Automated Patch Management: An Empirical Evaluation Study

Vida Ahmadi Mehri
Department of Computer Science
Blekinge Institute of Technology
Karlskrona, Sweden
vida.ahmadi.mehri@bth.se

Patrik Arlos
Department of Computer Science
Blekinge Institute of Technology
Karlskrona, Sweden
patrik.arlos@bth.se

Emiliano Casalicchio
Department of Computer Science
Sapienza University of Rome, Italy
casalicchio@di.uniroma1.it

***Abstract*—Vulnerability patch management is one of IT organizations' most complex issues due to the increasing number of publicly known vulnerabilities and explicit patch deadlines for compliance. Patch management requires human involvement in testing, deploying, and verifying the patch and its potential side effects. Hence, there is a need to automate the patch management procedure to keep the patch deadline with a limited number of available experts. This study proposed and implemented an automated patch management procedure to address mentioned challenges. The method also includes logic to automatically handle errors that might occur in patch deployment and verification. Moreover, the authors added an automated review step before patch management to adjust the patch prioritization list if multiple cumulative patches or dependencies are detected. The result indicated that our method reduced the need for human intervention, increased the ratio of successfully patched vulnerabilities, and decreased the execution time of vulnerability risk management.**

***Index Terms*—Vulnerability, Risk Management, Cybersecurity, Patch Management**

I. Introduction

Vulnerability Risk Management (VRM) is one of the critical aspects of information security that has been ranked in the top 10 by CIS [1]. Unpatched vulnerabilities expose organizations and individuals to cyber attacks. One well-known example is the Log4j (i.e., CVE-2021-44228, CVE-2021-45046, CVE-2021-45105, and CVE-2021-44832) which was one of the most severe threats in recent years due to the number of vulnerable systems and the ease of exploit (i.e., ten million attempts per hour [2]). VRM is a cyclic process that aims to identify, classify, evaluate, and remediate vulnerabilities and reduce an organization's attack surface. Currently, VRM is challenging due to the dramatic increase of known vulnerabilities (i.e., 40% in 2019-2022) and the explicit patch deadline enforced by regulation for public sectors. For instance, the patch deadline for federal agencies in the U.S. is 15 days for critical vulnerabilities and 30 days for vulnerabilities with high severity. For UK officials, 14 days for critical vulnerabilities[1][2]. According to the NIST National Vulnerability Database (NVD) [3], 57.69% of the new vulnerabilities reported in 2022 ranked with critical and high severity (40.61% high severity and 17.08% critical severity). Hence, automated VRM is vital to support security specialists in keeping the patch deadline. Today, we lack the tools that automatically conduct the four stages, identification, classification, evaluation, and remediation in the VRM process. In [4]–[6], we proposed Automated Context-aware Vulnerability Risk Management (ACVRM) to improve the VRM procedure by 1) reducing the labor-intensive tasks of security experts in patch prioritization; 2) customizing the patch prioritization for a given organization by learning about the organization's assets and the vulnerabilities that affect these assets; 3) automating VRM procedure to reduce processing time.

[1]https://www.ncsc.gov.uk/cyberessentials/overview
[2]https://www.cisa.gov/binding-operational-directive-19-02

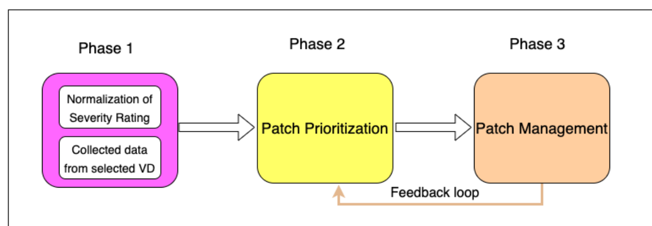

Fig. 1. Relationship between our current work (Phase 3 and feedback loop to phase 2) and previous work (Phase 1 & 2)

ACVRM consists of three phases, cf. Figure 1. In phase 1, described in [4], [6], we collected the publicly known vulnerabilities from multiple Vulnerability Databases (VD) and normalized the severity score for each vulnerability based on the selected vulnerability management mode by the organization. The collected data identifies existing vulnerabilities in the organization's assets. In phase 2 described in [5], we automatically scan the organization's asset inventory against the collected data to detect the vulnerabilities that affect the organization. We defined the criteria to be considered in patch prioritization via literature study and experts interview. The criteria weighted by security experts are based on the organization's policy and risk appetite to calculate the patch score. We use the patch score to determine the patched vulnerability's order.

This study focuses on the third phase, patch management, and the feedback loop to improve patch prioritization. Patch management is a process to mitigate the vulnerability in the organizations' assets by deploying and verifying the patch.

Patch Management (PM) is one of the most complex processes in information technology, as it requires a strong understanding of the system components and the potential patch. It is also challenging due to the uncertainty of the system's reaction to the patch and the problem with the patch released by vendors (i.e., the patched version of one vulnerability introduces another vulnerability) [7]. For example, Microsoft's security patch addressed Meltdown and Spectre vulnerabilities, which are hardware vulnerabilities that affect nearly every computer processor, causing some computers to become unbootable [8].

In this paper, we design and implement phase 3 of ACVRM, PM, and show the capability of the patch feedback loop to improve prioritization for the given organization. Our solution was deployed on a test environment where we applied our method to patch software vulnerabilities.

The paper is organized as in what follows. Section II provides background on patch management and analyzes the related literature. Section III describes our contribution, and Section IV introduces the ACVRM's patch management phase, where the paper's core contribution is. Section V describes the design and implementation of a proof of concept for patch management and feedback loop. Experiments and results are reported in Sections VI and VII, respectively. Section VIII concludes the paper.

II. Background and related work

Patch Management (PM) helps organizations keep their assets secure, reliable, and up-to-date with the required features and functionality. It is also essential for ensuring compliance with security and privacy regulations such as EU Cybersecurity act [9], EU Cybersecurity Certificate (EUCS) [10], USA homeland security act [11]. Security patches (hereafter patch) are released by hardware or software vendors to address the identified vulnerabilities in their products. PM procedure is responsible to remediate the vulnerabilities in the organization's environment. The PM generally consists of three steps [12] to address vulnerabilities in the organization:

- Patch testing tests the patch on an isolated system similar to the production to verify the impact on system/software performance or instability.
- Patch deployment is applying a patch in production environments.
- Post-deployment or patch verification is an activity to detect malfunctions or instability on the system/software post-patching vulnerability.

One of the complex issues in PM is verifying the vulnerability patches in an organization environment as each organization has its unique combination of system and configuration [12]. The patch verification answers two questions 1) Does the patch remediate the vulnerability? 2) Does the patch have side effects (i.e., does not break any other software, application, or system)? The first could be verified with vulnerability scanning or checking the version of the software or application, or system. However, the second one required a deep understanding of the architectural design of environments (e.g., system, application, software) to evaluate the impact of the patch and prepare a rollback plan [13].

TABLE I
Statistic report on the number of reported Vulnerabilities in NVD [27]

Year	No. vulnerability	Critical	High
2019	17305	2640	7243
2020	18351 (+6%)	2720 (+3%)	7708 (+6%)
2021	20158 (+10%)	2677 (-2%)	8553 (+11%)
2022	25064 (+24%)	4282 (+60%)	10179 (+19%)

Some studies [14]–[19] have reported the need for human expertise in PM due to the increased complexity of security patching and the limitations of the current technologies to provide solutions covering the entire process. However, the authors in [14], [18], [20]–[23] highlighted the significant gap in the required skills and knowledge expertise in PM. The experts' involvement in the PM procedure increases the time to patch [14]. Therefore, keeping up with the time to patch became a challenge in IT due to the increasing number of publicly known vulnerabilities. For example, the number of vulnerabilities in 2022 is 25064 (i.e., average 68 vulnerabilities per day) where 14461 (i.e., average 40 vulnerabilities per day) of them ranked as critical and high presented in Table I. Given that each vulnerability might affect N number of assets in the organization, the time required for patching each vulnerability depends on the availability of the security resources [14], [17], [24], [25]. According to [26], the mean time to remediation of the vulnerability varies in different industries (e.g., 44 days in healthcare and 92 days in public administration). An automated PM facilitates the learning of organizational context, which is an inevitable need for each organization to remain secure and compliant. Our proposed solution, ACVRM, addresses this need. ACVRM introduced the feedback loop and learned from the historical event in PM.

The authors in [16] determined that the downtime of the business critical system is a major obstacle in vulnerability patching. It also identified the struggle of the system administrators to verify the dependencies in complex applications and systems. ACVRM facilitates the creation of a dependency tree for each software and service in an organization. The dependency reflects the patch prioritization list to support the system administrator. Midtrapanon and Wills proposed automated patch management for Linux-based servers. They orchestrated the existing open-source tools to deploy the patch automatically and simultaneously for many servers. The authors claimed the set-up time was reduced and the tool was cost-efficient [18]. Our work proposed automating the entire VRM procedure for organizations independently of the platform. We improve time to patch by automated vulnerability assessment and patch prioritization in the organization's context. We also address the patch verification.

III. Contribution

In our previous work [5], we designed and implemented phase 2 of ACVRM, patch prioritization without a feedback

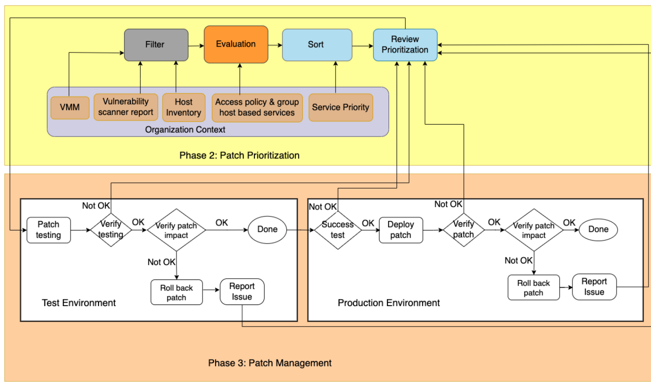

Fig. 2. ACVRM Phase 2 and 3

loop from the patch verification task in phase 3. Moreover, the review prioritization block (cf. Figure 2) used only the patch score (i.e., the score is calculated by weighting the selected criteria in the context of the organization for each vulnerability) to determine the patch order. We discovered our method in [5] needs improvement because historical data and practical experience are the inevitable factors to increase the success rate of patch [28]. We also learned from the practical experience shared in the patch management community of experts[3], some patch failures could be avoided by prioritization (e.g., patching a microcode vulnerability in Ubuntu requires the kernel to be patched in advance; otherwise, the mitigation failed). In this study, therefore, we design and implement a proof of concept of phase 3 and improve patch prioritization in phase 2. We enhance the task for review prioritization in phase 2 through a feedback loop, which facilitates review and error handling capability based on the history of events. The feedback loop introduced a learning opportunity to adjust the prioritization to reduce the patch failure rate. The feedback loop helps the organization to capture the knowledge in the organization. Hence, it reduces the experts' intervention by automating the response to the known patch failure, which is one of the goals for ACVRM.

IV. ACVRM Phase 3: Patch Management

Phase 3 and its coordination with phase 2 in handling the patches present in Figure 2. The stages in phase 3 are divided into two parts. First, patch prioritization applies in test environments similar to production. If the patch is successful, the second part will be initiated. The stages in the phase 3 describe briefly as follows:

- *Patch testing* is the first stage in PM, where the patch priority list will be deployed in a test environment. This stage aims to determine if a patch will cause problems for an organization's unique combination of hardware, software, and configuration settings. The test environment should be created similarly to the production environments. In this stage, the vulnerabilities in the patch prioritization list for the selected host will be patched sequentially in a test host for better visibility of potential issues.
- *Verify testing* is a stage to identify if the patches were successful in a test environment. The output of the patch testing will be captured and reviewed for success or failure. In case of an error, the output of patch execution will be sent to the Review Prioritization stage. If no error is reported by executing the patch in the test host, it verifies the remediation of the vulnerability. If the patch does not mitigate the vulnerability in a tested host, an alert will be sent to the Review Prioritization stage.
- *Verify patch impact* aims at detecting any side effect of the patch on the functionality of the services or applications in the organization. This stage consists of predefined functional tests to identify unexpected behavior of the organization systems or applications. If all tests were successful, the process is documented as successfully done.
- *Rollback patch* is a stage for returning the system to the prior state of patching. If the vulnerability patch causes an issue in the system or application functionality or behavior, the first response to address the issue is to roll back the patch. It brings the system or application to the latest working state before patch deployment.
- *Report issue* is an alert to experts for unexpected behavior. The report is sent to the Review Prioritization stage in phase 2, where the security and system experts should investigate the root cause and apply the lesson learned to the next patch prioritization. The report includes the CVE-ID, name of vulnerable software or service, hostname, error message, and time stamp.
- Success test is a step in the production environment to ensure all the vulnerabilities have been tested successfully in a test environment.
- Deploy patch is a stage similar to patch testing for patching vulnerabilities in a production environment. This stage depends on the output of the previous stage, test verification. If the patch is successful in a test environment, it will deploy automatically in production.
- Verify patch is a stage similar to the verify testing for the host in production environments. It verifies the status of the remediation in a production environment.

As we described above, all the errors or feedback loops return to the Review Prioritization stage in the phase 2, where the patch prioritization list is reviewed for the organization. All the stages could be automated for the organization, but any error needs experts' involvement. We consider the feedback loop a learning opportunity as the reported issues are recorded in a database for organizations to identify the challenges in their unique system and address them automatically. We expected the learning to improve the patch prioritization decision over time and reduce the number of errors, decreasing the patch failure rate and experts' intervention.

[3]http://patchmanagement.org/

V. Design and implementation

The implementation of a PoC for the PM phase (phase 3) and Review Prioritization of phase 2, shown in Figure 3, is designed as a group of functions split into five modules; *Deploy patch*,*Verify patch*, *Verify patch impact*, *Learning*, and *Automated Review*. Each module represents the implementation of each stage and the output of each module is the input for the next one. Phase 2 PoC implementation was described in [5]; hence is omitted in this paper. The output of phase 2 is a patch prioritization list (`patch_prioritization.json`) for each host in the organization, and it is the input to phase 3. The PM process is responsible for testing the patch in the test environment before patching it in production. The PM process, presented in Figure 3, starts with executing *Deploy patch*,*Verify patch*, and *Verify patch impact* modules in a test environment. If patches were successful in a test environment, the *Deploy patch* module is initiated for a selected host in production.

The *Deploy patch* process is iterated for each item (vulnerability) in the file until the stop signal is generated. ACVRM will stop patch execution if the acceptance rate of a failure [28] (i.e., the percentage of the unsuccessful patch accepted by security experts in the organization) is exceeded. The default acceptance rate in our design is up to 10% failure of patches in `patch_prioritization.json` for each host. The rate could be customized based on the organization's desires. The deploy patch module allows the second execution of the patch for each item if the first execution encounters an error. Sometimes, the error resolves in a second run based on the patch management community practices. This module generates an error report for each persistent error (an error occurred after two tries), and sends it to the *Learning* module, within the Review Prioritization.

The *Verify patch* module begins if no error is detected in the *Deploy patch* module. This module verifies the remediation by examining the running version of vulnerable software or using a vulnerability scanner. In our PoC, we use version control, where the running software version must be the same as the version installed by patch execution. If remediation fails, the system or service restart condition would examine. Some vulnerability patches require a system or service restart to be effected (e.g., Linux Kernel vulnerability requires a system restart because the installed version is loaded into memory when a system starts). In case of an error, report the error to the *Learning* module. If remediation is confirmed, ACVRM moves to the next stage.

The *Verify patch impact* module initiates when remediation is validated. This module verifies the patch's impact on the system or application functionality by running predefined functional tests. The test depends on the organization's unique services and applications and should be defined by the organization's experts (e.g., the impact of the patch in a node hosting the organization's website, could be verified by the status of the web server, and website response). ACVRM could automate the test and verify the expected result. If the side effect is detected, ACVRM will rollback the patch and report the impact to the *Learning* module. Otherwise, it jumps to the next item in the `patch_prioritization.json` until the last item. The *Learning* module is the enhancement in the Review prioritization stage. This module builds knowledge from the organization's past experiences and the current one to improve patch prioritization decisions. The `knowledge.json` is created from the historical events data and the data provided by a feedback loop in JSON format. Security experts could feed organizational historical data into the *Learning* module. If the historical data are not available, the module builds the knowledge based on the feedback loop only. The *Learning* module helps the organization handle known errors automatically without the expert's involvement. In our design, the learning module will check three conditions before escalating the error to the security expert. First, it inspects the existence of errors in `knowledge.json`. Second, it checks the dependencies in the error report. Finally, it searches for a proven solution in `knowledge.json`. If knowledge data matches the condition, it updates the patch prioritization. Otherwise, it sends a report to the experts for review and response.

Moreover, we introduce *Automated Review (AR)* in our design to improve the patch prioritization list before PM cf. yellow box in Figure 3. We find that there is a possibility of having multiple CVE-IDs for each software or application in the patch prioritization list of the host. AR reviews and sorts the CVE-IDs in the patch list based on the name of the vulnerable software or application and the patch version. If AR finds multiple patches of the same software or application, which is cumulative, it removes the older patches from the priority list. AR also checks the dependency for each vulnerability in the list because dependency is a common reason for patch failure [15], [16], [29], [30]. The dependencies are considered prerequisites, which directly impact the outcome of PM. If the organization provides a dependency tree for each software or application, AR inspects and reflects them in the prioritization list.

VI. Experiment

We plan to test our PM PoC by executing a controlled experiment. In the experiment, we set up a test organization by creating some nodes, injecting random vulnerabilities into nodes, verifying the vulnerabilities are detected, and a patch prioritization list is created for each node. Then, we apply the PM process to patch the detected vulnerabilities and verify the remediation and patch impact in a test organization based on the process presented in Figure 3. In our experiment, we assumed that the patch testing and verification were successful for a test organization. The test organization is created by a network of virtual servers deployed on a public cloud platform. Figure 4 shows our test organization setup; it consists of eight virtual servers (Host 1-8), one storage node (Local Storage), one Rudder node [31], and one Nessus node [32]. All nodes are connected to a switch. The servers are running Ubuntu as an Operating System (OS). Rudder node is a host running

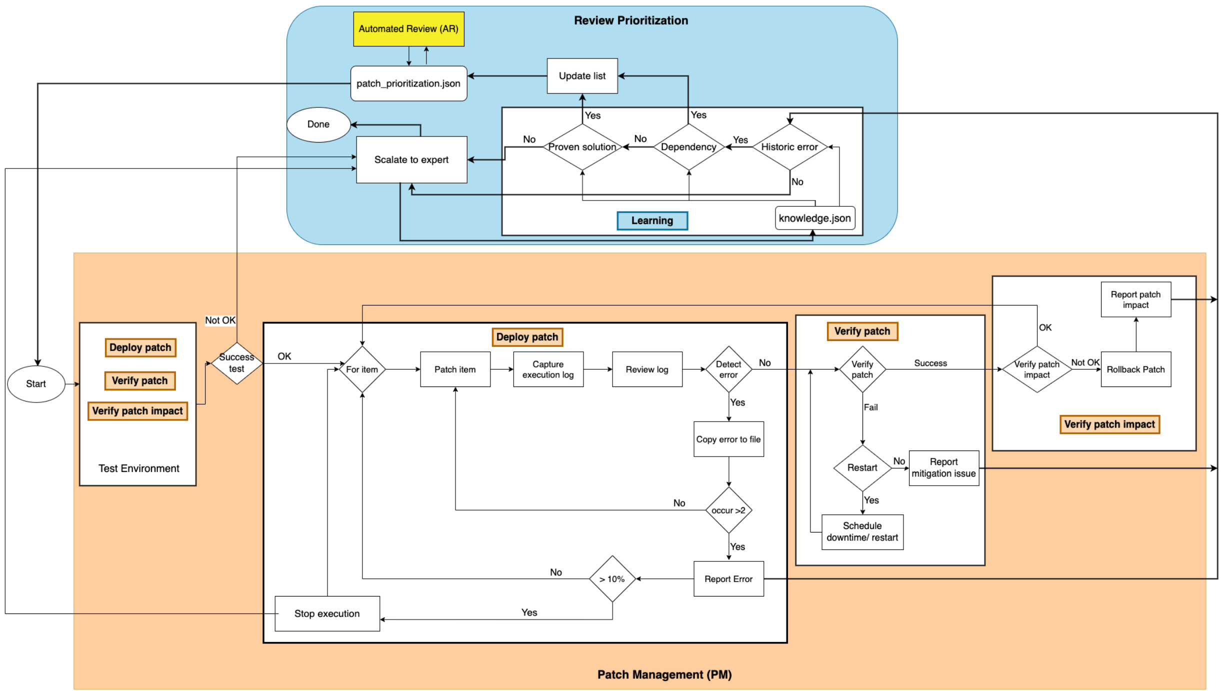

Fig. 3. ACVRM Phase 3 process including feedback loop to Phase 2 for test and production environments

the *Rudder.io* manager version 6.2 as an inventory tool. The Rudder manager receives the nodes' data through the installed Rudder agent on the eight virtual servers. Nessus node is a host running the Nessus vulnerability scanner community edition, version 8.14.0-ubuntu110_amd64. The nodes are created using the OS image provided by the cloud provider and then updated to the latest stable version available at the testing time (December 2022). Each node has 1 CPU core, 1GB RAM, and OS version 18.04.4 LTS.

After initiating the test organization, we randomly select 21 CVE-IDs (relevant to our virtual servers) and install their vulnerable version on our eight virtual servers (Host 1-8). We deploy phase 1 and phase 2 of ACVRM to identify the installed vulnerability and obtain a patch prioritization for each host. In our experiment, we weight the prioritization criteria homogeneously, i.e., $w_i = 0.1667$ in Patch Score (PS) calculation. Table II shows the patch priority list for Hosts 1-8, the selected CVE-IDs, the name of the software, the Severity Score (SC), PS, and the patched version of the software. SC is a normalized score with a standard vulnerability management module [6].

We implement a PM PoC, our core contribution in this study, in our test organization according to our defined process. The aim is to investigate the impact of the patch review and feedback loop on the success of automated patching and reducing expert intervention. Hence, we define three cases:

- Case 1: PM based on the patch prioritization list in our previous work [5], where the PS determines the patch order. In this case, there is no automated review of the patch prioritization. We also consider the organization does not have any historical data from its previous patch. Moreover, we allowed the patch execution without interruption until the end of the priority list (i.e., do not check the error percentage for a test host).

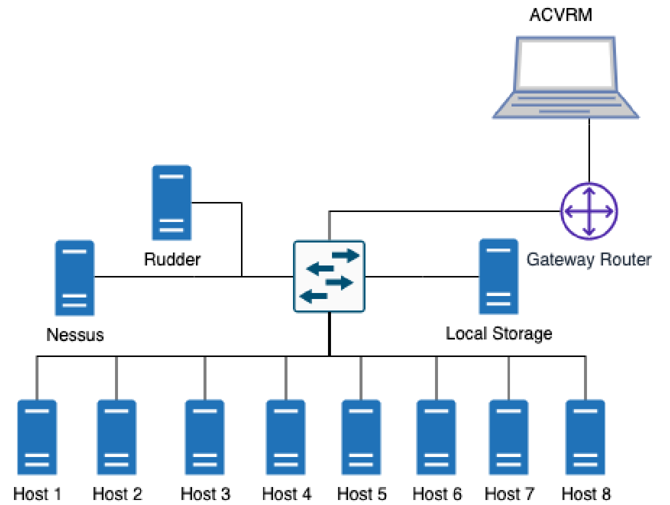

Fig. 4. Test environments

- Case 2: PM based on the updated `patch_prioritization.json` by adding an AR in the process. In this case, the organization does not provide dependencies. Therefore, the update is based on removing unnecessary patches (e.g., multiple cumulative patches of software or application) from the list.
- Case 3: PM based on an improved priority list by

automatically checking dependencies of the vulnerabilities. In this case, we consider the test organization has a record of dependencies, and AR will update the `patch_prioritization.json` accordingly.

For each case, we keep the state of the virtual servers (e.g., with 21 installed vulnerabilities) unchanged and review the PM output for each case. We also disabled the check percentage of error for stop execution in our experiment as we wanted to capture all errors without interruption.

VII. Results and Discussion

In this study, we used the success rate of the patch as a metric to evaluate the proposed PM approach. We verified the patch impact in our test organization through a GET request to the website of the test organization and the state of the system (up and running) after the patch.

The result of Case 1 in Figure 5 shows that 48% of the patches escalate to the expert because of the errors in patch execution. We also observed that 19% of the patches failed in verification because they required a system or service restart after the patch. We noticed the success rate of the patch was 33% without a feedback loop and 52% with a feedback loop (i.e., the condition of 19% of verification failure resolved after system or service restart). Moreover, we identify the majority of the errors in Case 1 belong to software with a different version in the priority list, but the error messages are not the same. The column Case 1 Result in Table III presents the patch result for Case 1.

The PM results of Case 2 in Figure 5 show the improvement in the success rate of the patch by 60% without the feedback loop and by 90% with a feedback loop. Also, the expert intervention decreased from 48% to 10% comparing Case 1. 30% of the patch failed in verification has been resolved by feedback loop condition as all required system or service restart. We noticed that the number of CVE-IDs in our sample list decreased from 21 to 10 with AR.

The Case 3 result in Figure 5 shows the 80% success rate without a feedback loop and 0% unknown error to escalate to the expert. 20% of verification failures were addressed by feedback loop as they required system restart.

Table III presents the patch result without a feedback loop to visualize the impact of the feedback loop for three cases (e.g., Case 1 Result, Case 2 Result, and Case 3 Result). The patch priority has been changed for Case 2 and Case 3 compared to Case 1. We define Δ as the difference between the position in patch priority ($P_{k,*}$) between Case 1 and other cases as:

$$\Delta_{k,j} = P_{\text{k,Case 1}} - P_{\text{k,Case j}} \quad j = 2, 3 \tag{1}$$

where k is CVE-ID in the list. A positive value of Δ indicates that the CVE-ID got a higher priority concerning Case 1. The negative value of Δ refers to lower priority compared with Case 1, while the zero value of Δ marks no changes in the priority. The Case 2/3 priority (Δ) column in Table III shows the priority of Case 2/3, and the Δ value in parenthesis indicates changes in the CVE-ID position compared with Case 1. For example, the CVE-2022-2526 has a priority 3 by Case 1 while it becomes a priority 1 in Case 2 and Case 3. The lack of values for the Case 2 and 3 priority (Δ) column means that the CVE-ID has been removed from the priority list. We only see the $Delta$ with a positive value as the number of the CVE-IDs decreases 52% for Case 2 and Case 3. For example, Case 1 consists of four patches for apache (CVE-2021-44790, CVE-2021-39275, CEV-2022-23943, and CVE-2022-31813), which is reduced to one in Case 2 and Case 3.

We observe that Case 2 and Case 3 have the same CVE-IDs in the patch list, but the prioritization is different. For example, CVE-2022-31813 has a priority 2 in Case 2 and a priority 4 in Case 3. The differences in Case 2 and Case 3 prioritization are due to the impact of the dependencies. In Case 3, the organization provides a dependency tree of each vulnerability, and AR verifies dependencies and adjusts the priority order accordingly. In our test organization, three CVE-IDs (CVE-2022-31813, CVE-2022-2068, and CVE-2022-32221) have the `libc6` in their dependencies tree. The position of `libc6` vulnerability (CVE-2022-23219) is after those three CVE-IDs based on PS. As described in V, dependencies are prerequisites. The dependence got a higher priority despite having a lower PS. Hence, the position of CVE-2022-23219 was adjusted to be patched before the software, depending on it. The result from all three case studies shows the impact of automation on reducing the expert intervention in patch vulnerabilities. We also verify the improvement in the patch prioritization by adding a feedback loop and learning from patch history.

Furthermore, we observe that AR improved efficiency in patch prioritization and PM in Case 1 and 2 because AR removes the unnecessary patch from the list and adjusts the list based on dependencies. Moreover, the verification of patch impact for all three cases was positive, and the patches did not break the system functionality.

VIII. Conclusion and Future Work

The increasing number of publicly known vulnerabilities introduces the challenge in VRM as it requires experts' intervention. The average number of vulnerabilities with critical and high scores was 40 per day in 2022, according to NVD [27], which forced the organizations to improve their VRM procedure to remain secure and compliant. In this study, we introduced the PM of ACVRM to automate patching, reduce experts' intervention, and improve the success rate of the patch. We performed an analysis as follows:

1) We learned the challenges in PM and the criteria that should be considered from the literature review and patch management community best practices. We determined that time to patch, expert availability, system downtime, and dependencies are important criteria in patching vulnerabilities. Therefore, we define our automated patching process to reduce expert intervention while increasing the success rate of the patch.
2) We designed and implemented phase 3 of ACVRM, which consists of testing and verifying the vulnerabil-

TABLE II
THE SAMPLE OF PATCH PRIORITIZATION LIST FOR HOST 1-4 WITH UBUNTU OPERATING SYSTEM

Priority	CVE-ID	Name	SC	PS	Patched version
1	CVE-2021-3711	openssl	9.1833	4.0810	1.1.1-1ubuntu2.1∼18.04.13
2	CVE-2021-44790	apache	8.3500	3.9421	2.4.29-1ubuntu4.21
3	CVE-2022-2526	systemd	8.3500	3.9421	237-3ubuntu10.56
4	CVE-2021-39275	apache	7.7833	3.8476	2.4.29-1ubuntu4.17
5	CVE-2022-23943	apache	7.7833	3.8476	2.4.29-1ubuntu4.22
6	CVE-2022-31813	apache	7.5167	3.8031	2.4.29-1ubuntu4.24
7	CVE-2022-2068	openssl	7.3167	3.7698	1.1.1-1ubuntu2.1 18.04.19
8	CVE-2022-32221	curl	6.6833	3.6642	7.58.0-2ubuntu3.21
9	CVE-2022-22576	curl	7.2167	3.6598	7.58.0-2ubuntu3.17
10	CVE-2022-23219	glibc	6.2667	3.5948	2.27-3ubuntu1.5
11	CVE-2022-45061	python	6.8167	3.4997	3.6.9-1∼18.04ubuntu1.9
12	CVE-2020-24489	intel-microcode	8.5167	1.9198	3.20210608.0ubuntu0.18.04.1
13	CVE-2022-42896	Linux Kernel	7.8500	1.8204	4.15.0-202.213
14	CVE-2021-4034	policykit-1	7.8500	1.8087	0.105-20ubuntu0.18.04.6
15	CVE-2022-34918	Linux Kernel	7.8500	1.8087	4.15.0-191.202
16	CVE-2022-0392	vim	7.0167	1.6698	2:8.0.1453-1ubuntu1.10
17	CVE-2022-1621	vim	6.8500	1.6420	2:8.0.1453-1ubuntu1.9
18	CVE-2021-0146	intel-microcode	6.4500	1.5170	3.20220510.0ubuntu0.18.04.1
19	CVE-2021-33910	systemd	6.3167	1.3664	237-3ubuntu10.49
20	CVE-2020-24513	intel-microcode	5.8500	1.2886	3.20210608.0ubuntu0.18.04.1
21	CVE-2022-21233	intel-microcode	5.6500	1.2553	3.20220809.0ubuntu0.18.04.1

TABLE III
THE PATCH RESULT OF CASE 1-3; THE EMPTY CELLS INDICATE THE CVE-ID REMOVED FROM THE PATCH PRIORITY

Priority	CVE-IDs	Name	Case 1 Result	Case 2 Priority (∆)	Case 2 Result	Case 3 Priority (∆)	Case 3 Result
1	CVE-2021-3711	openssl	Verification fail				
2	CVE-2021-44790	apache	Error escalate to expert				
3	CVE-2022-2526	systemd	Successful	1 (2)	Successful	1 (2)	Successful
4	CVE-2021-39275	apache	Error escalate to expert				
5	CVE-2022-23943	apache	Error escalate to expert				
6	CVE-2022-31813	apache	Error escalate to expert	2 (4)	Error escalate to expert	4 (2)	Successful
7	CVE-2022-2068	openssl	Error escalate to expert	3 (4)	Verification fail	3 (4)	Successful
8	CVE-2022-32221	curl	Successful	4 (4)	Successful	5 (3)	Successful
9	CVE-2022-22576	curl	Error escalate to expert				
10	CVE-2022-23219	libc6	Successful	5 (5)	Successful	2 (8)	Successful
11	CVE-2022-45061	python	Successful	6 (5)	Successful	6 (5)	Successful
12	CVE-2020-24489	intel-microcode	Error escalate to expert				
13	CVE-2022-42896	Linux	Verification fail	7 (6)	Verification fail	7 (6)	Verification fail
14	CVE-2021-4034	policykit-1	Successful	8 (6)	Successful	8 (6)	Successful
15	CVE-2022-34918	Linux	Verification fail				
16	CVE-2022-0392	vim	Successful				
17	CVE-2022-1621	vim	Error secalate to expert	9 (8)	Successful	9 (8)	Successful
18	CVE-2021-0146	intel-microcode	Verification fail				
19	CVE-2021-33910	systemd	Successful				
20	CVE-2020-24513	intel-microcode	Error escalate to expert				
21	CVE-2022-21233	intel-microcode	Error escalate to expert	10 (11)	Verification fail	10 (11)	Verification fail

ity patches and the impact of patches on the system functionality. We learned that reviewing prioritization based on the history of the patch will improve the patch prioritization in the organization's context.

3) We verified the result of our proposed PM by analyzing the outcome of each case. Our result shows that the ACVRM could adjust the patch prioritization for each organization with less effort from security experts. Automated Review has been introduced to remove unnecessary patches and reduce errors due to dependencies. Our solution allows security experts to set the patch failure rate and stop the execution of the error-prone patch prioritization.

Our study shows how the organization could automate VRM based on its context. ACVRM facilitates improvement in the VRM procedure by patch prioritization in the organization's context and reducing experts' intervention. The feedback loop provides an opportunity of learning from historical data and enhances the organization's knowledge which increases the success rate of patching. In the future, we could add the learning from the patch management community, social media, and threat intelligence into the review prioritization to support better decisions on the vulnerability ranking. Hence, the success rate of patching will improve. Another possible future direction could be using a machine learning algorithm such as a decision tree in the *Learning module* to improve error handling.

REFERENCES

[1] CIS Controls . http://www.cisecurity.org/controls/. [Online; accessed 10-Jan-2023].

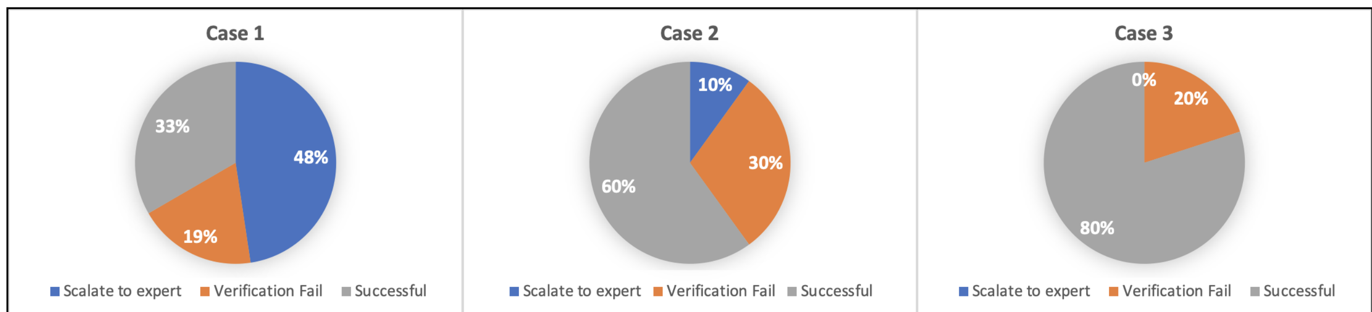

Fig. 5. The status of the patch management for cases 1-3

[2] The Log4j Vulnerability. https://www.wsj.com/articles/what-is-the-log4j-vulnerability-11639446180. [Online; accessed 16-November-2022].

[3] NIST National Vulnerability Database. https://nvd.nist.gov/. [Online; accessed 10-Jan-2023].

[4] Vida Ahmadi, Patrik Arlos, and Emiliano Casalicchio. Normalization of severity rating for automated context-aware vulnerability risk management. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*, pages 200–205. IEEE, 2020.

[5] Vida Ahmadi Mehri, Patrik Arlos, and Emiliano Casalicchio. Automated context-aware vulnerability risk management for patch prioritization. *Electronics*, 11(21):3580, 2022.

[6] Vida Mehri, Patrik Arlos, Emiliano Casalicchio, et al. Normalization framework for vulnerability risk management in cloud. In *IEEE International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, 2021.

[7] Steve Beattie, Seth Arnold, Crispin Cowan, Perry Wagle, Chris Wright, and Adam Shostack. Timing the application of security patches for optimal uptime. In *LISA*, volume 2, pages 233–242, 2002.

[8] Microsoft halts AMD meltdown and spectre patches after reports of unbootable pcs. https://www.theverge.com/2018/1/9/16867068/microsoft-meltdown-spectre-security-updates-amd-pcs-issues. [Online; accessed 10-Jan-2023].

[9] EU Cybersecurity Act. https://eur-lex.europa.eu/eli/reg/2019/881/oj. [Online; accessed 11-January-2023].

[10] European Cybersecurity Certification Scheme for Cloud Services. https://www.enisa.europa.eu/publications/eucs-cloud-service-scheme. [Online; accessed 11-January-2023].

[11] Homland Security Act 2002. https://www.dhs.gov/homeland-security-act-2002. [Online; accessed 11-January-2023].

[12] Nesara Dissanayake, Asangi Jayatilaka, Mansooreh Zahedi, and M Ali Babar. Software security patch management-a systematic literature review of challenges, approaches, tools and practices. *Information and Software Technology*, 144:106771, 2022.

[13] Ugo Gentile and Luigi Serio. Survey on international standards and best practices for patch management of complex industrial control systems: the critical infrastructure of particle accelerators case study. *International Journal of Critical Computer-Based Systems*, 9(1-2):115–132, 2019.

[14] Gerald Post and Albert Kagan. Computer security and operating system updates. *Information and Software Technology*, 45(8):461–467, 2003.

[15] John Dunagan, Roussi Roussev, Brad Daniels, Aaron Johnson, Chad Verbowski, and Yi-Min Wang. Towards a self-managing software patching process using black-box persistent-state manifests. In *International Conference on Autonomic Computing, 2004. Proceedings.*, pages 106–113. IEEE, 2004.

[16] Christian Tiefenau, Maximilian Häring, Katharina Krombholz, and Emanuel Von Zezschwitz. Security, availability, and multiple information sources: Exploring update behavior of system administrators. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 239–258, 2020.

[17] Hai Huang, Salman Baset, Chunqiang Tang, Ashu Gupta, KN Madhu Sudhan, Fazal Feroze, Rajesh Garg, and Sumithra Ravichandran. Patch management automation for enterprise cloud. In *2012 IEEE Network Operations and Management Symposium*, pages 691–705. IEEE, 2012.

[18] Soranut Midtrapanon and Gary Wills. Linux patch management: with security assessment features. In *4th International Conference on Internet of Things, Big Data and Security (IoTBDS)*, 2019.

[19] Fengli Zhang, Philip Huff, Kylie McClanahan, and Qinghua Li. A machine learning-based approach for automated vulnerability remediation analysis. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9. IEEE, 2020.

[20] Chuan-Wen Chang, Dwen-Ren Tsai, and Jui-Mi Tsai. A cross-site patch management model and architecture design for large scale heterogeneous environment. In *Proceedings 39th Annual 2005 International Carnahan Conference on Security Technology*, pages 41–46. IEEE, 2005.

[21] Michal Procházka, Daniel Kouril, Romain Wartel, Christos Kanellopoulos, and Christos Triantafyllidis. A race for security: Identifying vulnerabilities on 50 000 hosts faster than attackers. In *Proceedings of Science (PoS). International Symposium on Grid and Clouds*, 2011.

[22] Jong-Hyouk Lee, Seon-Gyoung Sohn, Beom-Hwan Chang, and Tai-Myoung Chung. Pkg-vul: Security vulnerability evaluation and patch framework for package-based systems. *ETRI journal*, 31(5):554–564, 2009.

[23] Ben Marx and Deon Oosthuizen. Risk assessment and mitigation at the information technology companies. *RISK GOVERNANCE & CONTROL: Financial markets and institutions*, page 44.

[24] Frank Li, Lisa Rogers, Arunesh Mathur, Nathan Malkin, and Marshini Chetty. Keepers of the machines: Examining how system administrators manage software updates for multiple machines. In *Fifteenth Symposium on Usable Privacy and Security (SOUPS 2019)*, pages 273–288, 2019.

[25] Ankit Shah, Katheryn A Farris, Rajesh Ganesan, and Sushil Jajodia. Vulnerability selection for remediation: An empirical analysis. *The Journal of Defense Modeling and Simulation*, 2022.

[26] 2022 Vulnerability statistics report. https://www.edgescan.com/2022-vulnerability-statistics-report-lp/. [Online; accessed 11-Jan-2023].

[27] NIST National Vulnerability Database search. https://nvd.nist.gov/vuln/. [Online; accessed 10-Jan-2023].

[28] Yogita Kansal, Deepak Kumar, and PK Kapur. Vulnerability patch modeling. *International Journal of Reliability, Quality and Safety Engineering*, 23(06):1640013, 2016.

[29] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE symposium on security and privacy*, pages 692–708. IEEE, 2015.

[30] Roland Schwarzkopf, Matthias Schmidt, Christian Strack, and Bernd Freisleben. Checking running and dormant virtual machines for the necessity of security updates in cloud environments. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 239–246. IEEE, 2011.

[31] Rudder. https://www.rudder.io/. [Online; accessed 14-January-2023].

[32] Nessus Vulnerability Scanner. https://www.tenable.com/products/nessus. [Online; accessed 12-January-2023].