Dynamic Risk-Aware Patch Scheduling

Fengli Zhang
University of Arkansas
fz002@uark.edu

Qinghua Li
University of Arkansas
qinghual@uark.edu

***Abstract*—Every month, many new software vulnerabilities are discovered and published which will pose security risks to power grid systems if they are exploited by attackers. Thus the vulnerabilities must be patched in a timely manner to reduce the chance of being exploited. However, not all vulnerabilities can be patched quickly due to limited security resources at many electric utility companies. This paper studies dynamic risk-aware patch scheduling to determine the order of patching vulnerabilities and minimize the security risk brought by vulnerabilities. We first predict the dynamic probability of exploit over time for each vulnerability and define a metric to compute the vulnerability's dynamic risk based on the predicted probability. We then formulate two patch scheduling approaches. Evaluations on real datasets show high accuracy in predicting the dynamic probability of exploit and high effectiveness of our solutions in risk reduction compared with other scheduling methods.**

***Index Terms*—Vulnerability, risk, patching, scheduling**

I. Introduction

Vulnerabilities in software/firmware pose significant security risks to power systems since they can be exploited by attackers. Vulnerabilities are unavoidable and could affect all installed software and firmware components. Moreover, the power grid is moving toward a more interconnected grid, which carries the potential for higher risk exposure for those vulnerabilities. Every month, many new vulnerabilities are discovered and released [1], which need to be patched timely to reduce the risk that they pose to the system. However, not all vulnerabilities can be patched quickly due to limited resources and security personnel at many electric utility companies. Properly scheduling the vulnerability patching order can significantly reduce the overall security risk that the system faces.

In practice, many electric utility organizations determine the patching order for vulnerabilities based on security operators' subjective assessment and/or preference. That could result in random, ad hoc patching orders. Others schedule patches based on the prioritization order provided by the patch management software, where the patches are usually prioritized by vulnerabilities' risks measured by Common Vulnerability Scoring System (CVSS) scores [2]. CVSS score shows the severity of software vulnerabilities which is assessed and maintained by the National Vulnerability Database (NVD) [3]. However, existing metrics for prioritizing patches have two weaknesses. First, they are static and hence cannot capture the time-dependent nature of risks as elaborated below. Second, they do not consider the time cost of applying patches. Thus, although they could be used for prioritizing patches and generating a simple patch schedule, the ignorance of patching cost could lead to suboptimal decisions especially for small-sized and medium-sized electric utilities with limited security resources.

The risk of a vulnerability depends on the likelihood of exploitation and the impact of exploitation. The likelihood of exploitation is dynamic in nature and changes with time. The difficulty of developing exploit for a vulnerability depends on the nature of the software (e.g., Windows or Linux software), the nature of the vulnerability (e.g., buffer overflow or protocol design flaws), whether exploits have been developed before for similar vulnerabilities, and how much value the software/vulnerability has to the adversary. They all affect how long it will take the adversary to develop exploits for vulnerabilities. Our experimental study (see more details in Section III-B and IV-A) shows that different vulnerabilities have different time-likelihood curves.

Neglecting such dynamic natures will cause ineffective risk reduction. For a simple example, suppose two vulnerabilities v_a and v_b are published at the same time, have the same overall risk score (e.g., CVSS score), and each takes one day to patch. v_a is highly likely to be exploited in day 1 after public release. v_b has a very low likelihood to be exploited in the first two days after release but is more likely to be exploited after day 2. If patch scheduling only considers the overall risk score, either of them can be patched first. However, in this example, patching v_a first and v_b next will have lower security risks.

Although the impact score of published vulnerabilities is usually available (e.g., the CVSS impact score) as a good estimate of their impact, their likelihood of exploitation is usually unknown, not to mention the dynamic likelihood over time. In this work, we first design a method to predict vulnerabilities' dynamic probability of exploit over time (we define a vulnerability's *probability of exploit at time* τ as the probability that known exploit code[1] for the vulnerability is available by time τ after the vulnerability is released at time 0). A risk metric is defined based on the dynamic probability of exploit. Then we formulate dynamic risk-aware patching scheduling problems to minimize the total security risks posed by vulnerabilities. This paper's contributions are summarized as follows:

- We formulate dynamic risk-aware patch scheduling problems. To the best of our knowledge, this is the first work that considers the dynamic risks of vulnerabilities into patch scheduling for electric utility companies. It can reduce security risks posed by vulnerabilities and provide formal guidance to security operations at electric utilities.
- To estimate dynamic risks, we design a method for predicting vulnerabilities' dynamic probability of exploit

[1]It is possible that an adversary has developed exploit code for a vulnerability but does not publish it. However, we only consider known exploit in this paper to make the problem tractable.

over time based on neural network and natural language processing techniques.
- We propose a baseline scheduling method that has optimal risk reduction but high computation overhead, and a group-based scheduling method that has much lower computation cost but still effective risk reduction.
- We evaluate the proposed patch scheduling solution based on a real dataset from an electric utility company and in comparison with three other scheduling methods, and demonstrate that our solution can significantly reduce the security risks.

Although this work is presented in the context of electric utility systems, it can be applied to many other applications especially critical infrastructure systems.

The paper is organized as follows. Section II discusses related work. Section III describes the method to predict vulnerability's probability of exploit and the patch scheduling formulations. Section IV presents evaluation results and the last section concludes the paper.

II. Related Work

To the best of our knowledge, no work has been done to schedule the patching order by considering vulnerabilities' dynamic probabilities of exploit to minimize the security risk. There have been some studies about patching prioritization [4]–[9] and vulnerability remediation [10], [11]. Some of them prioritize vulnerabilities by quantifying the vulnerabilities' risk based on CVSS temporal and environmental scores [4] and commercial metrics [9]. The work in [5] measures the importance of vulnerabilities and their hosts by attack cost and system risk respectively, and then combines the measures to prioritize vulnerabilities. The work in [6] considers patch operators and attackers as two players and models the patching management as a search game framework. It assumes both players are rational and thus can predict the players' behaviours with the game theory framework. The work in [7] and [8] determines the patching order through game theory in an attack/defense scenario. These works do not consider the vulnerabilities' dynamic nature and that the probability of exploit is changing over time. Thus, they cannot adapt to the dynamic risks posed by vulnerabilities.

There have been some existing researches on vulnerability's exploitability prediction. The work in [12] describes how to predict the exploitability with Support Vector Machine (SVM). The work in [13], [14] applies and compares several machine learning algorithms such as Naive Bayes and SVM to predict the exploitability. Sabottke et. al. [15] design an exploit detector to detect exploits on Twitter, where vulnerability and exploit related information might be posted. The work in [16] predicts exploits with neural language model by collecting the web post data. However, these works mainly predict whether vulnerabilities can be exploited, but do not address how the probability of exploit changes over time which is studied in this paper.

III. Approach

The patching order of vulnerabilities can significantly affect the total security risk that a system is facing. In order to schedule the patching order, we need to know each vulnerability's risk. Thus in this section we first define the vulnerability risk metric based on probability of exploit and impact of exploit. We then predict vulnerabilities' dynamic probability of exploit which will allow calculation of vulnerability risks. After getting each vulnerability's risk, we formulate scheduling problems to minimize the vulnerabilities' total risk.

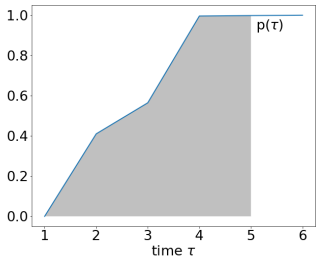

Fig. 1. Probability of exploit for vulnerability v_a

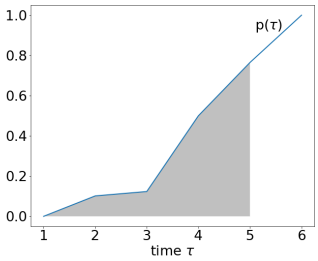

Fig. 2. Probability of exploit for vulnerability v_b

A. Risk Metric

In practice, electric utility companies usually collect the vulnerabilities of their assets from vendors, third party services, public databases such as the NVD, or a combination of these sources. The vulnerabilities come with or can be easily matched to the standard CVSS metrics, which show the principal characteristics of a vulnerability and provide a numerical score (called *CVSS score*) reflecting the vulnerability's overall severity [17]. CVSS score can be used to quantify a vulnerability's risk (and is used by many companies to do so). However, it is static and does not capture a vulnerability's dynamic probability of exploit that changes with time. Here we define a new metric to quantify the dynamic risk that a vulnerability poses to the system by considering its dynamic probability of exploit and its impact on the system:

$$r(t) = I \cdot \int_0^t p(\tau) d\tau \tag{1}$$

Here $r(t)$ denotes the risk that the vulnerability has posed to the system by time point t (assuming the vulnerability is published at time 0), $p(\tau)$ is the vulnerability's probability of exploit at time τ, and I is the vulnerability's impact score on the system. For impact score, we follow the calculation of the standard CVSS scoring system [18] which makes it a constant irrelevant to time. Specifically, $I = 10.41 \cdot (1 - (1 - ConfImpact) \cdot (1 - IntegImpact) \cdot (1 - AvailImpact))$. Here $ConfImpact$, $IntegImpact$, and $AvailImpact$ are three characteristics defined in CVSS metrics to show the vulnerability's impact on the system's confidentiality, integrity and availability respectively. Each characteristic has three different levels, 'complete', 'partial' and 'none', whose corresponding numeric values are 0.660, 0.275 and 0.0 respectively [18].

The risk definition involves two factors. The first factor is impact score, which is easy to understand. The second factor is $\int_0^t p(\tau)d\tau$. If the function $p(\tau)$ is plotted as a curve (see Fig. 1 and 2 for examples), this factor is essentially the size of the grey area under the curve. It in turn considers two factors, the probability that exploit code is available and the time duration of exploit code being available. This makes sense since the more quickly a vulnerability's probability of exploit increases and the longer time a vulnerability is exposed (i.e., stays unpatched) the higher risk it could pose to the system. For the examples in Fig. 1 and 2, vulnerability v_a has higher risk than v_b when considering the exposure time up to day 5.

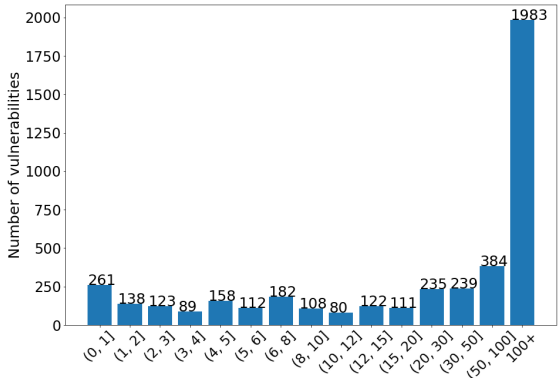

Fig. 3. Distribution of vulnerabilities in the day/time window of exploit after vulnerability release

B. Predicting the Probability of Exploit

The exploit code for a vulnerability can be developed and released any time after the vulnerability is published and, in some cases, even before the vulnerability is published. Knowing how soon the vulnerabilities are likely to have exploit code can help schedule when to patch them. In this section, we describe how to leverage neural network models to predict a vulnerability's probability of exploit.

1) Vulnerability Dataset: We collected all the vulnerabilities in the Vulners database [19] in June 2019, which consist of 67,965 vulnerabilities. 46,380 of them have no known exploits, 17,260 already have exploits before vulnerabilities are published, and for the other 4,325 vulnerabilities their exploits are released after they are published. For the 4,325 vulnerabilities, the distribution of vulnerability exploit time (i.e., the day when a vulnerability's exploit code becomes available) after a vulnerability is released is shown in Fig. 3. The X-axis shows the day (or time window) when exploit becomes available for the vulnerability and the Y-axis shows the number of vulnerabilities whose exploits become available in that day (or time window). For example, 261 vulnerabilities become to have exploit in day 1 after their release, 182 vulnerabilities become to have exploit in day 7 and day 8 after their release, and 1,983 vulnerabilities become to have exploit available 100 or more days (up to 5,000 days) after their release. We use the 4,325 vulnerabilities whose exploits appear after they are published as the dataset for training the prediction model. To keep the dataset's balance, we randomly sampled 4,325 out of the 46,380 vulnerabilities that do not have known exploits and add them to the dataset as well. This aggregate dataset will be used as training data to train the neural network model.

2) Feature Extraction: To train the neural network model, we need to select important features to represent each vulnerability as similarly done in prior work [12], [14], [15]. Since CVSS is used to describe the primary characteristic of vulnerabilities, we use CVSS metrics as part of the features which include: attack vector (how the vulnerability is exploited, e.g. through network or local access), attack complexity, user interaction (whether user interaction is needed to exploit the vulnerability), privilege (the privilege level required to exploit vulnerability), confidentiality impact (the impact on the system's confidentiality if the vulnerability is exploited), integrity impact, availability impact, and CVSS score (the vulnerability's overall severity). Common weakness enumeration (CWE) [20] which depicts the vulnerability type, and software name are also used as features. Each vulnerability has its description and title which also provide valuable information. Since descriptions and titles are descriptive texts, we apply tf-idf (term frequency-inverse document frequency) [21] to extract important words from description and title texts as features. Tf-idf is able to identify important words and deprive the words that do not carry useful information such as "the" and the common words that appear in most descriptions and titles such as "CVE". We use tf-idf to extract bi-grams (two adjacent words) as features, which we found has better performances than uni-gram (single word) in this prediction task. The top 3,000 most important bi-grams such as "windows server", "execute arbitrary", "verify certificates", "spoof servers", "access restriction" from descriptions and the top 500 most important bi-grams from titles are extracted as features used in prediction (we tried other numbers but found these two perform best as shown in table III and IV in Section IV-A). Besides, each vulnerability has publish date, modify date and last seen date. Publish date is the day when the vulnerability is published; modify date is the day when the vulnerability is modified such as the modification of its title or description; and last seen date is the last date when the vulnerability was seen. If a vulnerability is modified or is still seen long after being published, it shows that this vulnerability still persists and attracts people's attention. Such vulnerabilities may carry high risk and are more likely to be attackers' targets. Therefore, similar to [12], we also use the time difference between modify date and publish date and the time difference between last seen date and publish date as the features. The number of features is shown in Table I. Each vulnerability record has 3,512 features in total.

3) Prediction with Neural Network: We build neural network models to predict the probability of exploit. The collected vulnerability dataset described above can be used to train the neural network models. We predict the probability of exploit for vulnerabilities by each day. To do this, we build one neural network model for each day. When building the

TABLE I
FEATURES USED IN PREDICTING THE PROBABILITY OF EXPLOIT

Feature Family	CVSS metric	CWE	software name	description	title	modify date - publish date	last seen date - publish date
Number	8	1	1	3000	500	1	1

TABLE II
A MOTIVATING EXAMPLE FOR PATCH SCHEDULING

Vulnerability	Impact score	Patching time cost	Exploit probability by the 1st day	Exploit probability by the 2nd day	Exploit probability by the 3rd day
v_a	8.0	2 days	0.6	0.65	0.7
v_b	7.0	1 day	0.6	0.8	0.9

model for predicting the probability of exploit by the n_{th} day after vulnerability release, we generate the training data for this model by relabeling the vulnerability dataset. If a vulnerability's exploit code becomes available within n or less days, it is labeled as 1; otherwise, it is labeled as 0. Then this relabeled dataset is used to train the neural network model and this model will be used to predict the probability of exploit for the n^{th} day. In this paper, we assume vulnerability patch scheduling is done monthly, since most electric utility companies follow this scheduling frequency. That means we need to predict the exploit probability and build one model for each day in the following month. Therefore, about 30 neural network models need to be built in each scheduling cycle.

To predict for a new vulnerability, the vulnerability's corresponding features can be fed into the 30 trained neural models which will output the exploit probability by each day of the 30 days.

C. Patch Schedule Optimization

A vulnerability's probability of exploit is changing over time and so is the risk that the vulnerability poses to the system. It is important to carefully schedule the patching order of vulnerabilities so that the system's total security risk is as low as possible. Here we take an example to show how the patching order affects the total risk. Suppose that a system has two vulnerabilities to patch, v_a and v_b. v_a's impact score is 8.0, v_b's impact score is 7.0 and they are published at the same time. The probabilities of exploit are shown in Table II. It takes about 2 days to patch v_a and 1 day to patch v_b. If we prioritize the patches only by impact score, v_a will be patched first. Then the risk from V_a is $8.0 \cdot 0.6 \cdot 1 + 8.0 \cdot 0.65 \cdot 1 = 10.0$, the risk from V_b is $7.0 \cdot 0.6 \cdot 1 + 7.0 \cdot 0.8 \cdot 1 + 7.0 \cdot 0.9 \cdot 1 = 16.1$, and the total risk is 26.1. If V_b is patched first, the risk from V_a is $8.0 \cdot 0.6 \cdot 1 + 8.0 \cdot 0.65 \cdot 1 + 8.0 \cdot 0.7 \cdot 1 = 15.6$, the risk from V_b is $7.0 \cdot 0.6 \cdot 1$, and the total risk is 19.8. From the example, it can be seen that different patching orders can significantly affect the total risk, and considering dynamic risks matters. Next, we formulate patch scheduling as optimization problems.

1) Baseline Scheduling: Let v_i denote vulnerability i, $p_i(\tau)$ denote the exploit probability of vulnerability v_i over time τ, I_i denote the impact score of v_i, and d_i denote the time cost of processing/installing patch i. Our objective is to schedule the patch order so that the total risk that the vulnerabilities pose to the system is minimized. The risk of an unpatched vulnerability at time t is $r(t)$ as defined in Equation 1. Since a vulnerability cannot be exploited after it is patched, this vulnerability will not pose any risk to the system after being patched. Thus the total risk caused by vulnerability i is: $r_i(s_i + d_i) = I_i \cdot \int_0^{s_i+d_i} p_i(\tau) \cdot d\tau$, where s_i is the time point when patch starts. We convert this formula to discrete calculation which is: $r_i(s_i + d_i) = I_i \cdot \sum_{\tau=0}^{\tau=s_i+d_i-1} \frac{p_i(\tau)+p_i(\tau+1)}{2}$. In this work, we set 30 minutes as the time unit (but it can be changed based on different application scenarios). Then d_i = 2 means it takes 2 time units to complete patching, which is 60 minutes. s_i =10 means the patching starts at time unit 10. For all the vulnerabilities, the total risk is: $\sum_{i=0}^{n} r_i(s_i + d_i)$, where n is the number of vulnerabilities. The goal is to minimize the total risk that the unpatched vulnerabilities pose to the system.

The formulation is given in Algorithm Baseline Scheduling. The input of the model includes the set of vulnerabilities to be patched, V, the set of available operators to apply patches, A, the time cost d_i to patch vulnerability i, and the risk of each vulnerability $r_i(t)$. s_i is the scheduled start patching time for vulnerability i and $X_{(i,a)}$ indicates whether vulnerability i is allocated to operator a. Both s_i and $X_{(i,a)}$ are variables that need to be determined during the optimization. $\theta_{(i,k)}$ is a support variable to determine whether the patching time of two vulnerabilities are overlapped. Three constraints are set: each patch must be assigned once to exactly one operator; if two patches are assigned to the same operator, their execution times cannot overlap; and if a patch is dependent on another, this patch cannot start until its predecessor are completed.

This formulated scheduling model is a NP-hard problem. The job shop scheduling problem (JSP) can be reduced to our problem. JSP is a common scheduling problem where jobs of varying time are assigned to resources (e.g. machines) [22]. The vulnerabilities to patch in our problem can be regarded as the jobs and the operators can be regarded the resources in JSP. Since JSP is NP-hard, this formulated scheduling model is also NP-hard.

2) Group-based scheduling: The baseline scheduling model can give an optimal patching order for all the vulnerabilities, but since it is NP-hard, it takes too long time to get the optimized schedule when there are many vulnerabilities. Our experiments show that when there are 200 vulnerabilities, it will take more than 2 days to solve it which is not practical. However, in practice it is not uncommon for an organization to have several hundred of vulnerabilities to patch in each month.

Thus, a computationally more efficient solution is needed. To address this need, we propose a group-based scheduling approach which can greatly reduce the computation overhead while still providing good effectiveness in risk reduction.

The basic idea is to divide assets into groups based on their functions and other relevant factors. Then the scheduling problem can be solved in two phases. In the first phase (group-level scheduling), we consider the vulnerabilities in each asset group as one vulnerability (or task) and determine the order of groups to be patched. In the second phase (intra-group scheduling), we consider each group separately and schedule the vulnerabilities in each group to determine their order of patching. The computation cost of the first phase depends on the number of groups, and the computation cost of the second phase depends on the number of vulnerabilities in each group. Since both numbers are much smaller than the total number of vulnerabilities (i.e., the problem size in the baseline scheduling), this group-based solution should have much lower computation cost.

Algorithm Baseline Scheduling

Inputs:
V set of applicable vulnerabilities $i \in V$
A set of all available operators $a \in A$ to apply patches
d_i execution delay to patch vulnerability i
$r_i(t)$ risk of vulnerability i
s_i scheduled start time of vulnerability i
$X_{(i,a)}$ allocation of vulnerability i, $X_{(i,a)} = 1$ if operator a is assigned to vulnerability i; otherwise, $X_{(i,a)} = 0$
$\theta_{(i,k)}$ support variable used to determine whether the processing time of vulnerability i and k are overlapped. $\theta_{(i,k)} = 1$ if vulnerability k is started before i is completed, otherwise $\theta_{(i,k)} = 0$

Goal:
Minimize: $\sum_{i \in V} \sum_{a \in A} X_{(i,a)} r_i(s_i + d_i)$

Subject to:
- $\sum_{a \in A} X_{(i,a)} = 1$ (each patch must be assigned once to exactly one operator)
- $X_{(i,a)} + X_{(k,a)} + \theta_{(i,k)} + \theta_{(k,i)} <= 3$ (if patch i and k are assigned to the same operator, their execution times cannot overlap)
- $s_k >= s_i + d_i$ (if a patch depends on another patch, it cannot start until its predecessor is completed)

The idea of group-based scheduling is also consistent with existing practices in vulnerability and patch management. In current practices, organizations usually divide their assets into different groups based on the assets' functions and physical locations for easier management. Security operators need to coordinate with other system users to schedule downtime for machines before applying patches. Once an asset group's machines are scheduled, all the patches applicable to the asset group can be applied in succession.

Phase I: group-level scheduling In this phase, each asset group is taken as one task. We use $R_j(t)$ to denote the risk of all vulnerabilities in asset group j over time t and D_j is the execution delay to process all vulnerabilities in asset group j, where $R_j(t) = \sum_{i=0}^{i=n_j} r_i(t)$ and $D_j = \sum_{i=0}^{i=n_j} d_i$. n_j is the number of vulnerabilities in asset group j. In an organization, there might be multiple operators applying patches. One operator can be responsible for one or several asset groups, but when one asset group is assigned to an operator, the operator will be responsible for all the vulnerabilities in that asset group. With these considerations, the asset group scheduling optimization problem can be formulated in Algorithm Group-Based Scheduling - Group-Level Scheduling.

Algorithm Group-Based Scheduling - Group-Level Scheduling

Inputs:
J set of all asset groups $j \in J$ in the organization
A set of all available operators $a \in A$ to apply patches
D_j execution delay to process all patches in asset group j
$R_j(t)$ total risk of all vulnerabilities in asset group j
S_j scheduled start time of asset group j
$X_{(j,a)}$ allocation of asset group j, $X_{(j,a)} = 1$ if operator a is assigned to asset group j; otherwise, $X_{(j,a)} = 0$
$\theta_{(j,k)}$ support variable used to determine whether the processing time of asset group j and k are overlapped for each operator. $\theta_{(j,k)} = 1$ if asset group k is started before j is completed, otherwise $\theta_{(j,k)} = 0$

Goal:
Minimize: $\sum_{j \in J} \sum_{a \in A} X_{(j,a)} R_j(S_j + D_j)$

Subject to:
- $\sum_{a \in A} X_{(j,a)} = 1$ (each group must be assigned once to exactly one operator)
- $X_{(j,a)} + X_{(k,a)} + \theta_{(j,k)} + \theta_{(k,j)} <= 3$ (if asset group j and k are assigned to the same operator, their execution times cannot overlap)

Algorithm Group-Based Scheduling - Intra-Group Scheduling

for each asset group do:
Inputs:
V set of applicable vulnerabilities $i \in V$ in the asset group
d_i execution delay to process vulnerability i
$r_i(t)$ risk of vulnerability i
s_i scheduled start time of vulnerability i
$\theta_{(i,k)}$ support variable used to determine whether the processing time of vulnerability i and k are overlapped. $\theta_{(i,k)} = 1$ if vulnerability k is started before i is completed, otherwise $\theta_{(i,k)} = 0$

Goal:
Minimize: $\sum_{i \in V} r_i(s_i + d_i)$

Subject to:
- $\theta_{(i,k)} + \theta_{(k,i)} <= 1$ (two patches' execution time cannot overlap)
- $s_k >= s_i + d_i$ (a patch cannot start until its predecessor are completed)
- $s_i >= S_j$ (vulnerabilities can be patched after its asset group start time S_j)

Phase II: intra-group scheduling After getting the asset group patching order, we need to schedule the patching order for the vulnerabilities in each group. We can get each asset group's patch start time S_j from first phase's scheduling. Then the vulnerabilities in this asset group should be patched starting from S_j. The formulated optimization algorithm is

TABLE III
PREDICTION ACCURACY VS. NUMBER OF DESCRIPTION FEATURES

Number of description features	0	500	1000	2000	3000	3500
Prediction accuracy	0.853	0.859	0.865	0.877	0.896	0.885

TABLE IV
PREDICTION ACCURACY VS. NUMBER OF TITLE FEATURES

Number of title features	0	100	200	400	500	600
prediction accuracy	0.871	0.875	0.881	0.891	0.896	0.893

described in Algorithm Group-Based Scheduling - Intra-Group Scheduling.

Similar to baseline scheduling, the group-based scheduling problem is also NP-hard.

3) Solving the optimization problems: The solver for the scheduling optimization problems is implemented with around 2,000 lines of Python code by using the OR-Tools library, which is developed by Google for optimization problems [23].

IV. EVALUATION

A. Prediction on Probability of Exploit

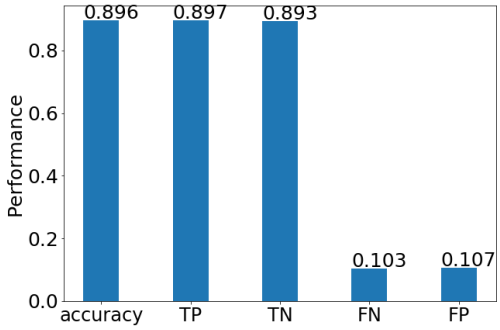

Fig. 4. Exploitability Prediction

The dataset used in this exploitability prediction experiment consists of 8,650 vulnerabilities, half of which have exploit and half do not have exploit as described in Section III-B1. In evaluations, we split the dataset into two parts: 67% as the training data and the remaining as testing data. The neural network prediction models were implemented with Python. The vulnerabilities' features as described in Section III-B2 are the input of the neural network. Each model has two hidden layers with 100 and 30 neurons respectively. The output layer outputs the probability of exploit.

In order to determine the best number of features for predictions, we first test the prediction accuracy over different number of description and title features. The results are shown in Table III and IV. It can be seen that when the number of features increases, the prediction accuracy first increases and then decreases. When there are 3,000 description features and 500 title features, the performance is the best. Thus we extracted the 3,000 most important bi-grams from descriptions and the 500 most important bi-grams from titles as features in this evaluation (see other features in Section III-B2).

TABLE V
TRAINING AND PREDICTION TIME

training time	63.5 minutes
prediction time	1.34 milliseconds

We then test the accuracy of prediction without considering when the vulnerabilities' exploits become available. No matter when a vulnerability starts to have exploit, if it has exploit, it will be labeled as 'exploited'. The prediction accuracy is shown in Fig. 4. It has 89.6% prediction accuracy (the portion of correct predictions out of all predictions) and the true positive (TP: having exploit and predicted to have it) is 89.7%. The true negative (TN: not having exploit and predicted to not have it), false negative (FN: having exploit but predicted to not have it) and false positive (FP: not having exploit but predicted to have it) are 89.3%, 10.3%, and 10.7% respectively.

Next we evaluate exploit predictions over time. For each day's exploit prediction, there is one neural network model trained and tested specially for that day as discussed in Section III-B3. Here we mainly predict the exploit probability for 30 days, and thus we have 30 neural network models. The models were run on a laptop computer with 1.60 GHz CPU and 16.0 GB memory. The training time and prediction time of those models are shown in Table V. It takes 63.5 minutes to train all the models. These models are trained once a month and can be trained in advance. Thus the training of the models will not delay any patching. The prediction for each vulnerability is super quick and it only takes 1.34 milliseconds. The prediction accuracy is shown in Table VI. As the time goes by, the prediction accuracy becomes higher and higher. For example, the prediction accuracy for Day 1 is 74.7%, and the prediction accuracy for Day 30 is 82.0%. This is because as the time goes by, more vulnerabilities have exploits and the dataset is more balanced between vulnerabilities with exploit and vulnerabilities without it.

TABLE VI
EXPLOIT PREDICTION OVER TIME

Day	**Accuracy**	**TP**	**FN**	**TN**	**FP**
1	0.747	0.742	0.258	0.753	0.247
2	0.783	0.772	0.228	0.795	0.205
3	0.784	0.768	0.232	0.803	0.197
4	0.799	0.788	0.212	0.810	0.190
5	0.801	0.789	0.211	0.814	0.186
6	0.802	0.789	0.211	0.818	0.182
8	0.804	0.784	0.216	0.827	0.173
10	0.809	0.809	0.191	0.808	0.192
15	0.816	0.801	0.199	0.834	0.166
20	0.812	0.81	0.19	0.818	0.182
30	0.820	0.819	0.181	0.821	0.179

We use two example vulnerabilities (v_i and v_k) and show their predicted exploit probabilities over time in Fig. 5 and Fig. 6. The X-axis shows the n_{th} day after being published and the Y-axis denotes the probability of exploit (i.e., $p(\tau)$) at that very day. As time goes by, the probability of exploit becomes

TABLE VII
COMPARISON BETWEEN BASELINE SCHEDULING AND GROUP-BASED SCHEDULING IN RUNNING TIME

Number of Vulnerabilities	120	200
Baseline scheduling	319 seconds	>48 hours
Group-based scheduling	14 seconds	29 seconds

higher and higher. For example, v_i's probability of exploit is around 0.5 at Day 8. In the real dataset, v_i also gets to have exploit in the 8th day after being published. The prediction can give a good estimate over when the vulnerabilities will have exploit.

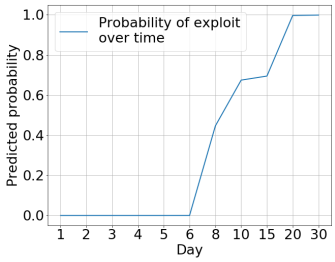

Fig. 5. Predicted probability of vulnerability v_i

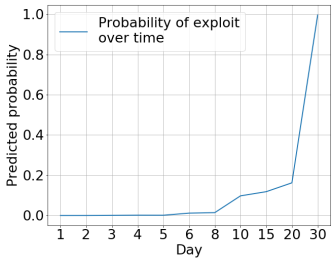

Fig. 6. Predicted probability of vulnerability v_k

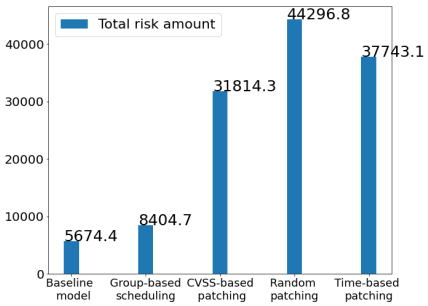

Fig. 7. Comparison with baseline model in total risk

B. Evaluation on Patch Scheduling

In this section, we test how our patch scheduling optimization solution performs. We use the real software asset data from one electric utility company to extract 390 applicable vulnerabilities published in January 2019 from the Vulners database. Based on the electric utility company's input on the time range of patching a vulnerability, we randomly generate the patching time cost for each vulnerability within the range from 0.5 hours to 4 hours. This utility company has 26 asset groups. We first predict the exploit probability for each retrieved vulnerability with our neural network models. Then we run the baseline scheduling and the group-based scheduling methods to get optimal patching orders. One organization usually has multiple security operators to apply patches. Thus we also test how our solution performs when the number of operators changes.

We compare our solution with three other patching methods widely used in practice, patching randomly, CVSS-based patching, and time-based patching. In the random patching method, the operators patch vulnerabilities in random orders. In CVSS-based patching, the vulnerabilities with higher CVSS scores are patched first. In time-based patching, the vulnerabilities which are published earlier are patched first.

These methods are compared along three metrics. The first metric is the total amount of risks that unpatched vulnerabilities pose to the system, which can be calculated as $\sum_{i=0}^{n}\sum_{t=0}^{s_i+d_i} r_i(t)$. The second metric is the number of vulnerabilities that are patched before exploits are available. If some vulnerabilities cannot be patched before exploits are available, we hope they can be patched as soon as possible so that the system is exposed to the exploits for a shorter time. Thus in the third metric, for those vulnerabilities which are not patched before exploits are available, we calculate the time difference between the patching time and the time of exploits being available to see how long the system is exposed to the exploitable vulnerabilities. It can be calculated as $\sum_i (t_{patch\_time} - t_{exploit\_time})$ if $t_{patch\_time} > t_{exploit\_time}$, where t_{patch_time} is the patching time and $t_{exploit\_time}$ is the time when exploit becomes available. We call the time difference as *recovery delay*.

1) Comparison between group-based scheduling and baseline scheduling: We first compare our group-based method with the baseline method. Since it takes too long time for the baseline model to get the optimized patching order for all the 390 vulnerabilities, we only use 120 vulnerabilities for this comparison. In this comparison, only one operator is considered to perform the vulnerability patching operations. The running time is shown in Table VII. The time was measured on a laptop computer with 1.60 GHZ CPU and 16.0 GB memory. The running time of the group-based model to get optimal schedule is 14 seconds, but that of the baseline model is 319 seconds which is about 22 times longer. When there are 200 vulnerabilities to patch, it only takes the group-based method 29 seconds to get the optimal schedule, but it takes more than 48 hours for the baseline model to get the optimal schedule (we ran the method for 48 hours but still did not get the optimal solution). It can be seen that the group-based method has much higher computation efficiency and can handle much larger problem sizes than the baseline method. We then compare how the models perform on risk reduction, which is shown in Fig. 7. For better comparison, CVSS-based patching, random patching and time-based patching are also compared. The total risk of group-based scheduling is higher than the baseline scheduling which is not unexpected, but it is much lower than the risks of the other three scheduling methods, which means group-based scheduling is still very effective in risk reduction. Thus, it achieves good tradeoff between computation cost and effectiveness of risk reduction.

2) Comparison between group-based scheduling and other scheduling methods: In this section, we compare our group-

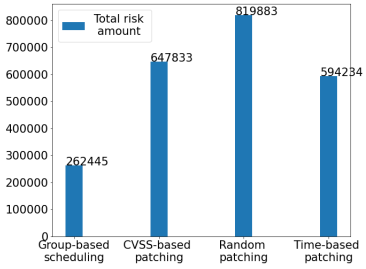

Fig. 8. Total amount of risks of unpatched vulnerabilities under one operator

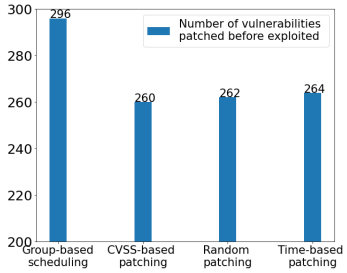

Fig. 9. Number of vulnerabilities patched before exploit appears under one operator

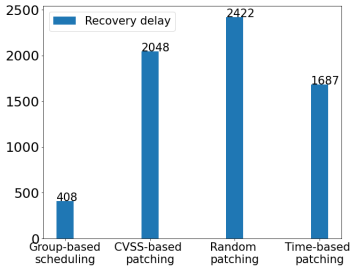

Fig. 10. Recovery delay under one operator

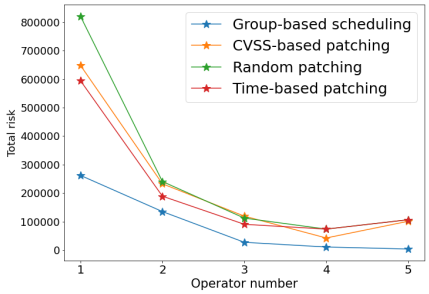

Fig. 11. Total amount of risks of unpatched vulnerabilities under multiple operators

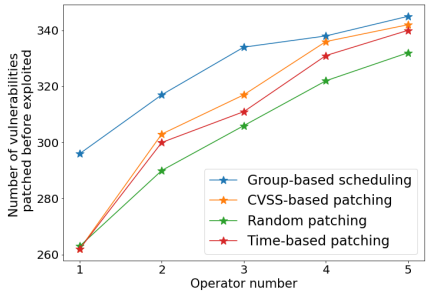

Fig. 12. Number of vulnerabilities patched before exploit appears under multiple operators

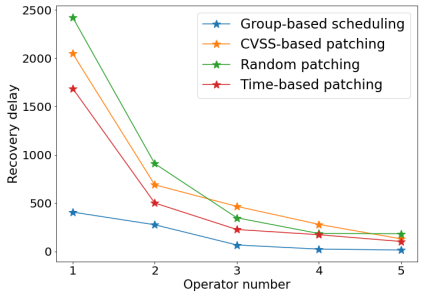

Fig. 13. Recovery delay under multiple operators

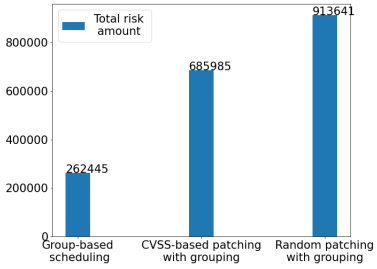

Fig. 14. Total amount of risks of unpatched vulnerabilities under one operator

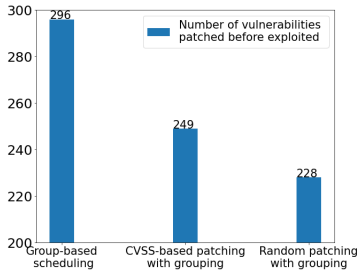

Fig. 15. Number of vulnerabilities patched before exploit appears under one operator

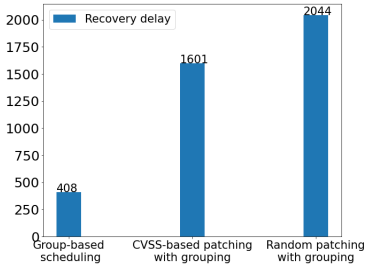

Fig. 16. Recovery delay under one operator

based method with random patching, CVSS-based patching and time-based patching using all the 390 vulnerabilities. The comparisons are shown in Fig. 8, 9 and 10 when there is only one operator. Fig. 8 shows that the total risk can be significantly reduced when scheduling patches with our proposed method. Our method reduces the total risk by about 70%, 60% and 56% respectively compared with random patching, CVSS-based patching and time-based patching. From Fig. 9, we can see that our method can patch most vulnerabilities before they have exploits. Out of those 94 vulnerabilities whose have exploits before they are patched, 49 of them already have exploits before they are published (i.e., zero-day vulnerabilities), which cannot be addressed by any scheduling algorithm. Fig. 10 shows that our method has the lowest recovery delay, which means the system is exposed to the exploitable vulnerabilities for the shortest time.

Fig. 11, 12 and 13 show the comparisons when there are multiple operators. From these figures, it can be seen our proposed method has the lowest total risk, patches more vulnerabilities before they have exploits, and has the lowest recovery delay. When there are more operators, the differences between these methods become smaller. This is because when there are more operators, more vulnerabilities can be patched timely no matter which method is used. Thus, our method is especially useful for small-sized and medium-sized companies that have relatively limited security personnel compared with the amount of vulnerabilities they need to handle.

3) Comparison with other grouping-aware methods: We also added grouping-awareness to random patching and CVSS-based patching and compare them with our group-based method. For the random patching with grouping method, we first randomly select which asset group should be patched first and then for the vulnerabilities in each asset group we randomly determine their patching order. For the CVSS-based patching with grouping method, we first calculate each asset group's CVSS score as the sum of the CVSS scores of the vulnerabilities in the group and patch the asset group with highest CVSS score first. Then for the vulnerabilities in each

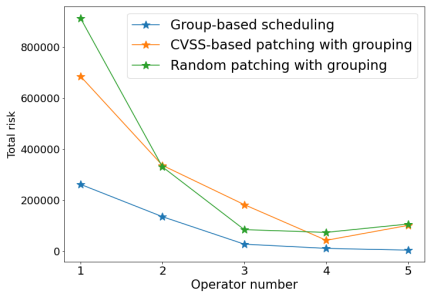

Fig. 17. Total amount of risks of unpatched vulnerabilities under multiple operators

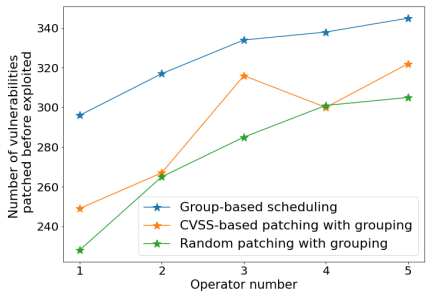

Fig. 18. Number of vulnerabilities patched before exploit appears under multiple operators

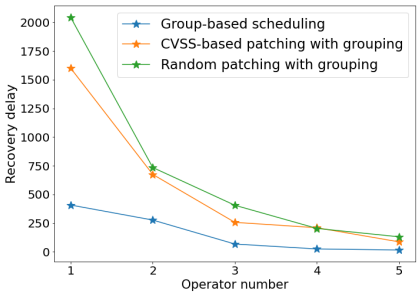

Fig. 19. Recovery delay under multiple operators

asset group, we determine their patch order based on their CVSS scores. We did not add grouping-awareness to time-based patching since it is hard to reasonably determine the time when an asset group becomes to have exploit.

The results are shown in Fig. 14, 15 and 16 when there is only one operator. It can be seen from Fig. 14 that our method performs much better than the other two grouping-based methods. The total amount of risk can be reduced from 685,985 to 262,445 when compared with the CVSS-based patching with grouping method, and from 913,641 to 262,445 when compared with the random patching with grouping method. Fig. 15 shows that our method can patch 296 vulnerabilities before they have exploits, while the other two can only patch 249 and 228 vulnerabilities respectively. Fig. 16 shows that our method exposes the system to exploits for the shortest time, which is 1,193 hours and 1,636 hours less than the other two methods respectively.

The comparison with the other two grouping-based methods under multiple operators is shown in Fig. 17, 18 and 19. The results show that our group-based scheduling method performs much better than the other two methods under multiple operators.

V. Conclusion

In this paper, we proposed a scheduling solution to schedule the patching order of vulnerabilities to minimize the risk that vulnerabilities pose to a system. We first predicted the vulnerabilities' dynamic probability of exploit over time and computed the dynamic risk for each vulnerability based on the predicted probability and its impact score on the system. Then we formulated the patching scheduling problem as optimization problems in two ways - baseline scheduling and group-based scheduling. The evaluation based on a real dataset showed our solution can significantly reduce the risk compared with several other patching methods.

Acknowledgment

This work is supported in part by NSF under award number 1751255. This material is also based upon work supported by the Department of Energy under Award Number DE-OE0000779.

References

[1] U.s. national institute of standards and technology. https://nvd.nist.gov/vuln/full-listing.
[2] Forum of incident response and security teams, cvss. "https://www.first.org/cvss/v2/guide".
[3] U.s. national institute of standards and technology, national vulnerability database. https://nvd.nist.gov/vuln.
[4] C. Fruhwirth and T. Mannisto, "Improving cvss-based vulnerability prioritization and response with context information," in *int'l Symp. on Empirical Software Engineering and Measurement*, 2009, pp. 535–544.
[5] J. B. Hong, D. S. Kim, and A. Haqiq, "What vulnerability do we need to patch first?" in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 684–689.
[6] G. Gianini, M. Cremonini, A. Rainini, G. L. Cota, and L. G. Fossi, "A game theoretic approach to vulnerability patching," in *Int'l Conf. on Information and Communication Technology Research*, 2015, pp. 88–91.
[7] L. Maghrabi, E. Pfluegel, L. Al-Fagih, R. Graf, G. Settanni, and F. Skopik, "Improved software vulnerability patching techniques using cvss and game theory," in *International Conference on Cyber Security And Protection Of Digital Services*, 2017, pp. 1–6.
[8] A. Alshawish and H. de Meer, "Risk-based decision-support for vulnerability remediation in electric power networks," in *ACM International Conference on Future Energy Systems*, 2019, pp. 378–380.
[9] Tanable. (2020) https://www.tenable.com/cyber-exposure/platform.
[10] F. Zhang, P. Huff, K. McClanahan, and Q. Li, "A machine learning-based approach for automated vulnerability remediation analysis," in *IEEE Conference on Communications and Network Security (CNS)*, 2020.
[11] F. Zhang and Q. Li, "Security vulnerability and patch management in electric utilities: A data-driven analysis," in *The 1st Radical and Experiential Security Workshop (RESEC)*, 2018.
[12] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 105–114.
[13] M. Edkrantz and A. Said, "Predicting cyber vulnerability exploits with machine learning." in *SCAI*, 2015, pp. 48–57.
[14] B. L. Bullough, A. K. Yanchenko, C. L. Smith, and J. R. Zipkin, "Predicting exploitation of disclosed software vulnerabilities using open-source data," in *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, 2017, pp. 45–53.
[15] C. Sabottke, O. Suciu, and T. Dumitraș, "Vulnerability disclosure in the age of social media: exploiting twitter for predicting real-world exploits," in *USENIX Security Symposium*, 2015, pp. 1041–1056.
[16] N. Tavabi, P. Goyal, M. Almukaynizi, P. Shakarian, and K. Lerman, "Darkembed: Exploit prediction with neural language models," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.
[17] Cvss. [Online]. Available: https://www.first.org/cvss/
[18] Impact score. [Online]. Available: https://www.first.org/cvss/v2/guide
[19] Vulners database. [Online]. Available: https://vulners.com/
[20] U.s. national institute of standards and technology, common weakness enumeration (cwe). https://nvd.nist.gov/vuln/categories.
[21] Tf-idf. [Online]. Available: http://www.tfidf.com/
[22] Job shop scheduling. https://en.wikipedia.org/wiki/Job_shop_scheduling.
[23] Or-tools. [Online]. Available: https://developers.google.com/optimization