Attacker-Defender perspectives for Dynamic Patch Prioritisation in EV Charging Networks

Roland Plaka[1], Mikael Asplund[1], and Simin Nadjm-Tehrani[1]

Department of Computer and Information Science, Linköping University, Sweden
{roland.plaka, mikael.asplund, simin.nadjm-tehrani}@liu.se

Abstract. Modern critical infrastructures, such as Electric Vehicle Charging Infrastructures (EVCI), are increasingly vulnerable to sophisticated cyberattacks due to their growing complexity and digital interconnectivity. To effectively manage these vulnerabilities, strategic prioritization is essential. Patching security vulnerabilities in large and critical systems is a challenging process that involves multiple entities making interdependent decisions. Given the ongoing patch management problem, it is important to identify the available patches. Building on earlier work that prioritizes patches by identifying the attacker router with highest risk, in this work, we add an attacker-defensive perspective. Our enhanced dynamic patch management approach automatically integrates patch detection and patch collection, to compare the cost-benefit attributes for various patches. We introduce a new metric named patch cost as a function of impact, size, and effort of patching a vulnerability, as a means of deciding the priority compared to other patches. Results show significant improvements in resource allocation efficiency, thus revealing efficient patching in central hosts of the network topology.

Keywords: Vulnerability, Patch, Prioritization, Topology

1 Introduction

As the adoption of electric vehicles accelerates, the charging ecosystem is becoming increasingly intertwined with the energy infrastructure and dependent on digital technologies. The ongoing digitalization of charging services brings notable advantages in terms of efficiency and functionality; however, it simultaneously introduces new attack surfaces that adversaries may exploit.

For example, Kern et al. [9] describe attack scenarios such as manipulation of demand (MAD) and false data injection (FDI), both of which can jeopardize power grid stability. Similarly, Vailoces et al. [23] analyzes weaknesses in electric vehicle supply equipment (EVSE), highlighting consequences that range from physical damage to privacy breaches. In another case, Alcaraz et al. [1] evaluates public charging infrastructure integrated with microgrids, applying STRIDE and DREAD to identify vulnerabilities and potential threats.

Further work has examined the backend itself. In particular, Sarieddine et al. [17] conducts a comprehensive security assessment of the Open Charge Point Protocol (OCPP) server implementation, demonstrating that attackers could impersonate legitimate charging stations. Such adversary-in-the-middle capabilities may allow malicious devices to register with the backend, disrupt normal charging operations, and escalate into more severe impacts on connected services and grid stability.

Because EVCI deployments continue to evolve, the number of reported vulnerabilities is rising. Addressing all of them is rarely feasible: even when patches are available, they cannot always be applied immediately. Evidence from industry underscores this challenge. A study by Kenna Security and Cyentia shows that the median monthly remediation rate across organizations was only 15.5%, and one quarter of the studied firms patched fewer than 6.6% of their known vulnerabilities. Such numbers underline the complexity of patch management, particularly in ecosystems characterized by multiple stakeholders and the prevalence of small and medium-sized enterprises.

At the same time, the complexity of modern systems makes it difficult to test every attack scenario comprehensively. Dependencies between hardware, software, and third-party libraries can harbor latent flaws that open new attack vectors. The sheer volume of open vulnerabilities further complicates systematic security analysis, reinforcing the need for prioritization strategies that can separate the most critical issues from the less urgent.

This paper is a substantial extension of an earlier work presented at a conference, Plaka et al. [15]. For the sake of self-continment, we will present the earlier work, including the method and the corresponding tool that is being extended, and introduce the extension of the method in Section 3.3. In risk management terms, prioritization is an essential activity: By identifying which vulnerabilities most endanger overall system security, practitioners can allocate limited resources toward those fixes that most effectively reduce exposure.

The contributions of this paper are as follows:

1. Extension of a *Dynamic Patch Management (DPM)* approach into DPM+ for vulnerability instance prioritization, aiming to minimize cumulative system risk using a cost-benefit analysis framework that incorporates host centrality and aggregated patch priority.
2. An empirical investigation into vulnerabilities, exploits, and remediation measures affecting EV charging systems.
3. Development of an automated tool named PatchD that detects patch availability from vulnerability databases, reducing the need for manual search.
4. An evaluation of DPM+ and PatchD, showing the performance comparison in regard with baseline works+.

The remainder of this article is organized as follows. Section 2 introduces our methodology in detail, including prerequisites and assumptions, refinements such as the notion of host centrality, and the newly proposed automated patch detector tool. Section 3 presents the design of the empirical study and summarizes the data gathered on vulnerabilities, exploits, and patches. Section 4

evaluates the proposed approach on a representative use case and compares the results with existing strategies. Section 5 reviews related work, and Section 6 concludes with a discussion of findings and future directions.

2 Dynamic Patch Management+

In this section, we describe our extended approach, called *Dynamic Patch Management+ (DPM+)*. The objective of DPM+ is to prioritize vulnerabilities based on the risk they pose, considering both their potential system impact and likelihood of exploitation, and to patch those identified as most critical. An overview of the method is provided in Figure 1.

DPM+ takes as input a system model that represents both the network topology and the set of identified vulnerabilities. This model includes standard vulnerability information, such as impact ratings and exploitability metrics. Using these inputs, the DPM+ process is organized into four steps:

1. **Risk calculation**: The initial step estimates the system's baseline risk. We introduce a unified metric that blends vulnerability impact with dynamic exposure information. In this step we integrate the notion of host centrality into system risk calculation.
2. **Attack simulation**: In this step, our attacker simulation algorithm identifies the shortest attack paths through the network, starting from defined entry points.
3. **Patch detection**: In this step, the method includes detection of the available patches of the vulnerabilities collected. In addition we provide a tool that performs the proposed method.
4. **Patch prioritization**: Finally, vulnerabilities are ranked according to their occurence on the most threatening attack paths. The patching process focuses on addressing those vulnerabilities that contribute the most significant risk to the system as a whole.

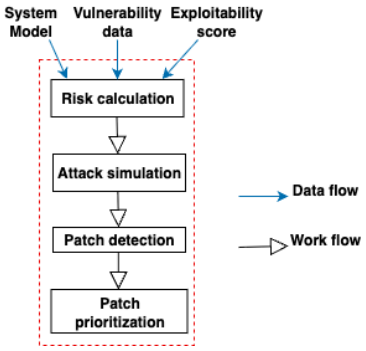

Fig. 1: Overview of the Dynamic Patch Management (DPM+) method (adapted from [15])

In the remainder of this section, we first describe the required system model and input assumptions. We then dedicate a subsection to each of the three phases of the DPM+ method, providing the details of their design and operation.

2.1 Prerequisites

System model: In our method, the topology of a system and its set of known vulnerabilities is defined using a graph. The *system* S is a tuple of *hosts* H, *links* L, *vulnerabilities* V and mapping c as follows :

$$\boldsymbol{S} = \langle H, L, V, c \rangle$$

- $\boldsymbol{H} = \{h_1, h_2, \ldots, h_n\}$: is the set of hosts in the system
- $\boldsymbol{L}$: is the set of logical links connecting the hosts; a link is a pair $l = \langle h_i, h_k \rangle, h_i, h_k \in H$ and $i \neq k$
- $\boldsymbol{V} = \{v_1, v_2, \ldots, v_m\}$: is the set of vulnerabilities in the system
- $\mathbf{c}(h, v)$: denotes that a host $h \in H$ contains the vulnerability $v \in V$.

We define the following predicates over **S**:

- *hasInitialAccess*(h): denotes that a host $h \in H$ is initially accessible by an adversary.
- *hasExploit*(v): denotes that a vulnerability $v \in V$ has an available exploit
- *isEntryPoint*(h): denotes that a host $h \in H$ is a potential entry point for an attacker. This is true when *hasInitialAccess*(h) = True and $c(h, v)$ = True for some vulnerability $v \in V$ such that *hasExploit*(v) = True. This means that each entry point must have at least one vulnerability with an available exploit.

Vulnerability data: Our approach assumes access to the Common Vulnerabilities and Exposures (CVE) database (or some similar database). Each CVE entry corresponds to a specific weakness in hardware or software components that an attacker may leverage. To measure the consequences of exploitation, we rely on impact scores that describe how a vulnerability affects system properties. Following the CVSS specification [6], we consider the three classical security dimensions—confidentiality, integrity, and availability—each of which can take the qualitative values *high*, *low*, or *none*. These categories are also mapped to numeric values, enabling both qualitative and quantitative assessments.

Exploitability score: Although CVE records provide exploitability information, the CVSS exploitability metric is static: once assigned, it remains unchanged, even as new exploits appear or system conditions evolve. This limitation highlights the need for a more dynamic measure that accurately reflects the likelihood of exploitation in real-world contexts.

For this purpose, we integrate the *Exploit Prediction Scoring System (EPSS)* [7]. EPSS estimates the probability that a vulnerability will be exploited within the next 30 days, producing a value in the range $[0, 1]$, where higher values correspond to greater risk of exploitation. Incorporating EPSS enables us to focus

not only on vulnerabilities with high impact but also on those most likely to be targeted.

In practice, we generate EPSS-based exploitability scores using the open-source tool Tesorion available by T-CERT[1]. This tool assists security and incident response teams by enriching CVE data with predictive exploitability metrics, making it easier to prioritize vulnerabilities that represent the most pressing threats.

2.2 Risk calculation

The first step of the DPM+ process is to establish the baseline system risk level. This number is calculated using the notions of *frequency* and *vulnerabilityRisk* as follows.

The frequency of a vulnerability $v \in V$ equals the number of hosts h for which $c(h, v)$ holds, divided by the total number of hosts in the system as shown in Equation 1:

$$Frequency(v) = \frac{1}{|H|} \sum_{h \in H} c(h, v) \tag{1}$$

vulnerabilityRisk: denotes the risk associated with a single vulnerability and is calculated as in Equation 2:

$$\text{vulnerabilityRisk}(v) = \text{EPSS}(v) \times \text{Impact}(v) \tag{2}$$

Quantified values of impact used in the calculation are inherited from [6] and are illustrated as follows:

- *High Impact*: A significant breach of any security property. The numerical value for this level is 0.56.
- *Medium Impact*: A minor compromise of any security property. The numerical value for this level is 0.22.
- *No Impact*: No considerable impact on security properties. The numerical value for this is 0.

Finally, the ***systemRisk*** is a quantitative value from 0 to 1 representing how much a given system is exposed to attacks based on the frequency of vulnerabilities and their associated risks. The *systemRisk* is calculated as in Algorithm 1 (line 2-7).

The estimation of the total system risk is obtained by summing up the contributions from all vulnerabilities. In our earlier work [15], we simplified the calculation of the risk by considering the exposure of hosts and assumed that all hosts are subject to the same risk due to known vulnerabilities. In the current extension of the approach, DPM+, we calculate host risk by associating weights to various hosts to signify that some hosts have a bigger impact on the system risk than others. In addition, we use the notion of *host centrality* (CT) which we explain further in Section 2.3.

[1] https://github.com/Tesorion/vulnerability-explorer

Algorithm 1: calculateSystemRisk

Require: Set of hosts H, Set of vulnerabilities V, function denoting vulnerable hosts c, normalized host centrality CT(h)
Ensure: systemRisk $sysRisk$

$sysRisk \leftarrow 0$
for all hosts $h \in H$ **do**
  $hostRisk \leftarrow 0$
  for all vulnerabilities $v \in V$ **do**
    $frequency \leftarrow \frac{1}{|H|} \sum_{h \in H} c(h, v)$
    $epss \leftarrow EPSS(v)$
    $impact \leftarrow impactScores(v)$
    $vulnerabilityRisk \leftarrow epss \times impact$
    $hostRisk \leftarrow hostRisk + (frequency \times vulnerabilityRisk)$
  end for
  $sysRisk \leftarrow sysRisk + \mathrm{CT}(h) \times hostRisk$
end for
return $sysRisk$

2.3 Weights via Host Centrality

Here, we extend our earlier work [15], by incorporating the notion of host centrality (CT). Although our earlier model treated all hosts equally, in practice, some hosts are more critical than others because not all hosts contribute equally to system risk; some serve as critical bridges in the network and are more impactful if compromised. An attacker who compromises such hosts can more easily reach other parts of the network.

Host centrality is measured as the normalized centrality of each host in the network graph, with values scaled between 0 and 1. A host with high centrality lies on many shortest paths generated by the DPM+ search algorithm, and securing such hosts can significantly reduce the attacker's ability to move laterally. In the context of patching, the protection of hosts with higher centrality may stop an attacker from traversing significant portions of the network.

Incorporating host centrality thus provides a topology-aware extension, enabling more effective identification of vulnerabilities and hosts that attackers could exploit.

Eq. (3) is the normalized centrality, where CT(h) is the normalized centrality of host h, n is the total number of hosts in the network, σ_{st} is the number of shortest paths from host s to host t, $\sigma_{st}(h)$ is the number of those paths that pass through host h. The sum is over all distinct pairs (s, t) with $s \neq h \neq t$.

$$\mathrm{CT}(h) = \frac{2}{(n-1)(n-2)} \sum_{\substack{s,t \in H \\ s \neq h \neq t}} \frac{\sigma_{st}(h)}{\sigma_{st}} \tag{3}$$

This weighting is later combined with patch cost analysis in Section 4, allowing us to consider both the benefit of patching(risk reduction) and the operational cost of patch deployment.

2.4 Attack simulation

In this subsection, we outline a potential strategy that an adversary might employ to move through the target system. To illustrate this process, we describe our attacker algorithm in detail. It is important to emphasize that the algorithm is not intended as a comprehensive representation of attacker behavior; rather, it serves as a tool to identify the most critical vulnerabilities within the system.

We consider an attacker that is either rational or destructive, with initial access limited to predefined entry points. A rational attacker is a term used in game theory and moving target defense, and is defined as an attacker who, knowing the pay-offs to his possible choices, selects the one with the highest pay-off. A destructive attacker tries to damage as many hosts as possible, even if it is not optimal for stealth or persistence. In our system model, we assume that the attacker's objective is to compromise as many hosts as possible. Progression is constrained, however: if the attacker encounters a host that is either patched or free of vulnerabilities, further movement from that point is blocked.

To capture this process, our approach models the attacker's movement using a breadth-first search (BFS). BFS is a well-known graph traversal algorithm that systematically explores all nodes at the current depth before proceeding to the next. In our setting, the root node corresponds to an entry point, and the algorithm expands outward to all neighboring hosts. The traversal continues until all reachable hosts have been examined, while also accounting for cycles to ensure termination.

This systematic exploration enables the attacker model to enumerate all reachable vulnerable hosts and to uncover every possible shortest attack path through the system. In doing so, the method highlights the vulnerabilities that, if left unpatched, could facilitate the widest spread of compromise.

The algorithm has three phases:

- **Queue initialization**: The algorithm begins by initializing a queue with a chosen entry point. Each item in the queue consists of the current host along with the path followed to reach it.
- **Neighbor traversal**: At each step, an element is removed from the queue and its neighboring hosts are examined. Any neighbor that has not yet been visited and is not patched is appended to the queue, with the corresponding path extended accordingly.
- **Termination**: This process is repeated until no further nodes can be reached from the entry point. The output is the complete set of paths that an attacker could potentially follow starting from the initial entry node.

We model an attacker's strategy to traverse the network. As illustrated in Attacker Simulation Algorithm 2, the attacker explores the network from a specified entry point e, and prioritizing hosts with high connectivity and exploitability. The algorithm initializes three variables: R (reached hosts), Q (queue) and P (path). R denotes a set to track visited hosts, starting with an entry point e *(line 2)*. Q contains a queue of pairs, where each pair consists of a host to visit,

Algorithm 2: Attacker Algorithm

```
   Input: S System model, e Entry Point, V Vulnerabilities
   Output: P Set of paths
 1 Initialize:
 2 R ← {e}; // Reached hosts
 3 Q ← [(e, [e])]; // Queue of hosts to visit, with path from entry point
 4 P ← ∅;
   // Set of completed attack paths
 5 while Q ≠ [] do
 6     (h, p) ← Q.pop();
       // Extract the current host and the path taken to reach it
 7     C ← []; // Candidate hosts to visit next
 8     for each n ∈ S.neighbors(h) do
 9         if n ∉ R then
10             conn ← getConnections(n, S);
11             EPSS ← getEpss(V, n);
12             C.append((n, conn, EPSS));
               // Add candidate hosts with their connection and EPSS
                  data
13     if C = [] then
14         P.add(p);
           // Add the current path to the set of paths if no new
              candidates
15     sort(C);
       // Sort the candidate hosts based on chosen criteria
16     while C ≠ [] do
17         next ← C.pop();
           // Choose the next candidate host
18         R.add(next);
           // Mark the host as reached
19         Q.append((next, p + [next]));
           // Append the next host to the path and continue exploring
20 return P;
   // Return the set of all paths explored by the attacker
```

together with the path leading to the host from the entry point (line 3). P is a set that stores all distinct paths explored by the attacker (line 4).

The main loop (line 5) processes elements from Q until the queue is empty. For each iteration, the algorithm pops a path from Q, and for each neighbor of the current host, evaluates unvisited neighbors based on the number of connections and the EPSS value. These properties are appended to the list of candidates C (line 6-14).

The sort(C) function, called on line 15, sorts the candidates first by the number of connections (descending), as the attacker prefers hosts with more connections for lateral movement. Since the attacker performs reconnaissance

and discovery to understand the target environment and they are able to identify vulnerabilities.

Once they have sufficient information, they can evaluate and decide which part of the system to attack. Then if two hosts have the same number of connections, the attacker compares their average EPSS scores. A higher EPSS indicates a greater likelihood of vulnerability exploitation. If the EPSS scores are equal, the attacker selects randomly.

The algorithm begins by selecting a candidate from the available options (line 16) . If a candidate is found, (line 17), it is added to the set R, which records the hosts that have been reached, preventing the attacker from revisiting previously explored hosts (line 18). The selected host is then appended to the path sequence, and a new path is created by extending the previous one with this host. The updated path is pushed back into the queue for further exploration (line 19).

The attacker continues exploring and extending paths until there are no more valid candidates or all reachable hosts have been explored. If no candidates are found, the current path is terminated and added to the list of paths. Once all paths have been explored, the algorithm returns the set of distinct paths, line 20, representing the different routes the attacker could take through the system.

2.5 Patch detection

Vulnerabilities are usually unpatched when initially published. Workarounds or hotfixes may offer temporary remediation until an official patch or upgrade is released. The less official and permanent a patch, the higher the vulnerability score. Remediation Level (RL) is a quantitative metric defined in [6] and is an important factor for prioritization.

Measuring the RL is still difficult, as many CVE databases do not provide this information and operators must invest significant manual effort. Therefore, we introduce a new methodology to help researchers and security analysts automatically extract patch information.

From the CVSS specification [6], the remediation levels are summarized in Table 1:

Table 1: CVSS Remediation Level (RL) Values and Descriptions

RL	**Value**	**Definition**
Not Defined	1	Insufficient information to choose one of the other values.
Unavailable	1	No solution available or it is impossible to apply.
Workaround	0.97	An unofficial, non-vendor solution is available. Users create their own patch or apply steps to mitigate the vulnerability.
Temporary Fix	0.96	A temporary official fix is available, such as a hotfix, tool, or workaround provided by the vendor.
Official Fix	0.95	A complete vendor solution is available; an official patch or upgrade is issued.

Since there are no clear guidelines on how to automatically map each remediation level, we introduce a keyword-based detection approach. The goal is to determine whether a patch is available, and if so, classify it into one of the five RL categories.

The framework uses both structured database fields (e.g., `recommended`, `countermeasure`, `mitigation`) and unstructured text extracted from vulnerability advisories. Relevant keywords include:

- **Not Defined (X)**: None; fallback if no patterns match.
- **Unavailable (U)**: no fix, not fixed, unsupported, not available.
- **Workaround (W)**: workaround, mitigation, disable, firewall.
- **Temporary Fix (T)**: temporary fix, hotfix, interim solution.
- **Official Fix (O)**: official fix, patch, upgrade, update.

The methodology consists of two phases: 1. **Textual extraction** scans free text for semantically meaningful phrases (e.g., "fixed in version 2.3.1", "no patch available"). 2. **Structural extraction** parses explicit API JSON fields (e.g., "recommended countermeasure").

The consolidated remediation snippet is then mapped to one of the five CVSS RL categories using a deterministic, rule-based classifier.

PatchD tool implementation To operationalize this methodology, we developed *PatchD*, a Python-based framework for patch detection. PatchD automates the collection and classification of remediation levels from the VulDB platform.

PatchD takes as input a list of CVE identifiers and an API key. It uses the Playwright library to automate browser sessions and query the VulDB API. After authentication, each CVE is submitted to retrieve remediation information. The tool fetches both the API JSON fields and the corresponding HTML content for more complete extraction.

We use BeautifulSoup [19] for HTML parsing and structure recognition. The combined textual and structural extraction modules identify remediation snippets, which are then classified into RL categories using regular expressions and keyword matching. The classifier applies rules in deterministic order to ensure consistency across vulnerabilities.

We now proceed to the final phase of our method, where we discuss the extended approach used for patch prioritization.

2.6 Patch prioritization

In this subsection, we explain how patch prioritization is done including cost and benefit analysis. Differently from the earlier work, where there was a defender algorithm that detected the riskiest path with maximal risk reduction, in the current work we adjust the patch prioritization step by introducing the calculations of cost and benefit per patch and host.

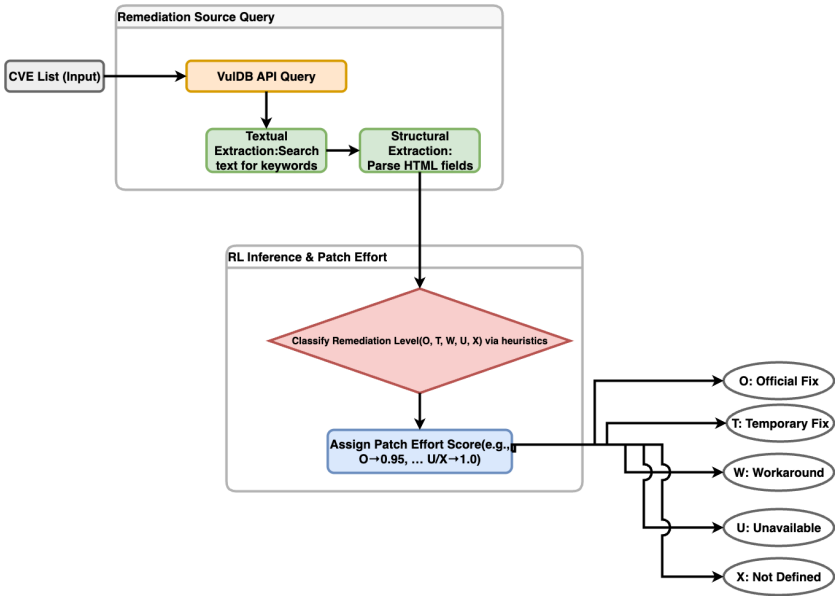

Fig. 2: PatchD tool framework

Cost-benefit analysis for patching Since organizations face an overwhelming number of vulnerabilities to patch, it is hard to know which ones to prioritize. Developers and software vendors struggle to deploy fixes quickly while making sure the patches are sufficient. Patching may be a risky operation for several reasons, including the fact that patches often affect numerous critical systems, libraries, and other software used by various applications. Patches also tend to be large and complex operations with even minor configuration variances that can cause drastically different results.

Patches must be installed on time, balancing the risks of potential attacks and the time required for proper patch testing, while effectively managing organizational constraints and resources. These factors can make the success rate for patch changes much lower than that of other changes, thus requiring more comprehensive testing.

Therefore to prioritize which patch to install first we calculate the prioritization values using the Eq. (4) which vary from [-1,1]. For ranking purpose, we define a priority score for each patch p as below, where $\alpha \in [0, 1]$ is a weighting factor, B denotes the benefit of applying the patch (measured as risk reduction when patching a particular vulnerability), and C which represents the normalized patch cost. In our evaluation, we set $\alpha = 0.5$.

$$\text{Prio}(p) = \alpha \cdot B - (1 - \alpha) \cdot C \tag{4}$$

In our formulation, the *benefit* of a patch is defined as the reduction in overall system risk that results from applying it. More formally, for a given patch p, the

benefit corresponds to the marginal contribution of p to the risk function, i.e., the difference between the system risk before and after its application:

$$\mathrm{B}(p) \;=\; \mathrm{systemRisk}_{\mathrm{before}} \;-\; \mathrm{systemRisk}_{\mathrm{after}} \tag{5}$$

We model the *cost of patching* to capture the practical burden on system operators when mitigating vulnerabilities. Patch cost is not limited to financial expenditure; instead, it reflects the combined technical and operational effort of deploying a patch in a live cyber-physical infrastructure such as electric vehicle charging networks. Assuming that system operators have a testing environment and know the time it takes to deploy a patch, we consider the following measurable dimensions:

1. **Vulnerability impact score**: Vulnerabilities of higher impact typically require more testing and validation before deployment. We normalize CVSS impact sub-scores into the interval $[0, 1]$, assigning values of 0.56 for high, 0.22 for medium, and 0.1 for low severity.
2. **Patch size**: Larger patch binaries imply longer download, verification, and installation times. We normalize the patch size by dividing by the maximum patch size observed in our data set (e.g. 5.4 GB), which yields values in [0,1].
3. **Installation effort**:The operational mode of patching strongly affects cost. Manual patching requires technician involvement (e.g., local EVSE updates), while remote patching is automated via the CSMS. We assign normalized values of 1.0 (manual) and 0.5(remote).

This interpretation quantifies the added value of each patch in terms of its direct impact on risk reduction, thereby supporting a principled prioritization of patches.

The overall patch cost is then calculated as a weighted combination of these factors, $I(v)$, $S(v)$, and $E(v)$ are the normalized impact, size, and effort for vulnerability v, and w_I, w_S, w_E are weights reflecting their relative importance.

$$\mathrm{C(v)} = w_I \cdot I(v) + w_S \cdot S(v) + w_E \cdot E(v) \tag{6}$$

The formulation balances benefit and cost equally. A patch with a higher risk reduction benefit and lower cost will obtain a higher priority value. Positive scores indicate that the expected security gain outweighs the associated effort, whereas negative scores signal that the patch is relatively costly compared to the benefit achieved.

After being able to compute patch priority, then we extend our calculation to identify the hosts that need to be patched first. The calculation allows operators to prioritize which host in the topology needs to be patched first. We introduce the notion of *host priority*. *Host priority* is computed using Eq. (7), and is derived by integrating two components: *host centrality* denoted as Centrality(h) and the *patch priority* denoted as Prio(p_i), representing the normalized priority value of patch p_i associated with host h, where m is the number of patches applicable to host h.

$$\text{HostPriority}(h) \;=\; \text{Centrality}(h) \times \Big(\sum_{i=1}^{m} \text{Prio}(p_i) \Big) \tag{7}$$

The outcome of the above formulation is that we can quantify each host in the network with a score, which says which hosts are the most critical to patch first, so in practice a host with high centrality and high patch priorities will get the highest value, meaning patching it yields the largest security benefit for the overall system.

3 Empirical Study

This section describes the methodology adopted in our empirical study, with a focus on how data concerning vulnerabilities, exploits, and vendor patches were collected and analyzed. These datasets represent the essential inputs to our approach, forming the basis for its application and evaluation. We further examine the nature of the identified vulnerabilities, the corresponding exploits, and the patches available, as these elements are central to our method.

Although prior research has investigated various aspects of Electric Vehicle Charging Infrastructure (EVCI) security, to the best of our knowledge this is the first work to empirically analyze vulnerabilities, exploits, and patches specifically in relation to EVCI assets. By mapping known CVEs to their exploitability and potential impacts, we highlight the components most at risk and offer insights how adversaries might attempt to compromise EVCI systems.

This section provides the foundation for the remainder of the paper, as the empirical data gathered here directly supports the analysis in subsequent sections. Our study is grounded in an architecture inspired by Alcaraz et al. [1], which models an electric vehicle charging infrastructure embedded within a microgrid, thereby ensuring that the evaluation remains representative of real-world EVCI deployments.

This section is critical for the rest of the paper as it provides the foundational data necessary for our method, which is directly applied in the subsequent analysis.

3.1 Vulnerability collection

We start by collecting EVCI-relevant data from various sources such as search engines and vulnerability databases using keywords like "OCPP," "charging stations," "CSMS," "EVSE," hardware/software components, and product names. After reviewing all the possible hits generated from our keywords, we removed the duplicates, resulting in 46 vulnerabilities related to EVCI with assigned CVE IDs. The severity scores of the selected vulnerabilities range from 4.3 to 9.8.

The fact that we only found 46 EVCI-related vulnerabilities shows that they are underreported. Table 2 shows the EVCI components ordered by the number of reported vulnerabilities related to each component. The analysis of this

table shows that charging stations (CSs), charging station management systems (CSMSs), and charging station controllers (CSCs) appear to be the most frequently targeted assets. They commonly suffer from firmware or software vulnerabilities, making them the primary entry points to the system architecture.

Table 2: Vulnerability distribution in EVCI (adapted from Plaka et al. [15])

Asset Type	**Known CVEs**
Charging station(CS)	21
Charging station controller(CSC)	6
Automation server(AS)	1
Electric Vehicle Supply Equipment (EVSE)	1
Charging station management system (CSMS)	6
Industrial network switches(SW)	5
Load Management System (LMS)	2
Power meter(PM)	1
Electric Vehicle(EV)	1
Industrial Gateway(GW)	1
Remote telemetry system(RTS)	1

3.2 Exploit collection

As discussed earlier, our primary focus is on the exploitability of the collected vulnerabilities. Hence, we investigate and find the available exploits in the wild that attackers can use to hack the EVCI infrastructure.

Yoon et al. [30] proposed a framework to detect exploits in the wild by collecting exploit data from sources like Exploit-DB, Github, and CISA, which provides Known Exploited Vulnerability (KEV) information. Building on their work, we extend this list of sources by including additional exploit databases such as Packetstorm, Coalition, Sploitus, and 0day.today. By querying these databases—comprising both the original sources from Yoon et al. [30] and the additional ones we identified — we detect 11 available exploits for the collected CVEs using CVE IDs and the affected products as search entries.

Table 3 shows the identified exploits aligned with their exploitability scores. From these two tables, we can see that for 46 vulnerabilities, there are 10 available exploits, which means 1 in 4 detected vulnerabilities can be immediately exploited with the help of online databases to damage the EVCI infrastructure.

3.3 Patch collection

The patching process requires continuous monitoring of the CVE patches, especially those released by vendors. Keeping track of the recent patches is time-consuming because one must monitor vendor websites, GitHub releases, or other communication channels. These channels are not fixed and can vary from vendor to vendor.

Table 3: Published exploits in EVCI (adapted from Plaka et al. [15])

CVE	Asset	Exploit	CVSS	EPSS	Source
2020-8006	CS	Buf. Overflow	9.8	0.1	PS
2018-12634	CSC	Cred. Disclosure	9.8	0.95	Sploitus
2023-28343	LMS	Comm. Inject.	9.8	0.95	EDB
2022-22808	LMS	CSRF	8.8	0.11	CESS
2016-5809	PM	CSRF	8.8	0.22	EDB
2021-34591	CSC	Priv. Esc.	7.8	0.04	CESS
2023-49955	CSMS	DoS	7.5	0.05	CESS
2021-22708	CS	Auth. Bypass	7.2	0.1	PS
2016-2278	AS	Multiple Vuln.	7.2	0.02	0day
2022-0878	CST	DoS	6.5	0.04	CESS
2020-15912	EV	Relay Attack	6.5	0.1	PS

In our initial empirical study, we did not find many available patches (13 in total) published by the vendors. This fact shows that not all vendors share their patches on their websites; possibly because they use subscription-based sharing with their clients via e-mails or other private channels. We detected the following patches from one vendor only on their website of security notifications [4]. The vendor provides clear instructions on performing the patching process and proposes mitigations. More details about various advisories can be found in a related article on ICS advisories [18].

In this paper, we extended our investigation by reviewing the literature and the state-of-the-art, with the goal to find more available patches. Previous works around patches fall into two threads: (i) patch detection, finding the precise fixing commits for disclosed vulnerabilities, and (ii) patch prioritization, deciding what to fix first and executing patch workflows.

Patch datasets and automated localization: PatchDB builds a large security patch dataset by combining NVD-linked patches, GitHub commits in the wild, improving ML-based patch identification [24]. Beyond datasets, several systems automate the mapping from CVEs to fixing commits. *PatchScout* locates patches over repository commits using vulnerability–commit correlation features (ID, location, type, text), achieving high coverage with low manual effort [22].Another work, *VCMatch* extends the idea with semantic features. *Tracer* constructs a multi-source reference network (NVD, vendor advisories, GitHub, etc.) and traces patches across sources, covering more CVEs than heuristic baselines while complementing industrial vulnerability databases [26].

Risk-driven prioritization and automation: ILLATION learns adversary patterns (motivation/ability) and tunes risk with network constraints (service presence and host reachability) via neurosymbolic reasoning to produce *network-specific* vulnerability rankings [31]. An empirical study of automated patch management proposes an executable workflow (pre-deployment review for cumulative/dependent patches, deployment/verification with error handling, and a feedback loop to prioritize after failures), showing reduced human effort and shorter remediation time [11]. A recent SLR on software security patch management highlights that research is focused on prioritization. In contrast, patch

information retrieval, testing, and post-deployment verification remain underexplored, underscoring the need for end-to-end human-in-the-loop automation [3].

Figure 3 emphasizes overall patch availability of 45 CVEs found by our automated tool PatchD, where 33 patches are released by official vendors and provided instructions such as software upgrade/update. For 6 vulnerabilities marked with T, there are temporal countermeasures such as using firewall. As indicated in the chart the tool did not detect any workaround at all. The remaining 6 CVEs has no mitigation known in the VulDB database. The numbers show that our tool is a promising concept that can be used to detect the available patches in vulnerability databases. Our limitation is that many trusted vulnerability databases tend to block the API requests when they are done with short time intervals, therefore we had to run the tool several times and wait for the output which sometimes it takes longer than expected. It is also possible to extend the search space by integrating other APIs in the tool, which can be interesting to investigate in a future work.

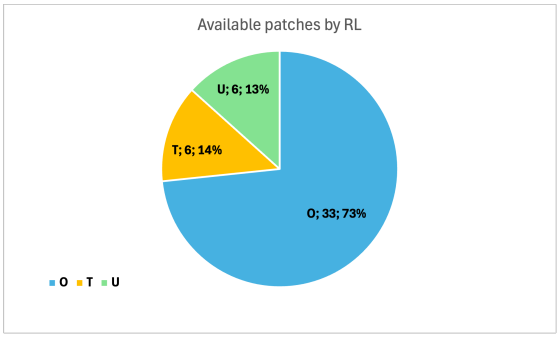

Fig. 3: Counts by remediation level (RL)

Table 4 gives an overview of the detected and collected patches and difference among results is remarkable.

Table 4: Comparison between manual and automated effort

Host with Patch	Manual detection	Automated detection
CS	11	19
LMS	2	2
SW	-	1
CSC	-	4
CSMS	-	5
EV	-	1
RTS	-	1

4 Evaluation

In this section, we evaluate the DPM+ method and compare its performance with 2 baseline approaches. We aim to assess the effectiveness of our patching strategy in reducing system risk. Specifically, we focus on how well DPM+ minimizes the total risk in the system after patching vulnerabilities along the attacker's most exploitable path. As defined in Section 2.3, host centrality weights are integrated into our prioritization model. In this section, we evaluate their impact.

4.1 Evaluation methodology

In our evaluation method, we develop a graph-based mesh topology to express the relationships between the EVCI infrastructure components, as shown in Figure 4. We model the system utilizing the NetworkX library[2]. NetworkX is a Python package mainly used for the creation and analysis of the structure, dynamics, and functions of complex networks. Our topology consists of nodes and edges where nodes (i.e., hosts) refer to the EVCI infrastructure assets. Edges represent the links between the hosts in our model. Every host has its name and additional properties. Utilising the empirical study knowledge, we assume that attackers can start exploiting from some of the entry points, e.g., "CSC," "CS", "CSMS", "LMS", and "SW", as they the components with the most vulnerabilities.

We compare DPM+ with three baseline methods. The first baseline for comparison is our earlier work DPM. Next is the one described by [13], which is the closest method we can compare DPM+ with. In their work, the authors assign weights to the links in the graph to represent the presence of security measures. For links connecting any type of component that does not have any protective measures in place, the authors assign weight 1 to the link. For the links connecting one graph component with a gateway and with no security measure in place, they assign weight 2. To ensure a fair comparison, we have adopted the same weighing scheme for the links in our graph, maintaining consistency with their approach.

To compare DPM+ with a third baseline, we apply Dijkstra's algorithm as the attacker algorithm, seeking the shortest attack path on a simpler model. In this case, each link is assigned a uniform weight of 1, and the attacker selects paths with the fewest hops between hosts, regardless of security measures. After finding the shortest paths, we apply our defender algorithm used in our earlier work to patch the vulnerabilities on the paths and calculate the overall system risk reduction.

4.2 Evaluation metric

The success of the compared methods is measured by the relative risk reduction metric in the system risk after patching. The system risk is quantified using an

[2] `https://networkx.org/`

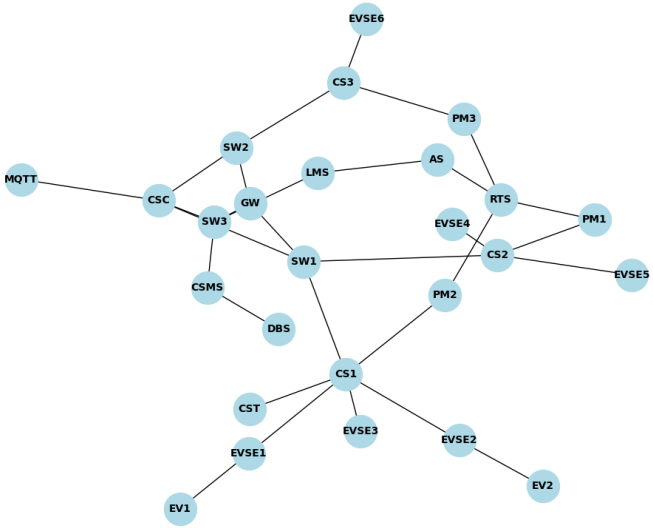

Fig. 4: EV charging infrastructure (adapted from [15])

initial score, which represents the vulnerabilities present in the system before any patches are applied. The formula for relative risk reduction is given by Equation (8) as follows:

$$\text{Risk Reduction } (\%) = \left(1 - \frac{R_{\text{post}}}{R_{\text{pre}}}\right) \times 100 \tag{8}$$

where R_{pre} represents the system risk before patching, and R_{post} represents the system risk after patching. This gives a value between 0% and 100%, where 100% represents complete risk reduction (i.e., no remaining risk), and 0% indicates no risk reduction.

In our experiments, we use the parametres shown at Table5

Table 5: Parameters for patch cost calculation

Parameter	Description
$I(v)$	Normalized impact of vulnerability v
$S(v)$	Normalized size of the patch for vulnerability v
$E(v)$	Normalized effort required to apply the patch for vulnerability v
w_I	Weight reflecting the relative importance of impact (0.4 in our experiments)
w_S	Weight reflecting the relative importance of patch size (0.3 in our experiments)
w_E	Weight reflecting the relative importance of patching effort (0.3 in our experiments)
$C(v)$	Overall patch cost for vulnerability v, re-normalized to [0, 1] for comparability

Finally, $C(v)$ is re-normalized to $[0, 1]$ to allow comparability across vulnerabilities. The formulation enables us to capture not only *what* needs to be patched (high-impact CVEs) but also *how costly* it is to do so under real operational constraints.

4.3 Results

In this section, we present the results of the evaluation. Table 6 shows the risk reduction when applying one of the four evaluated methods starting from a specific entry point. The results in DPM+ are calculated from a centrality-aware risk reduction perspective, while the remaining three methods are calculated from a centrality-blind risk reduction perspective. The entry points are all the hosts for which we found published exploits (Table 3). DPM+, along with the other three methods, generates different paths that are not guaranteed to be the same length or cover the same amount of vulnerabilities. We consider the riskiest path with the highest impact on security, integrating the centrality metric for hosts in DPM+, and then compare it.

The results in Table 6 are divided into two categories according to how DPM + compares with the other three methods: DPM+ shows a significant advantage in all 9 entry points, and in 3 entry points performs slightly worse than the other methods. For entry point (CS3), DPM+ achieves 93.82% in risk reduction, outperforming DPM, Petho et al. and Dijkstra. In the second category, DPM outperforms the other methods by a small margin. For the entry point (CSMS), DPM+ achieves 89.25%, compared to 67.97% of DPM, 62.76% of Petho et al. and Dijkstra 48.16%. However, in the entry point EV2, CS1, and CST case, the DPM + performance is worse than that of the other three methods.

We now proceed to investigate how the risk reduction increases with the number of patches increasing. We select cases where DPM+ outperforms the baseline methods with a significant risk reduction.

Entry	DPM+	DPM	Petho et al.	Dijkstra
CS3	**93.82%**	80.00%	74.16%	74.16%
AS	**97.67%**	79.35%	46.71%	46.69%
PM1	**87.52%**	74.38%	59.80%	59.78%
LMS	**88.67%**	65.98%	46.58%	46.77%
PM2	**84.33%**	60.86%	46.77%	46.77%
CSMS	**89.25%**	67.97%	62.76%	48.16%
EV2	**61.66%**	67.97%	62.36%	62.36%
CS1	**61.65%**	66.34%	60.68%	60.68%
CST	**61.65%**	66.34%	60.68%	60.68%
CSC	**88.01%**	65.84%	64.52%	64.52%
PM3	**92.19%**	65.67%	60.15%	60.15%
CS2	**87.72%**	59.85%	62.36%	62.36%

Table 6: Evaluation of DPM+,DPM, Petho et al., and Dijkstra for different entry points

Figure 5 presents the relative risk reduction for entry point (CS3) across four methods: DPM+, DPM, Petho et al., and Dijkstra. DPM+ demonstrates the most significant improvement in risk reduction, reaching a peak of 93.82%. Petho et al. and Dijkstra show a more gradual increase, with their maximum risk reduction at 74.16%. DPM+ outperforms the other three methods because it applies patches while considering host centrality, aiming to maximize risk reduction.

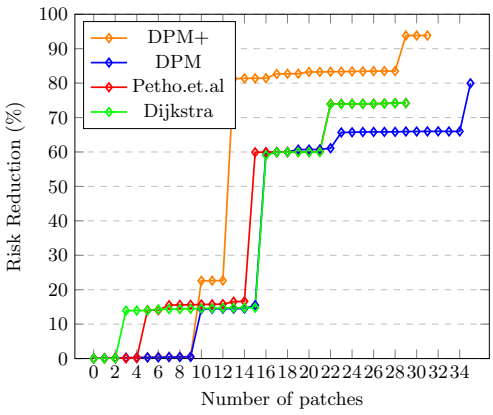

Fig. 5: Relative Risk Reduction for entry point CS3

Figure 6 illustrates the results for entry point CSMS. DPM+ achieves the highest reduction, peaking at 89.25% after applying 32 patches. DPM achieves

67.97%, Petho et al. reaches 62.3%, and Dijkstra also reaches 48.16%. Overall, DPM+ slightly outperforms the other methods.

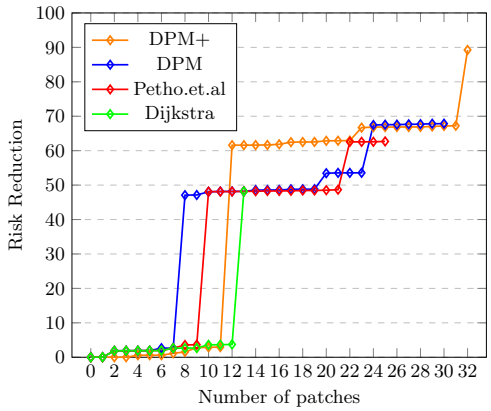

Fig. 6: Relative Risk Reduction for entry point CSMS

The findings for entry point CS2 are shown in Figure 7. DPM+ shows a significant increase in risk reduction, peaking at approximately 87.72% after 19 patches. DPM reaches approximately 60%, Petho et al. reach 62.3% after 23 patches, while Dijkstra also approaches 62.3%, but with a slightly slower rate of increase.

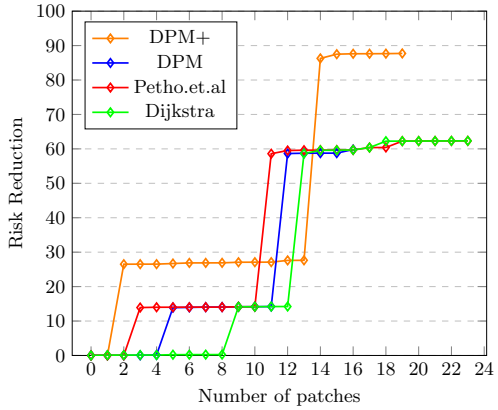

Fig. 7: Relative Risk Reduction for entry point CS2

Here, we proceed with new results from the extended content. As we introduced in Section 3, we analyzed calculated the patch priorities. Figure 8 shows

the distribution of normalized patch priorities in all analyzed CVEs. The y-axis spans from −0.5 to 0.5, capturing high-priority (positive) and low-priority (negative) patches. The yellow box indicates the interquartile range (IQR), while the red horizontal line marks the median priority. Most patches are distributed in the negative range, indicating a higher patching cost relative to benefit. A few outliers above zero suggest vulnerabilities where patching yields high benefit per unit cost. This reinforces the need for prioritization: defenders should focus first on patches above the median red line (positive priority).

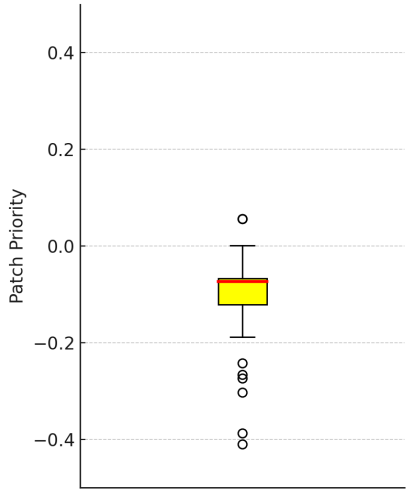

Fig. 8: Distribution of CVE Patch Priorities

We compute the CT of hosts in the topology, and as shown in Figure 9, the centrality of host `CS1, SW1, CSC` is highest among all hosts. It is easily detected by looking at the representative values. By incorporating CT into our patch prioritization logic, we explicitly account for the hosts that have the highest priority when it comes to patch installation.

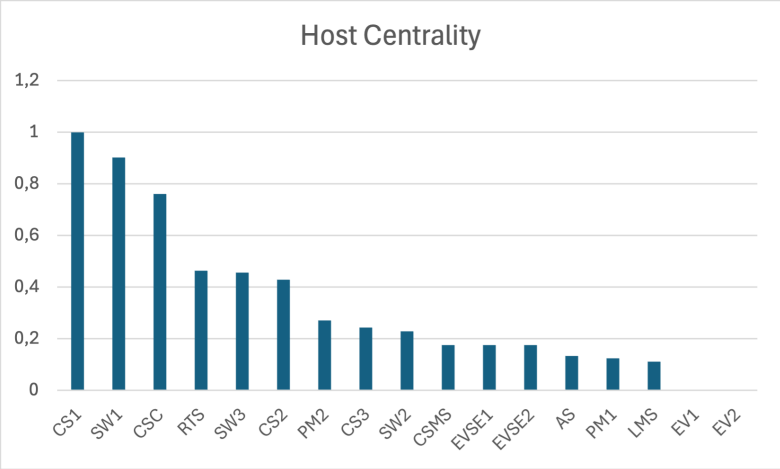

Fig. 9: Host Centrality in the Network Topology

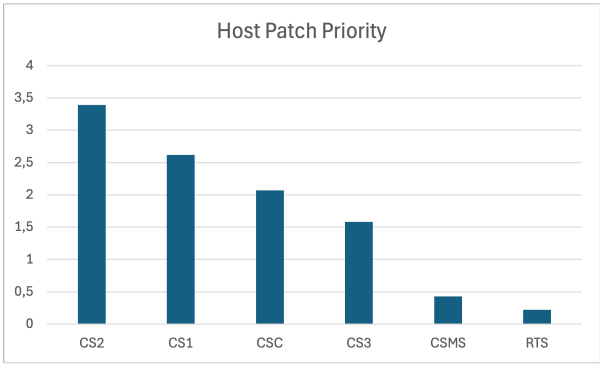

Fig. 10: Host Patch Priority in the Network Topology

Figure 10 provides a different approach to patch management, showing that the combination of host centrality and patch priority, as in the equation (7), can reveal the actual central hosts that should be prioritized when the available patches are released or exist. In our case, relying only on patch priority may not show the most urgent host to patch, since in the Figure 9 the host with the highest centrality was the host named CS1; however, after computing (7), the host named CS2 takes a higher priority.

5 Related Work

In this section, we examine previous works on vulnerability prioritization.

5.1 Topology-aware prioritization

Petho et al. [13] examined vulnerability levels within in-vehicle communication networks by employing undirected graph models. In their representation, edges capture the interconnections between components, and protection levels are expressed through resistance-like parameters. Security is quantified by summing the shortest paths between nodes, which are calculated using a generalized form of Dijkstra's algorithm. In contrast to our DPM method, their approach begins with a zero weight at the source node. It iteratively accumulates edge weights—interpreted as protection costs—to determine the lowest-cost paths. This provides a systematic way of assessing security based on component safeguards and connectivity. Nevertheless, their evaluation is limited to relatively small single-vehicle topologies, whereas EVCI environments are considerably larger and involve more actors and dependencies.

Alperin et al. [2] proposed a data-driven vulnerability assessment framework that extracts features from CVE and exploit descriptions. Their system tailors prioritization by estimating the likelihood that particular threats will target a network, relying on historical threat intelligence for prediction. While this method of path prioritization could be integrated with our topology-aware strategy to guide patch selection, it remains unclear how and when their threat risk assessment should be updated in response to newly emerging exploits.

Another relevant contribution is IoT-PEN [28], a penetration testing framework for IoT infrastructures supporting point, star, and mesh topologies in a client–server design. IoT-PEN models potential attacker movements through graphs to uncover multi-host and multi-stage attacks and recommends patch priorities primarily based on CVE severity. Our approach differs in that it emphasizes reducing overall system risk by evaluating attacker paths and prioritizing vulnerabilities according to their exploitability. While both techniques share the objective of improving resilience, our method specifically targets weaknesses within the most critical attack paths, thereby addressing vulnerabilities most likely to be exploited.

5.2 Non-graph based prioritization

Farris et al. [5] introduced VULCON, a vulnerability management framework that leverages performance indicators such as time-to-remediation and total vulnerability exposure. Their optimization strategy focuses on reducing long-term network risk by addressing the most dangerous vulnerabilities, even if this means fixing fewer issues overall.

In a similar vein, Jung et al. [8] proposed a context-aware vulnerability prioritization model that generates time-sensitive vulnerability scores for CVEs and presents them through visualizations. However, their approach often requires significant expert involvement to validate the identified vulnerabilities. Wu et al. [25] developed an operating-system-aware prioritization method that applies differential severity analysis, combining static program analysis with natural language processing to evaluate vulnerabilities in Linux and Android environments.

PatchRank, proposed by Yadav et al. [27], considers interdependencies within SCADA systems and ranks node vulnerabilities using CVSS-based severity scores together with their potential impact.

In contrast to these approaches, our work incorporates topology-awareness and attack-path selection algorithms to emulate adversarial movement through the system. We also integrate EPSS to dynamically rank vulnerabilities based on their exploitability, offering a more adaptive and real-time perspective. This makes our method particularly well-suited for distributed infrastructures such as EV charging networks, where a single compromised subsystem can trigger cascading effects on interconnected components.

More recently, Ma et al. [10] proposed VulNet, a tool that prioritizes vulnerabilities in software libraries by analyzing dependency depth, outperforming platforms like Maven Repository (MVN) and Open Source Insights (OSI). Their method ranks vulnerabilities primarily through CVSS scores and dependency structures, but it is limited to software library ecosystems and does not consider broader systems or attack-path modeling.

5.3 Graph-based prioritization

Olswang et al. [12] contributed by proposing a framework for vulnerability patch prioritization combined with attack graph visualization to support security decision-making. Their method emphasizes assessing the significance of nodes within a graph, where importance is determined by the number of attacks that traverse a given vulnerability on a specific device. In contrast, our DPM approach evaluates the overall number of vulnerabilities per host and considers the probability that each can be exploited.

Stergiopoulos et al. [21] introduced an automated method for analyzing complex attack graphs in multi-cloud environments. Their tool prioritizes vulnerabilities, examines how different system states influence the broader network, and identifies which states, weaknesses, and configurations present the most significant overall risk. The authors rely on Edmond's algorithm to generate a directed spanning tree rooted at the attacker's ultimate target. Our DPM approach differs in two key aspects: we model attacks from entry points using a breadth-first search (BFS) algorithm to uncover the riskiest attack paths, and we incorporate EPSS scores to reflect exploitability dynamics, whereas their analysis is based solely on CVSS metrics.

Pirani et al. [14] presented an attack graph framework that integrates network vulnerabilities and topology to minimize the probability of attack success at minimal cost. Their study explores connectivity measures across multiple vehicle platoon topologies, demonstrating how these measures affect resilience, detection, and disturbance rejection in distributed algorithms. However, their analysis remains focused on connectivity structures and does not address concrete vulnerabilities in software, hardware, or protocols. This limits its applicability in practical cybersecurity settings, where specific weaknesses exploited by adversaries must be considered.

5.4 Attack-path prioritization

Stellios et al. [20] introduce a source-driven, target-oriented methodology for evaluating attack paths against critical assets. Their approach extends CVSS metrics by employing vulnerability vectors to analyze potential attack scenarios. In contrast, our work relies on EPSS, which provides continuously updated information on the exploitability of CVEs.

Yang et al. [29] present a risk assessment framework for IoT hosts using attack graphs. Their method constructs attack graphs and quantifies risk by combining the probability of individual attacks with vulnerability impact values derived from multiple attributes. Host criticality, or "asset value," is determined through a hybrid approach that integrates expert judgment with the host's role in the overall topology. However, their study does not explicitly address vulnerability prioritization, which is a central focus of our work.

Table 7: Comparison of Various Vulnerability Assessment Approaches Based on Specific Criteria (adapted from [15])

Authors	Topology-aware	DS	Dynamic	Attack Path
[13]	Yes	Yes (In-vehicle)	No	Yes
[16]	No	Yes (ICS)	No	No
[29]	Yes	Yes (IoT)	No	Yes
[25]	No	No	No	No
[30]	No	Yes (ICS)	Yes	No
[7]	No	No	Yes	No
[5]	No	No	No	No
[8]	No	No	No	No
[21]	Yes	Yes (Company networks)	No	Yes
[28]	Yes	Yes (IoT)	No	Yes
[12]	Yes	Yes (Enterprise)	No	Yes
[27]	Yes	Yes (SCADA)	No	No
[20]	Yes	Yes (IoT)	No	Yes
[10]	No	Yes (Maven and OSI libraries)	No	No
Our work (DPM)	Yes	Yes (EV charging)	Yes	Yes

5.5 Summary

We summarize the related literature in Table 7, where four criteria are used to compare vulnerability prioritization approaches: topology-awareness, distributed system support, dynamic adaptation, and attack-path consideration. The column *Topology-aware* indicates whether the method incorporates the network topology when ranking vulnerabilities. *Distributed System (DS)* specifies if the approach models interconnected systems in which vulnerabilities may affect multiple components, or whether it assumes isolated, standalone environments. This distinction is crucial since EV charging infrastructures are inherently distributed, with information processed across several subsystems. The column *Dynamic* highlights whether the method leverages metrics that evolve with exploitability information, thereby reflecting newly discovered exploits and changes in the

threat landscape to enable real-time, context-aware assessments. Finally, the column *Attack Path Selection* captures whether vulnerabilities are prioritized by analyzing potential attacker paths. Among the surveyed works, our approach is the only one that satisfies all four criteria.

6 Conclusions

This paper introduces a centrality and cost-aware dynamic patch management method to enhance the cybersecurity of Electric Vehicle Charging Infrastructure. Unlike traditional approaches, our method leverages changes in vulnerability exploitability for efficient risk reduction. It improves risk mitigation by identifying the most vulnerable attack paths and prioritizing vulnerabilities based on exploitability and impact. Our empirical study on charging infrastructure vulnerabilities revealed 45 vulnerabilities that need prompt patching to avoid risks to energy infrastructures.

The results show that our method significantly reduces risk across various access locations, outperforming other methods, including closely related algorithms. Its design maximizes risk reduction, especially when prioritization is crucial and patches are limited.

In conclusion, we extend the previous work through four key modifications: redefining risk calculation with a host centrality metric, adding a Patch Detection step, enriching patch inputs via automated collection with the PatchD tool, and enhancing patch prioritization through cost–benefit analysis.

Our study findings indicate that integrating cost-benefit analysis for patches and their host centrality can support prioritizing patch installation and optimizing the use of available resources. Moreover, our approach is applicable and can be extended to any domain; therefore, vendors and operators who struggle to prioritize patches can see our work as an improvement of patch management by developing a measurement to quantify the costs of patching different vulnerabilities.

ACKNOWLEDGEMENT

This work was supported by Vinnova through the project CYREC: Cybersecurity for Resilient Energy Communities of the Future (2023-02987) and also by the RICS centre on Resilient Information and Control Systems funded by the Swedish Civil Contingencies Agency (MSB).

References

1. Alcaraz, C., Cumplido, J., Trivino, A.: Ocpp in the spotlight: threats and countermeasures for electric vehicle charging infrastructures 4.0. International Journal of Information Security (2023). https://doi.org/10.1007/s10207-023-00698-8

2. Alperin, K., Wollaber, A., Ross, D., Trepagnier, P., Leonard, L.: Risk prioritization by leveraging latent vulnerability features in a contested environment. In: Proceedings of the 12th ACM Workshop on Artificial Intelligence and Security (2019). https://doi.org/10.1145/3338501.3357365
3. Dissanayaka, N., coauthors: Software security patch management: A systematic literature review of challenges. Information and Software Technology (2022)
4. Electric, S.: Cybersecurity security notifications. Schneider Electric Cybersecurity (2025), https://www.se.com/ww/en/work/support/cybersecurity/security-notifications.jsp
5. Farris, K.A., Shah, A., Cybenko, G., Ganesan, R., Jajodia, S.: Vulcon: A system for vulnerability prioritization, mitigation, and management. ACM Transactions on Privacy and Security (TOPS) (2018). https://doi.org/10.1145/3196884
6. FIRST: CVSS v3.0 Specification Document. Forum of Incident Response and Security Teams (FIRST) (2019), accessed: 2024-11-18
7. Jacobs, J., Romanosky, S., Suciu, O., Edwards, B., Sarabi, A.: Enhancing vulnerability prioritization: Data-driven exploit predictions with community-driven insights. In: 2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) (2023). https://doi.org/10.1109/EuroSPW59978.2023.00027
8. Jung, B., Li, Y., Bechor, T.: Cavp: A context-aware vulnerability prioritization model. Computers & Security (2022). https://doi.org/10.1016/j.cose.2022.102639
9. Kern, D., Krauß, C., Hollick, M.: Attack analysis and detection for the combined electric vehicle charging and power grid domains. In: Proceedings of the 19th International Conference on Availability, Reliability and Security (2024). https://doi.org/10.1145/3664476.366451
10. Ma, Z., Mondal, S., Chen, T.H., Zhang, H., Hassan, A.E.: Vulnet: Towards improving vulnerability management in the maven ecosystem. Empirical Software Engineering (2024). https://doi.org/10.1007/s10664-024-10448-6
11. Mehri, V.A., Arlos, P., Casalicchio, E.: Automated patch management: An empirical evaluation study. Technical report / preprint (2023)
12. Olswang, A., Gonda, T., Puzis, R., Shani, G., Shapira, B., Tractinsky, N.: Prioritizing vulnerability patches in large networks. Expert Systems with Applications (2022). https://doi.org/10.1016/j.eswa.2021.116467
13. Petho, Z., Khan, I., Torok, Á.: Analysis of security vulnerability levels of in-vehicle network topologies applying graph representations. Journal of Electronic Testing (2021). https://doi.org/10.1007/s10836-021-05973-x
14. Pirani, M., Baldi, S., Johansson, K.H.: Impact of network topology on the resilience of vehicle platoons. IEEE Transactions on Intelligent Transportation Systems (2022). https://doi.org/10.1016/j.trc.2017.04.012
15. Plaka, R., Asplund, M., Nadjm-Tehrani, S.: Topology-aware prioritized patching for ev charging infrastructure vulnerabilities. In: VEHITS. pp. 113–124 (2025)
16. Rencelj Ling, E., Ekstedt, M.: Estimating time-to-compromise for industrial control system attack techniques through vulnerability data. SN Computer Science (2023). https://doi.org/10.1007/s42979-023-01750-z
17. Sarieddine, K., Sayed, M.A., Torabi, S., Attallah, R., Jafarigiv, D., Assi, C., Debbabi, M.: Uncovering covert attacks on ev charging infrastructure: How ocpp backend vulnerabilities could compromise your system. In: Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (2024). https://doi.org/0.1145/3634737.3644999

18. SecurityWeek: Ics patch tuesday advisories published by siemens, schneider electric, aveva, cisa. SecurityWeek (2025), https://www.securityweek.com/ics-patch-tuesday-advisories-published-by-siemens-schneider-electric-aveva-cisa/
19. Shaw, A.: Beautiful soup: Build a web scraper with python (2021), https://realpython.com/beautiful-soup-web-scraper-python/
20. Stellios, I., Kotzanikolaou, P., Grigoriadis, C.: Assessing iot enabled cyber-physical attack paths against critical systems. Computers & Security (2021). https://doi.org/10.1016/j.cose.2021.102316
21. Stergiopoulos, G., Dedousis, P., Gritzalis, D.: Automatic analysis of attack graphs for risk mitigation and prioritization on large-scale and complex networks in industry 4.0. International Journal of Information Security (2022). https://doi.org/10.1007/s10207-020-00533-4
22. Tan, X., Zhang, Y., Mi, C., Cao, J., Sun, K., Lin, Y., Yang, M.: Locating the security patches for disclosed OSS vulnerabilities with vulnerability-commit correlation ranking. In: Proc. ACM CCS (2021)
23. Vailoces, G., Keith, A., Almehmadi, A., El-Khatib, K.: Securing the electric vehicle charging infrastructure: An in-depth analysis of vulnerabilities and countermeasures. In: Proceedings of the Int'l ACM Symposium on Design and Analysis of Intelligent Vehicular Networks and Applications (2023)
24. Wang, X., Wang, S., Feng, P., Sun, K., Jajodia, S.: Patchdb: A large-scale security patch dataset. In: Proc. IEEE/IFIP DSN (2021)
25. Wu, Q., Xiao, Y., Liao, X., Lu, K.: {OS-Aware} vulnerability prioritization via differential severity analysis. In: 31st USENIX Security Symposium (USENIX Security 22) (2022)
26. Xu, C., Chen, B., Lu, C., Huang, K., Peng, X., Liu, Y.: Tracking patches for open source software vulnerabilities. In: Proc. ACM ESEC/FSE (2022)
27. Yadav, G., Paul, K.: Patchrank: Ordering updates for scada systems. In: 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA) (2019). https://doi.org/10.1109/ETFA.2019.8869110
28. Yadav, G., Paul, K., Allakany, A., Okamura, K.: Iot-pen: An e2e penetration testing framework for iot. Journal of Information Processing (2020). https://doi.org/10.2197/ipsjjip.28.633
29. Yang, H., Yuan, H., Zhang, L.: Risk assessment method of iot host based on attack graph. Mobile Networks and Applications (2023). https://doi.org/10.1007/s11036-023-02198-4
30. Yoon, S.S., Kim, D.Y., Kim, G.G., Euom, I.C.: Vulnerability assessment framework based on in-the-wild exploitability for prioritizing patch application in control system. In: International Conference on Information Security Applications (2023). https://doi.org/10.1007/978-981-99-8024-6_10
31. Zeng, Z., Huang, D., Liu, X., Deng, Y., Vadnere, N., Xie, Y.: Illation: Improving vulnerability risk prioritization by learning from network. IEEE Trans. Dependable and Secure Computing (2024)