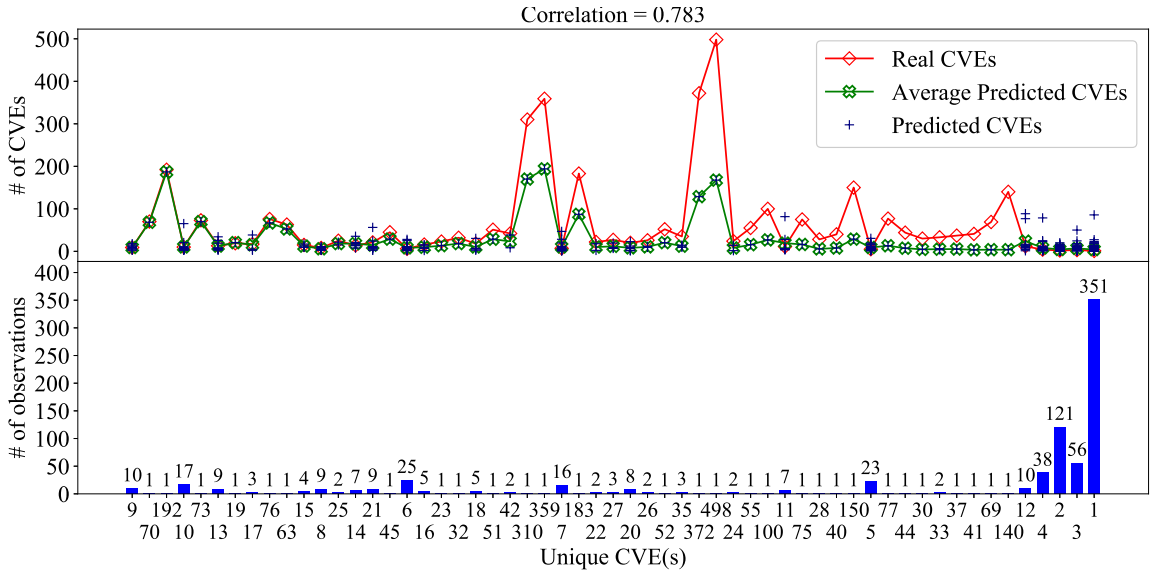



Figure 46: The Prediction Results from NN with BT Feature Set

number of MSE result from this classifier. Relatively low MAE results show the fast increase in accuracy with the increase of the tolerance range in Figure 47.

Figure 48 demonstrates the performance measures from SVM classifiers. In this experiment, we apply our dataset to three different solvers with 3 kernels. In the graph, the x-axis corresponds to the solvers; index 1-3 are for ISDA solver, index 4-6 are for the L1QP solver,

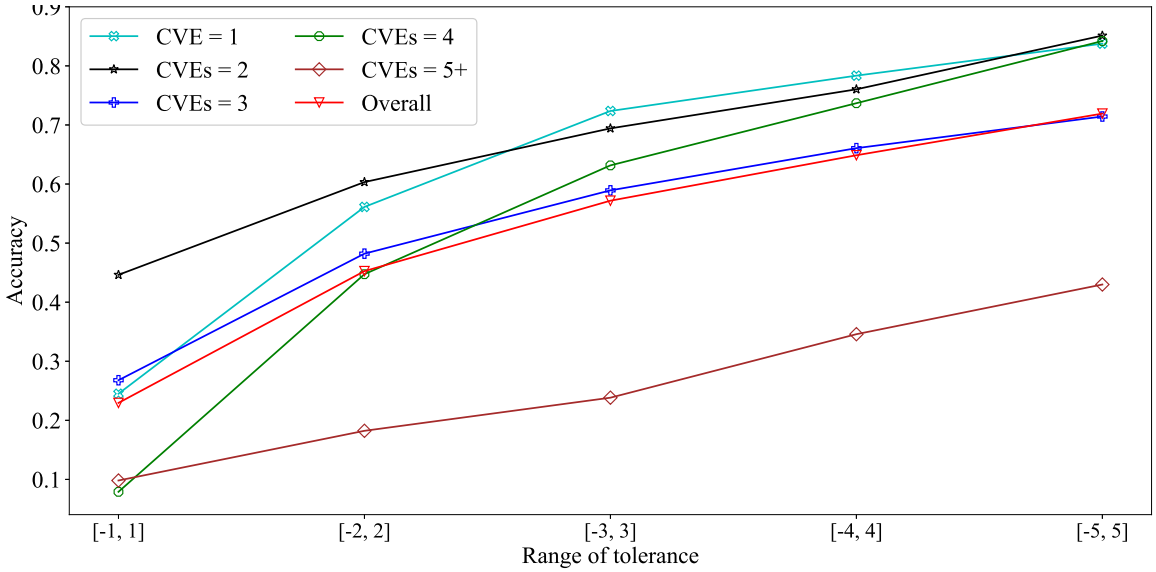



Figure 47: The Accuracy for Each CVEs from NN with BT Feature Set with Different Range of Tolerance

and index 7-9 are for the SMO solver. Inside the solver, we test it with different kernels. In the figure, the kernels are ordered as Gaussian, Linear and Polynomial; for example, index 2 is the result that uses ISDA as solver with Linear kernel. In the entire experiments, this classifier yields the best MAPE, which is 66.61% in the L1QP solver with the polynomial kernel by using the CFS feature set. We generate the predicted results corresponding to the accuracy in Figure 49 and Figure 50.

*Results and Implications:* Figure 49 demonstrates that the classifier predicts the number of CVEs to the majority of the data, which are the small number of CVEs in our case. The distribution in the lower part of the figure shows that most of the data, which has a lower relative percentage error, contains small number of CVEs. Therefore the MAPE is the lowest among all the experiments. However, we have extremely large MSE, 1021, which shows that the large number of CVEs are predicted inaccurately. Due to the fact the majority of our observations are with a low number of CVEs, the MAPE would be significantly better when the classifiers predict a low value for all the data. However, this

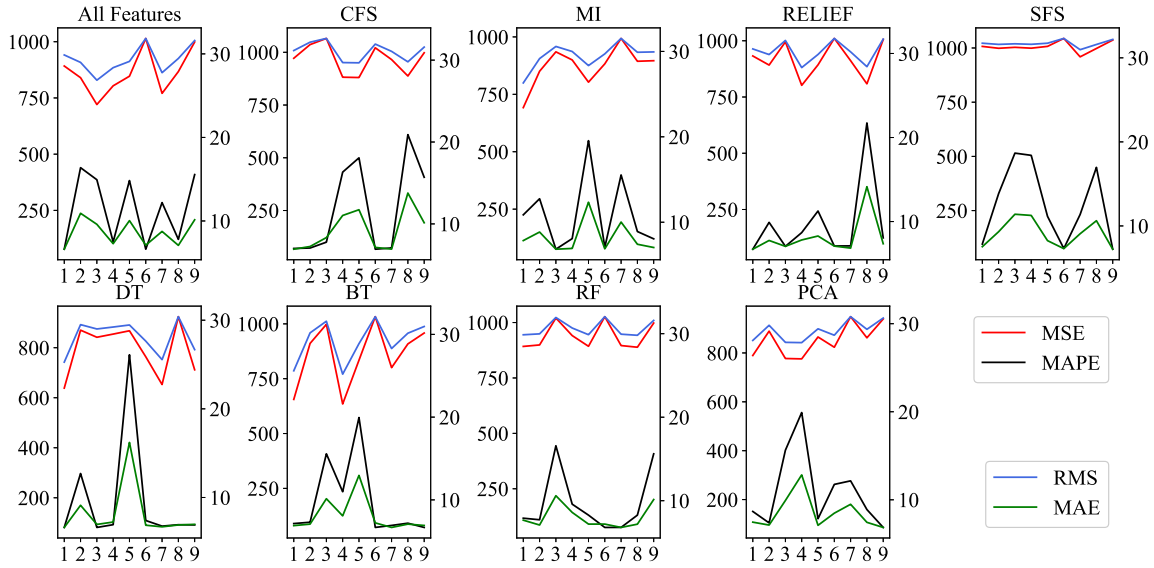



Figure 48: The Performance Measures in SVM Classifiers; index 1-3 are the results from ISDA solver, index 4-6 are the results from L1QP solver, index 7-9 are the results from SMO solver. In each solver the kernels are ordered as Gaussian, Linear and Polynomial

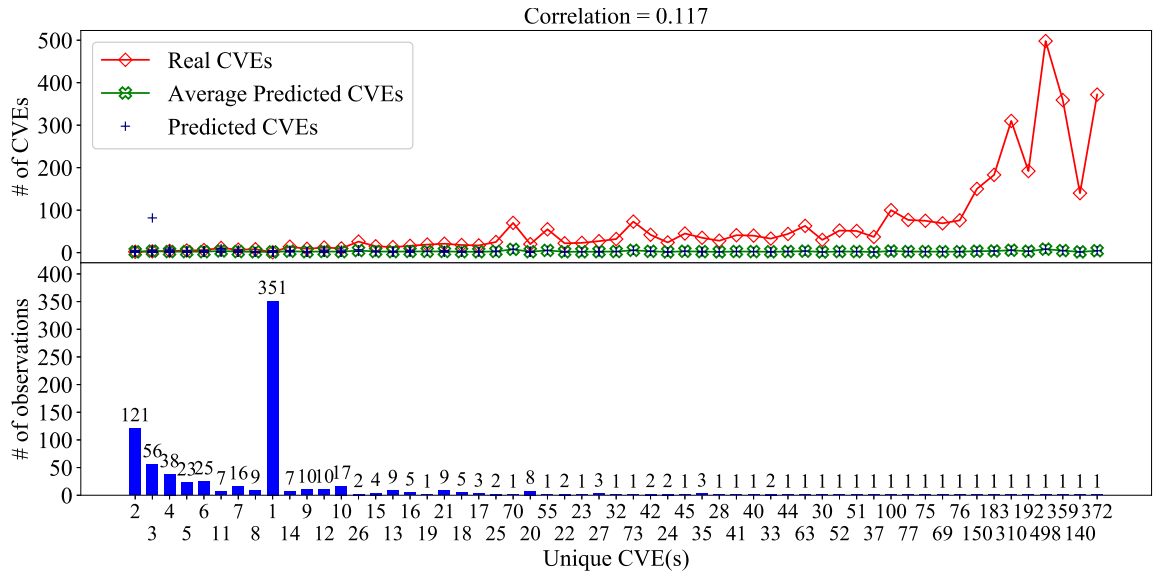



Figure 49: The Prediction Results from SVM with CFS Feature Set

classifier is not recommended because it does not have the ability to predict other number of CVEs.

Based on all the classifiers and the feature sets we study in this topic, the conclusion is

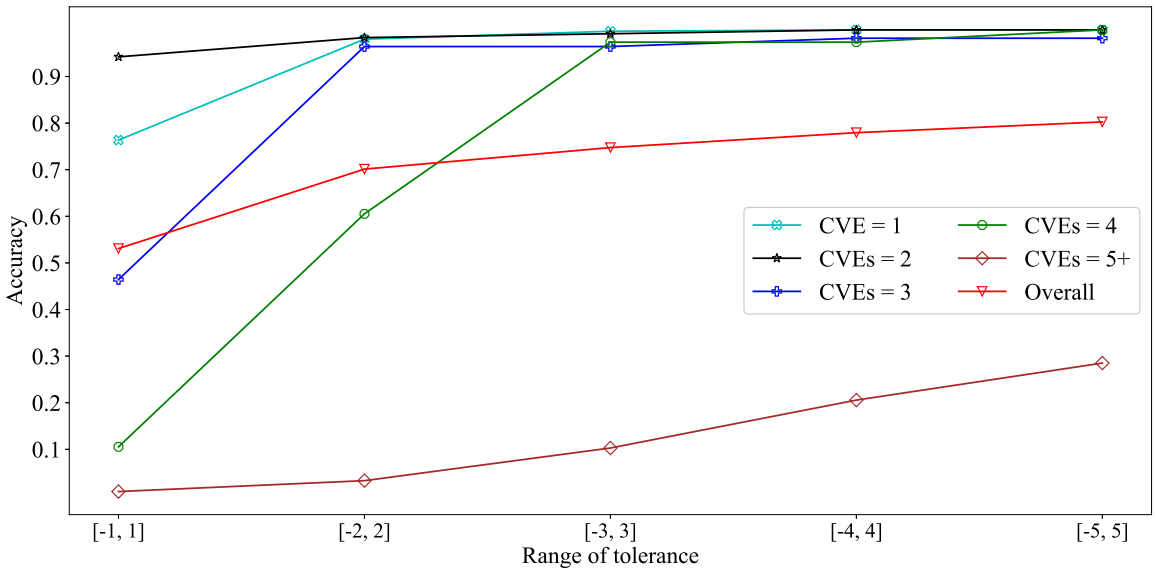



Figure 50: The Accuracy for Each CVEs from SVM with CFS Feature Set with Different Range of Tolerance

that BT classifier predicts the best CVEs with the DT feature set, and the overall accuracy is around 75% when tolerance range is [-5,5]. Also, DT and BT feature sets are only ranked as the best result from the experiments. Since the accuracy and the predicted values are both acceptable in many classifiers with different feature sets, we also accept **H2** and **H3**.

## 5.6 Analysis Software Vulnerability Model in Multi-Version Software Application

In this section we apply machine learning based model to evaluate the prediction with the features to the future number of CVEs in Chrome within certain time. Section 5.6.1 studies the **H1** in Section 5.5.1 and visualizes the dataset with the all features.

### 5.6.1 Statistical Analysis of Data and Visualization

In this section, we first study the data for the Chrome dataset based on Spearman's rank correlation coefficient and K-S test. Then, to have the general idea of our dataset, we also apply t-SNE to reduce the dimensions of the features and to visualize the data into 2D graph.

| CVE | | | | |
|---|---|---|---|---|
| | | Correlation | | K-S test |
| | | All Data | Data Before 2015 | |
| Developer Metrics | Release Date | -0.0064 | 0.003101989 | Reject |
| Software Property Metrics | age | **0.4942** | **0.537303068** | Reject |
| Software Metrics | number of files | -0.3499 | -0.673371372 | Reject |
| | numberof program files | -0.3909 | -0.5316 | Reject |
| | blank | -0.4113 | -0.4307 | Reject |
| | comment | -0.3911 | -0.393751313 | Reject |
| | code | -0.4124 | -0.425881847 | Reject |
| Combined Metric | code*age | **0.6361** | **0.446157825** | Reject |

Table 18: Results of the Statistical Analysis for Chrome Dataset based on Spearman's rank correlation coefficient and K-S test. K-S test significant if $p\text{-}value \leq 0.001$

Table 18 shows the results from the correlation and the K-S test. It is easy to notice that only age feature is positively correlated with the number of CVEs; other features are all negatively correlated. Intuitively, we relate this to the fact that we obtained the recent releases until 2017 April in our dataset which associate large number of software metrics and a relatively small number of CVEs. Therefore, the dataset only contains the older versions (the versions which have released before 2015) is applied to the correlation test. However, the software metrics are still negatively correlated with number of CVEs. Beside the correlation, we have tested the K-S test between the features and the number of CVEs. All the results have been rejected in our test. The studies from both datasets, the GitHub dataset and the Chrome dataset, have shown that we need to reject **H1**. In the features we have obtained in this study, we reject the **H1**.

As in the dataset in GitHub, we apply the data visualization technique to obtain a the bigger picture of our dataset. Figure 51 is the individualization for the Chrome dataset. In all the studies, we use 0.5 year model as the test because 3 models have shown similar

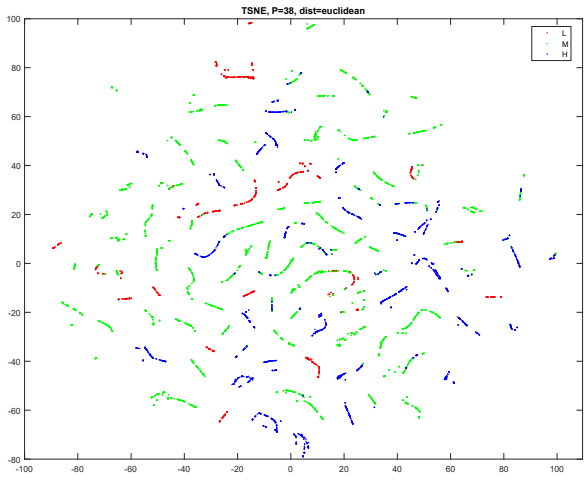



Figure 51: The Data Visualization for Chrome Dataset Based on the L,M,H Labels with Euclidean Algorithm

trends in the CVEs. To simplify the visualization process, we have labeled the observations L, M, H based on the number of CVEs. Figure 51 shows that the observations are clustered with the labels.

### 5.6.2 Learning Based Model

As opposed to the classifiers we use in Section 5.5, we apply time series, a technique used to aggregate past data points and extrapolate the relationships into the future in order to analyze the Chrome dataset. We divide our dataset into training and testing sets manually; the versions that are released before July 2014 are considered training data, and the versions that are released before the end of 2014 are considered testing data.

Figure 52 demonstrates the prediction results from the NN time series model. Blue plus markers are the outputs from training data, and green plus markers are the outputs from validation data (we set 90% percent of the training data as used for training and 10% are used for validation). The orange plus markers are the prediction outputs and the black line is the real CVEs. These results are from the open loop algorithm from the classifier, which predicts 2 versions continuously based on the history data points.

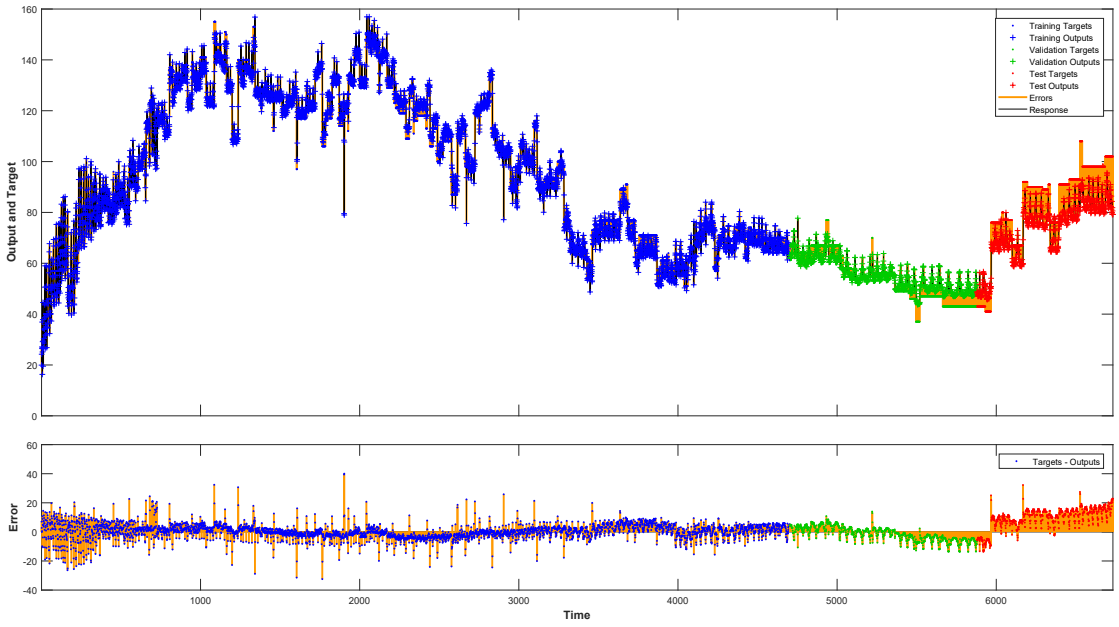



Figure 52: The Predicted Results from NN time series

*Results and Implications:* Figure 52 demonstrates that the predicted CVEs which are extrapolated from the history data points fits well with the real CVEs. The lower part of the graph gives the errors between the predicted value and the real CVEs. At the beginning of the training, large errors could be observed. The error drops with the increase of the training and validation.

Figure 53 illustrates the MSE and error distributions from the predicted results. The same conclusions could be verified from the MSE figure. The error in the training drops with the increase in the training data. Most of the predictions are in the margin of error, within a 10 to 16 difference with the real CVEs. Figure 54 shows the regression models for training, validation, testing and overall dataset. With all the R higher than 0.94 (except the validation model which is 0.89), we consider the NN time series could predict the number of CVEs for multi-version software applications.

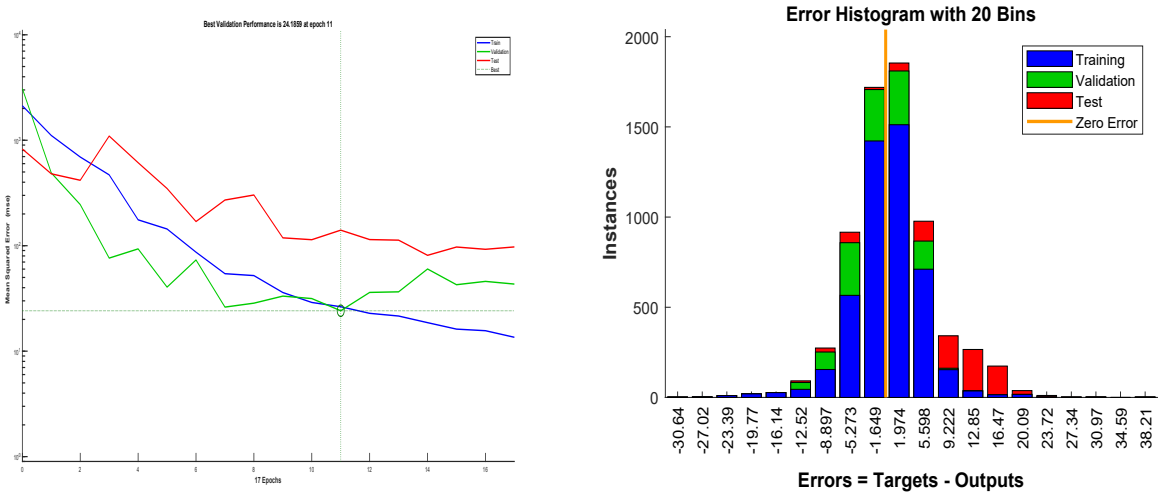



Figure 53: The MSE (Left) and the Error Distribution (Right) of predicted Results from NN time series

## 5.7 Conclusion

In this topic, we investigate the possible relationships between the software features and the number of vulnerabilities. To the best of our knowledge, this is so far the most comprehensive study with 780 software applications, including 6498 vulnerabilities in the GitHub dataset and the Chrome dataset, which includes 11454 versions and the corresponding features. Seven feature selection methods are then applied to the dataset to obtain the best feature sets for the learning models. Finally, nine feature sets (seven from feature selection, one with full features and one from feature combining algorithm (PCA)) are fed into six learning models and neural network time series in order to predict the number of CVEs for GitHub dataset and Chrome dataset, respectively. The predictive power has been evaluated though 4 performance measures. This study quantitatively demonstrates the importance of features in the vulnerability discovery process based on machine learning techniques, which provides inputs for later network level security metrics.

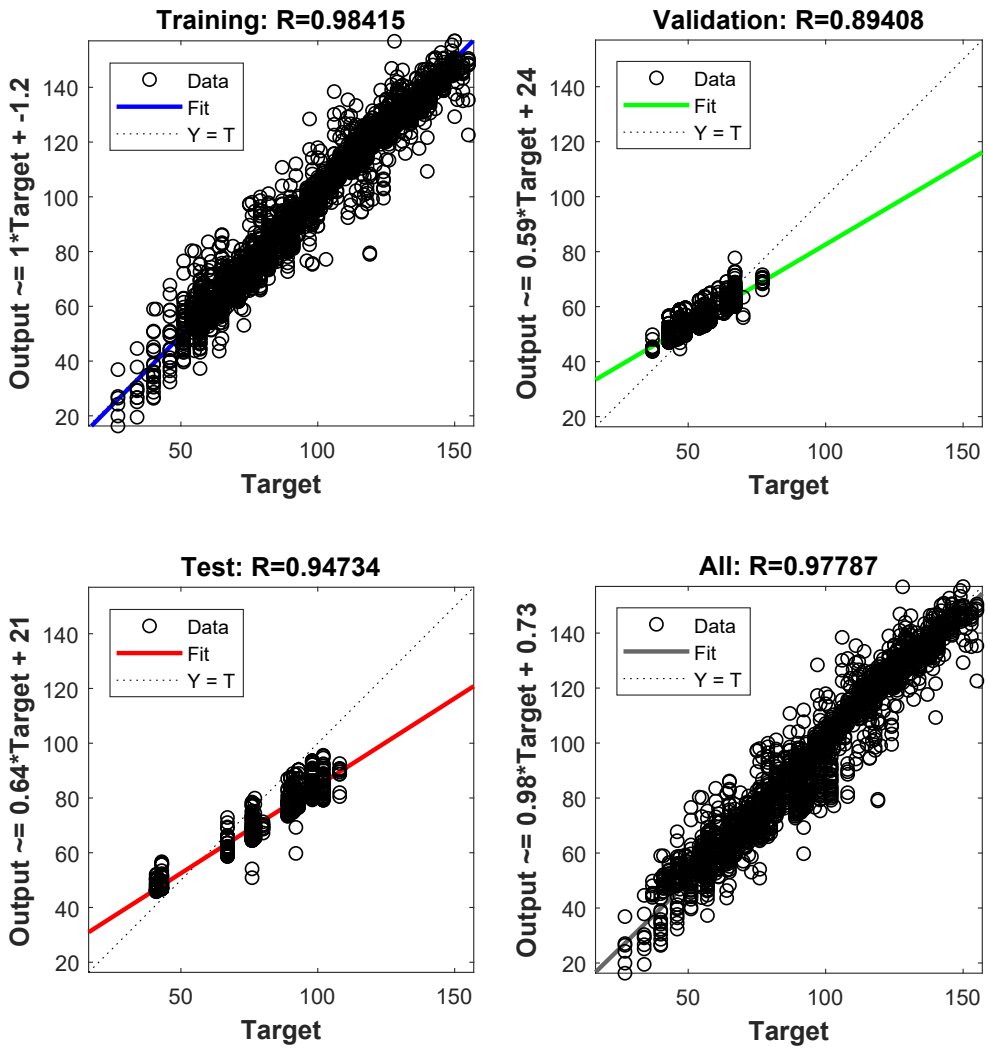



Figure 54: The Regression Model for predicted Results from NN time series

# Chapter 6

# Conclusion

Network security metrics are important to improve the security level of a network due to the fact that *"we can not improve what we cannot measure"*. My Ph.D research was among the first efforts on developing a systematic approach to network security metrics, which encompasses different abstraction levels of a network.

The three components of this thesis are related as follows. The topics are basically cover different abstraction levels of a network: at the lowest abstraction level, the empirical study (the last topic) focuses on individual software applications and services; the network diversity metrics (the first topic) study the inter-dependency between such software applications and services inside the same network; finally, at the highest abstraction level, the network attack surface (the second topic) aims to provide an indicator for the overall security of the network, taking into consideration of both software components and their diversity. Therefore, the empirical study and the network diversity would provide two complementary inputs, integrating in the network attack surface to quantify the overall network security.

# Bibliography


[1] Z. Abbadi. Security metrics what can we measure? In *Open Web Application Security Project (OWASP), Nova Chapter meeting presentation on security metrics, viewed*, volume 2, 2011.

[2] H. Abdi and L. J. Williams. Principal component analysis. *Wiley interdisciplinary reviews: computational statistics*, 2(4):433–459, 2010.

[3] M. Albanese, S. Jajodia, and S. Noel. A time-efficient approach to cost-effective network hardening using attack graphs. In *Dependable Systems and Networks (DSN'12)*, pages 1–12, 2012.

[4] O. H. Alhazmi and Y. K. Malaiya. Prediction capabilities of vulnerability discovery models. In *Reliability and Maintainability Symposium (RAMS'06)*, pages 86–91, 2006.

[5] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi. SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:61–71, 2015.

[6] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Computer and Communications Security (CCS'02)*, pages 217–224, 2002.

[7] A. Avizienis and L. Chen. On the implementation of n-version programming for software fault tolerance during execution. In *Computer Software and Applications Conference (COMPSAC'77)*, volume 77, pages 149–155, 1977.

[8] D. Balzarotti, M. Monga, and S. Sicari. Assessing the risk of using vulnerable components. In *Quality of Protection (QoP'06)*, pages 65–77, 2006.

[9] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*, pages 274–277, 2012.

[10] H. A. Basit and S. Jarzabek. Efficient token based clone detection with flexible tokenization. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT'07)*, pages 513–516, 2007.

[11] R. Battiti. Using mutual information for selecting features in supervised neural net learning. *IEEE Transactions on Neural Networks*, 5(4):537–550, 1994.

[12] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *USENIX Security Symposium*, volume 120, 2003.

[13] S. Bhatkar and R. Sekar. Data space randomization. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'08)*, pages 1–22, 2008.

[14] L. Breiman. Decision tree forest. *Machine Learning*, 45:5–32, 2001.

[15] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *Source Code Analysis and Manipulation (SCAM'10)*, pages 77–86, 2010.

[16] J. Caballero, T. Kampouris, D. Song, and J. Wang. Would diversity really increase the robustness of the routing infrastructure against software defects? In *Network and Distributed System Security (NDSS'08)*, page 40, 2008.

[17] S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, S. Savage, K. Koscher, A. Czeskis, F. Roesner, T. Kohno, et al. Comprehensive experimental analyses of automotive attack surfaces. In *USENIX Security Symposium*, 2011.

[18] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *USENIX Security Symposium*, pages 171–190, 2002.

[19] B. Chun, P. Maniatis, and S. Shenker. Diverse replication for single-machine byzantine-fault tolerance. In *USENIX Annual Technical Conference*, pages 287–292, 2008.

[20] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. *N-variant systems: A secretless framework for security through diversity*. Defense Technical Information Center, 2006.

[21] CVE Details. CVE for Ubuntu 11.04. `http://www.cvedetails.com/vulnerability-list/vendor_id-4781/product_id-20550/version_id-104819/Canonical-Ubuntu-Linux-11.04.html`.

[22] M. Dacier. Towards quantitative evaluation of computer security. Ph.D. Thesis, Institut National Polytechnique de Toulouse, 1994.

[23] A. Danial. Count lines of code (cloc). `https://github.com/AlDanial/cloc`.

[24] M. Davari and M. Zulkernine. Analysing vulnerability reproducibility for Firefox browser. In *Privacy, Security and Trust (PST'16)*, pages 674–681, 2016.

[25] M. Doyle and J. Walden. An empirical study of the evolution of php web application security. In *Security Measurements and Metrics (Metrisec'11)*, pages 11–20, 2011.

[26] T. Dullien, E. Carrera, S.-M. Eppler, and S. Porst. Automated attacker correlation for malicious code. Technical report, Ruhr-University Bochum (Germany), 2010.

[27] C. Elton. *The ecology of invasion by animals and plants*. University Of Chicago Press, Chicago, 1958.

[28] W. S. Evans, C. W. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

[29] N. Falliere, L. O. Murchu, and E. Chien. W32.stuxnet dossier. Symantec Security Response, 2011.

[30] D. Farmer and E. H. Spafford. The COPS security checker system. Technical Report 90-993, Purdue University, 1990. Computer Science Technical Reports.

[31] M. Frigault and L. Wang. Measuring network security using bayesian network-based attack graphs. In *Security, Trust, and Privacy for Software Applications (STPSA'08)*, pages 698–703, 2008.

[32] M. Frigault, L. Wang, A. Singhal, and S. Jajodia. Measuring network security using dynamic bayesian network. In *Quality of Protection (QoP'08)*, pages 23–30, 2008.

[33] K. Gaitanis and E. Cohen. Open bayes 0.1.0, 2013. `https://pypi.python.org/pypi/OpenBayes`.

[34] D. Gao, M. Reiter, and D. Song. Behavioral distance measurement using hidden markov models. In *Recent Advances in Intrusion Detection (RAID'06)*, pages 19–40, 2006.

[35] M. Garcia, A. Bessani, I. Gashi, N. Neves, and R. Obelheiro. OS diversity for intrusion tolerance: Myth or reality? In *Dependable Systems and Networks (DSN'11)*, pages 383–394, 2011.

[36] N. Gruschka and M. Jensen. Attack surfaces: A taxonomy for attacks on cloud services. In *International Conference on Cloud Computing (CLOUD'10)*, pages 276–279, 2010.

[37] G. Gu, P. Fogla, D. Dagon, W. Lee, and B. Skoric. Towards an information-theoretic framework for analyzing intrusion detection systems. In *European Symposium on Research in Computer Security (ESORICS'06)*, pages 527–546, 2006.

[38] M. A. Hall. *Correlation-based Feature Selection for Machine Learning*. PhD thesis, University of Waikato, Hamilton, New Zealand, Apr. 1999.

[39] M. Hill. Diversity and evenness: a unifying notation and its consequences. *Ecology*, 54(2):427–432, 1973.

[40] G. E. Hinton and S. T. Roweis. Stochastic neighbor embedding. In *Advances in Neural Information Processing Systems*, pages 857–864, 2003.

[41] H. Holm, M. Ekstedt, and D. Andersson. Empirical analysis of system-level vulnerability metrics through actual attacks. *IEEE Transactions on Dependable and Secure Computing*, 9(6):825–837, Nov. 2012.

[42] M. Howard, J. Pincus, and J. Wing. Measuring relative attack surfaces. In *Computer Security in the 21st Century*, pages 109–137, 2003.

[43] N. Idika and B. Bhargava. Extending attack graph-based security metrics and aggregating their application. *IEEE Transactions on Dependable and Secure Computing*, 9:75–85, 2012.

[44] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *Annual Computer Security Applications Conference (ACSAC'09)*, pages 117–126, 2009.

[45] A. Inselberg. The plane with parallel coordinates. *The visual computer*, 1(2):69–91, 1985.

[46] ISO/IEC 9899:2011 Programming languages – C. Standard, International Organization for Standardization, Geneva, Switzerland, 2011.

[47] S. Jajodia. Topological analysis of network attack vulnerability. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, page 2, 2007.

[48] S. Jajodia, A. Ghosh, V. S. Subrahmanian, V. Swarup, C. Wang, and X. Wang. *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*. Springer, 2012.

[49] S. Jajodia, A. Ghosh, V. Swarup, C. Wang, and X. Wang. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, 1st edition, 2011.

[50] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Fast, scalable malware triage. *Cylab*, 22, 2010.

[51] A. Jaquith. *Security Merics: Replacing Fear Uncertainity and Doubt*. Addison Wesley, 2007.

[52] G. Jay, J. E. Hale, R. K. Smith, D. P. Hale, N. A. Kraft, and C. Ward. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2(3):137–143, 2009.

[53] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Centre for Advanced Studies on Collaborative research*, volume 1, pages 171–183, 1993.

[54] D. Kantola, E. Chin, W. He, and D. Wagner. Reducing attack surfaces for intra-application communication in Android. In *Security and Privacy in Smartphones and Mobile Devices (SPSM'12)*, pages 69–80, 2012.

[55] G. Kc, A. Keromytis, and V. Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Computer and Communications Security (CCS'03)*, pages 272–280, 2003.

[56] N. Kheir, N. Cuppens-Boulahia, F. Cuppens, and H. Debar. A service dependency model for cost-sensitive intrusion response. In *European Symposium on Research in Computer Security (ESORICS'10)*, pages 626–642, 2010.

[57] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR'13)*, pages 329–338, 2013.

[58] K. Kira and L. A. Rendell. The feature selection problem: Traditional methods and a new algorithm. In *Artificial Intelligence (AAAI'92)*, volume 2, pages 129–134, 1992.

[59] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *Network and Distributed System Security Symposium (NDSS'13)*, 2013.

[60] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[61] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *IEEE Security & Privacy*, pages 130–143, 2001.

[62] T. Leinster and C. Cobbold. Measuring diversity: the importance of species similarity. *Ecology*, 93(3):477–489, 2012.

[63] D. J. Leversage and E. J. Byres. Estimating a system's mean time-to-compromise. *IEEE Security & Privacy*, 6(1):52–60, 2008.

[64] B. Littlewood, P. Popov, and L. Strigini. Modeling software design diversity: A review. *ACM Computer Survey*, 33(2):177–208, jun 2001.

[65] B. Littlewood and L. Strigini. Redundancy and diversity in security. In *European Symposium on Research in Computer Security (ESORICS'04)*, pages 423–438, 2004.

[66] L. v. d. Maaten and G. Hinton. Visualizing data using t-sne. *Journal of Machine Learning Research*, 9(Nov):2579–2605, 2008.

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

[68] R. Maxion. Use of diversity as a defense mechanism. In *New Security Paradigms (NSPW'05)*, pages 21–22, 2005.

[69] K. McCann. The diversity-stability debate. *Nature*, 405:228–233, 2000.

[70] J. McHugh. Quality of protection: Measuring the unmeasurable? In *Quality of Protection (QoP'06)*, pages 1–2, 2006.

[71] M. McQueen, T. McQueen, W. Boyer, and M. Chaffin. Empirical estimates and observations of 0day vulnerabilities. *Hawaii International Conference on System Sciences (HICSS'09)*, 0:1–12, 2009.

[72] M. A. McQueen, W. F. Boyer, M. A. Flynn, and G. A. Beitel. Time-to-compromise model for cyber risk reduction estimation. In *Quality of Protection (QoP'06)*, pages 49–64, 2006.

[73] P. Mell, K. Scarfone, and S. Romanosky. Common vulnerability scoring system. *IEEE Security & Privacy*, 4(6):85–89, 2006.

[74] A. Meneely and L. A. Williams. Strengthening the empirical analysis of the relationship between linus' law and software security. In *Empirical Software Engineering and Measurement (ESEM'10)*, 2010.

[75] J. Mingers. An empirical comparison of selection measures for decision-tree induction. *Machine learning*, 3(4):319–342, 1989.

[76] S. Mitra, N. Saxena, and E. McCluskey. A design diversity metric and analysis of redundant systems. *IEEE Transaction Computer*, 51(5):498–510, May 2002.

[77] J. D. Musa and K. Okumoto. A logarithmic poisson execution time model for software reliability measurement. In *Software Engineering (ICSE'84)*, pages 230–238, 1984.

[78] National Institute of Standards and Technology. Common Vulnerability Scoring System Version 2 Calculator. `https://nvd.nist.gov/cvss/v2-calculator`.

[79] National Institute of Standards and Technology. Technology assessment: Methods for measuring the level of computer security. NIST Special Publication 500-133, 1985.

[80] NetCitadel, Inc. Firewall builder, 2012. `http://www.fwbuilder.org/4.0/documentation.shtml`.

[81] NIST. National vulnerability database. https://nvd.nist.gov.

[82] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *Annual Computer Security Applications Conference (ACSAC'03)*, pages 86–95, 2003.

[83] Offensive Security. Penetration testing virtual labs. https://www.offensive-security.com/offensive-security-solutions/virtual-penetration-testing-labs/.

[84] Open hub. available at:https://www.openhub.net/, April 19, 2017.

[85] R. Ortalo, Y. Deswarte, and M. Kaâniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions Software Engineering*, 25(5):633–650, Sep/Oct 1999.

[86] X. Ou, W. Boyer, and M. McQueen. A scalable approach to attack graph generation. In *Computer and communications security (CCS'06)*, pages 336–345, 2006.

[87] J. W. P. Manadhata. Measuring a system's attack surface. Technical Report CMU-CS-04-102, 2004.

[88] J. W. P. Manadhata. An attack surface metric. Technical Report CMU-CS-05-155, 2005.

[89] J. Pamula, S. Jajodia, P. Ammann, and V. Swarup. A weakest-adversary security metric for network configuration security analysis. In *Quality of Protection (QoP'06)*, pages 31–38, 2006.

[90] S. C. Payne. A guide to security metrics. *SANS Institute Information Security Reading Room*, 2006.

[91] K. Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[92] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Computer and Communications Security (SIGSAC'15)*, pages 426–437, 2015.

[93] C. Phillips and L. Swiler. A graph-based system for network-vulnerability analysis. In *New Security Paradigms Workshop (NSPW'98)*, pages 71–79, 1998.

[94] E. Pielou. *Ecological diversity*. Wiley New York, 1975.

[95] J. Pohjalainen, O. Räsänen, and S. Kadioglu. Feature selection methods and their combinations in high-dimensional classification of speaker likability, intelligibility and personality traits. *Computer Speech & Language*, 29(1):145–171, 2015.

[96] S. Poznyakoff. Gnu cflow, 2011. `http://www.gnu.org/software/cflow/`.

[97] S. Rahimi and M. Zargham. Vulnerability scrying method for software vulnerability discovery prediction without a vulnerability database. *IEEE Transactions on Reliability*, 62(2):395–407, 2013.

[98] A. Reid, J. Lorenz, and C. A. Schmidt. *Introducing Routing And Switching In The Enterprise, CCNA Discovery Learning Guide*. Cisco Press, 2008.

[99] M. K. Reiter and S. G. Stubblebine. Toward acceptable metrics of authentication. In *IEEE Security & Privacy*, pages 10–20, 1997.

[100] M. K. Reiter and S. G. Stubblebine. Authentication metric analysis and design. *ACM Transactions on Information and System Security*, 2(2):138–158, 5 1999.

[101] K. Ren, C. Wang, and Q. Wang. Security challenges for the public cloud. *IEEE Internet Computing*, 16(1):69–73, 2012.

[102] E. Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, 2005.

[103] J. Reunanen. Overfitting in making comparisons between variable selection methods. *Journal of Machine Learning Research*, 3(Mar):1371–1382, 2003.

[104] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *IEEE Security & Privacy*, pages 156–165, 2000.

[105] J. J. C. H. Ryan and D. J. Ryan. Performance metrics for information security risk management. *IEEE Security & Privacy*, 6(5):38–44, 2008.

[106] A. Sæbjørnsen, J. Willcock, T. Panas, D. J. Quinlan, and Z. Su. Detecting code clones in binary executables. In *International Symposium on Software Testing and Analysis (ISSTA'09)*, pages 117–128, 2009.

[107] A. Saïdane, V. Nicomette, and Y. Deswarte. The design of a generic intrusion-tolerant architecture for web servers. *IEEE Transactions on Dependable and Secure Computing*, 6(1):45–58, 2009.

[108] P. Samarati. Protecting respondents' identities in microdata release. In *IEEE Transactions on Knowledge and Data Engineering (TKDE'01)*, pages 1010–1027, 2001.

[109] J. W. Sammon. A nonlinear mapping for data structure analysis. *IEEE Transactions on Computers*, 100(5):401–409, 1969.

[110] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J. Wing. Automated generation and analysis of attack graphs. In *IEEE Security & Privacy*, pages 273–284, 2002.

[111] O. Sheyner and J. M. Wing. Tools for generating and analyzing attack graphs. In *Formal Methods for Components and Objects (FMCO'03)*, pages 344–372, 2003.

[112] Y. Shin and L. Williams. Is complexity really the enemy of software security? In *Quality of protection (QoP'08)*, pages 47–50, 2008.

[113] Y. Shin and L. Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.

[114] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5, 2005.

[115] StatCounter. Desktop Operating System Market Share Worldwide. `http://gs.statcounter.com/os-market-share/desktop/worldwide`.

[116] J. Stuckman, J. Walden, and R. Scandariato. The effect of dimensionality reduction on software vulnerability prediction models. *IEEE Transactions on Reliability*, 66(1):17–37, 2017.

[117] M. Swanson, N. Bartol, J. Sabato, J. Hash, and L. Graffo. Security metrics guide for information technology systems. NIST Special Publication 800-55, 2003.

[118] L. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer attack graph generation tool. In *DARPA Information Survivability Conference & Exposition II (DISCEX'01)*, pages 307–321, 2001.

[119] Tenable, Inc. Nessus network security scanner. `http://www.tenable.com/products/nessus-vulnerability-scanner`.

[120] C. Theisen, K. Herzig, P. Morrison, B. Murphy, and L. Williams. Approximating attack surfaces with stack traces. In *Software Engineering-Volume 2 (ICSE(2)'15)*, pages 199–208, 2015.

[121] E. Totel, F. Majorczyk, and L. Mé. Cots diversity based intrusion detection and application to web servers. In *Recent Advances in Intrusion Detection (RAID'05)*, pages 43–62, 2005.

[122] D. Turner, R. Wilhelm, and W. Lemberg. Freetype. `https://www.freetype.org/`.

[123] University of Maryland. Amanda protocol. `http://wiki.zmanda.com/index.php/Developer_documentation`.

[124] J. Walden, J. Stuckman, and R. Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *Software Reliability Engineering (ISSRE'14')*, pages 23–33, 2014.

[125] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia. An attack graph-based probabilistic security metric. In *Conference on Data and Applications Security and Privacy (DBSec'08)*, pages 283–296, 2008.

[126] L. Wang, S. Jajodia, A. Singhal, P. Cheng, and S. Noel. k-zero day safety: A network security metric for measuring the risk of unknown vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 11(1):30–44, 2013.

[127] L. Wang, S. Jajodia, A. Singhal, and S. Noel. k-zero day safety: Measuring the security risk of networks against unknown attacks. In *European Symposium on Research in Computer Security (ESORICS'10)*, pages 573–587, 2010.

[128] L. Wang, S. Noel, and S. Jajodia. Minimum-cost network hardening using attack graphs. *Computer Communications*, 29(18):3812–3824, 11 2006.

[129] L. Wang, A. Singhal, and S. Jajodia. Measuring the overall security of network configurations using attack graphs. In *Conference on Data and Applications Security and Privacy (DBSec'07)*, pages 98–112, 2007.

[130] L. Wang, A. Singhal, and S. Jajodia. Toward measuring network security using attack graphs. In *Quality of Protection (QoP'07)*, pages 49–54, 2007.

[131] L. Wang, M. Zhang, S. Jajodia, A. Singhal, and M. Albanese. Modeling network diversity for evaluating the robustness of networks against zero-day attacks. In *European Symposium on Research in Computer Security (ESORICS'14)*, pages 494–511, 2014.

[132] D. Wheeler. Flawfinder home page. `http://www.dwheeler.com/flawfinder`.

[133] K. Yang, X. Jia, K. Ren, R. Xie, and L. Huang. Enabling efficient access control with dynamic policy updating for big data in the cloud. In *IEEE Conference on Computer Communications (INFOCOM'14)*, pages 2013–2021, 2014.

[134] K. Yang, X. Jia, K. Ren, B. Zhang, and R. Xie. Dac-macs: Effective data access control for multiauthority cloud storage systems. *IEEE Transactions on Information Forensics and Security*, 8(11):1790–1801, 2013.

[135] Y. Yang, S. Zhu, and G. Cao. Improving sensor network immunity under worm attacks: A software diversity approach. *Ad Hoc Networks*, 47:26–40, 2016.

[136] A. Younis, Y. Malaiya, C. Anderson, and I. Ray. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In *ACM Conference on Data and Application Security and Privacy (CODASPY'16)*, pages 97–104, 2016.

[137] A. A. Younis and Y. K. Malaiya. Relationship between attack surface and vulnerability density: A case study on apache HTTP server. In *International Conference on Internet Computing (ICOMP'12)*, pages 197–203, 2012.

[138] A. A. Younis, Y. K. Malaiya, and I. Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *High-Assurance Systems Engineering (HASE'14)*, pages 1–8, 2014.

[139] D. Zerkle and K. Levitt. Netkuang - a multi-host configuration vulnerability checker. In *USENIX Security Symposium (USENIX'96)*, 1996.

[140] M. Zhang, L. Wang, S. Jajodia, A. Singhal, and M. Albanese. Network diversity: a security metric for evaluating the resilience of networks against zero-day attacks. *IEEE Transactions on Information Forensics and Security*, 11(5):1071–1086, 2016.

[141] R. Zhuang, S. Zhang, S. A. DeLoach, X. Ou, and A. Singhal. Simulation-based approaches to studying effectiveness of moving-target network defense. In *National Symposium on Moving Target Research*, pages 1–12, 2012.

[142] T. Zimmermann, N. Nagappan, and L. A. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Software Testing, Verification and Validation (ICST'10)*, pages 421–428, 2010.

# Appendix A

# Notations

| Symbol | Page | Meaning |
|---|---|---|
| $d_1$ | p.19 | Biodiversity-Inspired Network Diversity Metric |
| $d_2$ | p.20 | Least Attacking Effort-based Network Diversity Metric |
| $d_3$ | p.25 | Probabilistic Network Diversity |
| $p$ | p.64 | Probability that software contains at least one explorable zero day vulnerability from CVSS-based method |
| $p$ | p.66 | Probability that software contains at least one explorable zero-day vulnerability from Graph-based method |
| $p(ap)$ | p.70 | Network Attack Surface |
| $\alpha$ | p.86 | Percentage of resources whose attack surface is calculated |
| $\beta$ | p.89 | Approximation rate of attack surface |

Table 19: Table of Notations

# Appendix B

# Appendices for Chapter 4

*Proof.* Assume $\alpha$ is the percent of the budget effort to calculate the attack surface in one network, and $\beta$ is the percentage to calculate in each resource. Let $n$ be the number of the resources can be calculated, and $N$ be the total number of resources in one network

$$\frac{n}{N} = \alpha + \alpha - \alpha * \beta + \alpha - \alpha * \beta - (\alpha - \alpha * \beta) * \beta ...$$

this is a geometric series when the constant ratio as (1-$\beta$) therefore we have

$$\frac{n}{N} = \frac{\alpha}{\beta}$$

□

| Index | Software |
|---|---|
| 1 | Libfm-1.2.3 |
| 2 | apcupsd-3.14.13 |
| 3 | sox-14.4.2 |
| 4 | w3m-0.5.3 |
| 5 | squashfs4.3 |
| 6 | libtirpc-1.0.1 |
| 7 | ultradefrag |
| 8 | fwbuilder-5.1.0.3599 |
| 9 | dosbox-0.74 |
| 10 | gnucash-2.6.7 |
| 11 | tcl8.6.4 |
| 12 | icinga-1.10.1 |
| 13 | fuse-2.9.4 |
| 14 | mcrypt-2.6.8 |
| 15 | pnp4nagios-0.6.25 |
| 16 | expat-2.1.0 |
| 17 | flac-1.3.1 |
| 18 | lcms2-2.7 |
| 19 | e2fsprogs-1.42.13 |
| 20 | libpng-1.6.19 |
| 21 | unzip610b |
| 22 | mpg123-1.22.4 |
| 23 | ganglia-3.7.2 |
| 24 | nagios-4.1.1 |
| 25 | clamav-0.98.7 |
| 26 | net-snmp-5.4.5.pre1 |
| 27 | flex-2.6.0 |
| 28 | vice-2.4 |
| 29 | gnuplot-5.0.1 |
| 30 | zabbix-2.4.7 |
| 31 | optipng-0.7.5 |
| 32 | pcre2-10.20 |
| 33 | freetype-2.6 |
| 34 | amanda-3.3.7p1 |

Table 20: Tested Software

# Appendix C

# Appendices for Chapter 5

| Index | Version | Index | Version |
|---|---|---|---|
| 1 | 5.0.375.1 | 17 | 7.0.517.2 |
| 2 | 5.0.375.2 | 18 | 7.0.517.4 |
| 3 | 5.0.375.3 | 19 | 7.0.517.5 |
| 4 | 5.0.375.4 | 20 | 7.0.517.6 |
| 5 | 5.0.375.5 | 21 | 7.0.517.7 |
| 6 | 5.0.375.6 | 22 | 7.0.517.8 |
| 7 | 5.0.375.7 | 23 | 7.0.517.9 |
| 8 | 5.0.375.8 | 24 | 7.0.517.10 |
| 9 | 5.0.375.9 | 25 | 7.0.517.11 |
| 10 | 5.0.375.10 | 26 | 7.0.517.12 |
| 11 | 5.0.375.11 | 27 | 7.0.517.13 |
| 12 | 5.0.375.12 | 28 | 7.0.517.14 |
| 13 | 5.0.375.13 | 29 | 17.0.921.3 |
| 14 | 5.0.375.14 | 30 | 24.0.1298.1 |
| 15 | 6.0.495.1 | 31 | 31.0.1614.2 |
| 16 | 7.0.517.1 | | |

Table 21: Mishandled Versions

| version | nNCVE | n0.5YCVE | nuF | nuPF | nuBlk | nuCmt | nuC | 1CVE | t1CVE | lCVE | tLCVE | dRlse | dayRlse |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3.0.195.25 | 459 | 27 | 36195 | 8218 | 329033 | 471171 | 1740406 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1236 | 4/1/2010 15:30 | 10/8/2009 16:05 | Thursday |
| 3.0.195.27 | 459 | 27 | 36195 | 8218 | 329033 | 471171 | 1740406 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1236 | 4/1/2010 15:30 | 10/10/2009 15:52 | Saturday |
| 3.0.195.33 | 484 | 38 | 36198 | 8221 | 329051 | 471193 | 1740593 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1767 | 9/24/2010 12:00 | 11/11/2009 1:58 | Wednesday |
| 3.0.195.36 | 459 | 55 | 36199 | 8222 | 329064 | 471194 | 1740669 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1773 | 9/24/2010 12:00 | 12/7/2009 15:18 | Monday |
| 3.0.195.37 | 459 | 55 | 36199 | 8222 | 329067 | 471198 | 1740689 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1773 | 9/24/2010 12:00 | 12/8/2009 15:13 | Tuesday |
| 3.0.195.38 | 459 | 55 | 36199 | 8222 | 329066 | 471197 | 1740687 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1773 | 9/24/2010 12:00 | 12/9/2009 11:31 | Wednesday |
| 4.0.212.1 | 458 | 27 | 36882 | 9305 | 350922 | 503113 | 1859941 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1236 | 4/1/2010 15:30 | 9/22/2009 15:03 | Tuesday |
| 4.0.221.8 | 458 | 27 | 37617 | 9697 | 364746 | 512195 | 1927081 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1236 | 4/1/2010 15:30 | 10/6/2009 14:09 | Tuesday |
| 4.0.222.0 | 458 | 27 | 37712 | 9703 | 364848 | 511409 | 1928738 | CVE-2010-0315 | 1/14/2010 11:30 | CVE-2010-1236 | 4/1/2010 15:30 | 10/7/2009 21:14 | Wednesday |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 60.0.3079.2 | 0 | 0 | 261554 | 2909576 | 151354 | 3022186 | 16523993 | Null | Null | Null | Null | 4/23/2017 22:01 | Sunday |
| 60.0.3079.3 | 0 | 0 | 261554 | 2909576 | 151354 | 3022186 | 16523993 | Null | Null | Null | Null | 4/24/2017 16:04 | Monday |
| 60.0.3080.0 | 0 | 0 | 261603 | 2910521 | 151388 | 3023009 | 16528977 | Null | Null | Null | Null | 4/24/2017 20:01 | Monday |
| 60.0.3080.1 | 0 | 0 | 261603 | 2910521 | 151388 | 3023009 | 16528977 | Null | Null | Null | Null | 4/24/2017 21:02 | Monday |
| 60.0.3080.2 | 0 | 0 | 261603 | 2910521 | 151388 | 3023177 | 16528809 | Null | Null | Null | Null | 4/24/2017 22:02 | Monday |

Table 22: Chrome Dataset

| repo | CVE | #stars | #watches | #forks | #issues | #contributors | #commits | creD | puD | #labels | %Java | %JavaScript | %PHP | %C | %C++ | %Python | %Go | %Ruby |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Microsoft/ChakraCore | 21 | 6523 | 423 | 821 | 1171 | 95 | 8454 | 2016-01-05T19:05:31Z | 2017-11-14T02:43:29Z | 16 | 0 | 53 | 0 | 2 | 38 | 0 | 0 | 0 |
| lota/phamm | 1 | 14 | 8 | 19 | 13 | 4 | 60 | 2014-12-06T13:34:34Z | 2017-07-30T15:11:32Z | 0 | 0 | 1 | 94 | 0 | 0 | 0 | 0 | 0 |
| zulip/zulip | 2 | 5526 | 256 | 1684 | 2798 | 360 | 22985 | 2015-09-25T16:37:25Z | 2017-11-14T06:34:20Z | 1 | 0 | 28 | 0 | 0 | 0 | 50 | 0 | 3 |
| repo | %Perl | %HTML | %C# | %Shell | size | #files | #functions | #program-files | blank | comment | code | hits | c-sloc | L1 | L2 | L3 | L4 | L5 |
| Microsoft/ChakraCore | 0 | 0 | 0 | 0 | 153222312 | 6493 | 33259 | 4396 | 219983 | 174869 | 2251192 | 1559 | 720699 | 462 | 555 | 96 | 439 | 7 |
| lota/phamm | 0 | 0 | 0 | 0 | 1619973 | 149 | 59 | 60 | 2888 | 3632 | 17204 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| zulip/zulip | 5 | 6 | 0 | 1 | 48003460 | 3327 | 2060 | 2341 | 55981 | 40968 | 290524 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 23: GitHub Project Dataset

| Features | Meaning |
|---|---|
| repo | the name of the GitHub project |
| CVE | the number of CVEs for this project |
| #stars | the number of GitHub registers who liked this project |
| #watches | the number of GitHub registers who receive notification from this project |
| #forks | the number of GitHub registers who code with this project |
| #issues | the number of issues in this project |
| #contributors | The number of contributors in this project |
| #commits | The number of the commits in this project |
| creD | the created date of the project |
| puD | the last push date of the project |
| #labels | the number of labels of the project |
| %Java | the percentage of the Java language of the project |
| %JavaScript | the percentage of the JavaScript language of the project |
| %PHP | the percentage of the PHP language of the project |
| %C | the percentage of the C language of the project |
| %C++ | the percentage of the C++ language of the project |
| %Python | the percentage of the Python language of the project |
| %Go | the percentage of the Go language of the project |
| %Ruby | the percentage of the Ruby language of the project |
| %Perl | the percentage of the Perl language of the project |
| %HTML | the percentage of the HTML language of the project |
| %C# | the percentage of the C# language of the project |
| %Shell | the percentage of the Shell language of the project |
| size | the size of this project |
| #files | the number of the files |
| #functions | the number of the functions |
| #program-files | the number of program files |
| blank | the line number of the blanks |
| comment | the line number of the comments |
| code | the line number of the codes |
| hits | the total number of the flaws found by Flawfinder |
| c-sloc | the line number of C/C++ code |
| L1 | the number of the flaws in severity level 1 |
| L2 | the number of the flaws in severity level 2 |
| L3 | the number of the flaws in severity level 3 |
| L4 | the number of the flaws in severity level 4 |
| L5 | the number of the flaws in severity level 5 |

Table 24: Meaning of the Features in Table 23

| Features | Meaning |
|---|---|
| version | the version number of Chrome |
| nNCVE | the number of CVEs from NVD interpretation |
| n0.5YCVE | the number of CVEs based on 0.5 year model |
| nuF | number of files |
| nuPF | number of program files |
| nuBlk | line number of blanks |
| nuCmt | line number of comments |
| nuC | line number of code |
| 1CVE | the first CVE published in this version |
| t1CVE | the published date for the first CVE |
| lCVE | the last CVE published in this version |
| tLCVE | the published date for the last CVE |
| dRlse | the release date |
| dayRlse | the release day |

Table 25: Meaning of the Features in Table 22