Towards Automated Patch Management in a Hybrid Cloud

Ubaid Ullah Hafeez[1], Alexei Karve[2], Braulio Dumba[2], Anshul Gandhi[1], and Sai Zeng[2]

[1] PACE Lab @ Stony Brook University, Stony Brook NY 11794, USA
[2] IBM Thomas J. Watson Research Center, Yorktown Heights NY 10598, USA

Abstract. Software patching is routinely employed for enterprise online applications to guard against ever-increasing security risks and to keep up with customer requirements. However, in a hybrid cloud setting, where an application deployment can span across diverse cloud environments, patching becomes challenging, especially since application components may be deployed as containers or VMs or bare-metal machines. Further, application tiers may have dependencies, which need to be respected. Worse, to minimize application downtime, selected patches need to be applied in a finite time period. This paper presents an automated patching strategy for hybrid-cloud-deployed applications that leverages a greedy algorithm design to optimally patch applications. Our implementation and evaluation results highlight the efficacy of our strategy and its superiority over alternative patching strategies.

1 Introduction

Online enterprise applications today are often deployed in a hybrid cloud — a computing environment that combines the benefits of public and private clouds by sharing data and application deployment between them. A hybrid cloud is cost-effective and elastic as the public cloud portion of the application follows a pay-as-you-go model. On the other hand, the private cloud portion can be kept behind a firewall, on compliant machines, to ensure data security and privacy.

To avoid security breaches and keep the application updated according to customer requirements, most online applications employ software patching. Applying a security patch as soon as it is available can prevent 57% of the security breaches [2]. In addition to security patches, application update patches are also important. For example, an online application launched a time ticker sidebar which was regarded as "spambar" by users and decreased the popularity of the application among desktop users; this sidebar was taken down shortly to avoid any further customer disappointment [6]. Thus, timely patching is one of the key requirements for secure and performant functioning of online applications [2].

Patch management of applications deployed in a hybrid cloud environment is complicated and tedious. Applications in hybrid cloud often span multiple components, also referred to as services or tiers. There may be numerous pending patches, with different priorities, including those that are critical. Typically, there is only a limited time period, referred to as *maintenance window*, during which the application can be brought down and patched in an offline manner. Also, while patching, application components should be turned off in a specific order to avoid violating dependencies and to prevent the application from crashing. This makes manual patching for different tiers across different types of clouds and deployment types expensive (and possibly infeasible) and prone to human error. Clearly, an automated patch management system would be invaluable for applications deployed in a hybrid cloud.

There are some existing application management tools, e.g., Puppet [12], RCP [11] which are used for automating application patching. However, these tools do not allow application owners to limit downtime for patching by specifying a maintenance window. Enterprise applications cannot be taken offline for arbitrarily long time periods, making existing tools ineffective for automated patching of enterprise applications. There is another prior work on patching which focus on applications deployed in just a single cloud [10]. While there are some works that discuss patch management in a hybrid cloud [8], they assume that all components of the application have replicas and can be patched online.

This paper presents Hybrid Cloud Patch Manager (HCPM), a framework for automated patch management for applications deployed in a hybrid cloud. HCPM applies the optimal subset of patches in any given maintenance window and patches tiers that can be patched online whenever possible. For patch selection in a given window, HCPM employs PatchSelect, a greedy yet optimal algorithm. While patching, HCPM takes application dependencies into account, by constructing a dependency graph, thus ensuring that the application is always healthy and does not crash. Experiments confirm that HCPM effectively patches hybrid-cloud–deployed multi-tier applications within the given offline window. We further evaluate HCPM using simulations, and compare against other patching strategies, for a complex, 11-tier application. Our results, in various patching scenarios, show that HCPM outperforms other strategies by 2-29%, on average, and by as much as 2×. To make HCPM easily deployable in practice, we implement it as a plug-in that can be integrated with existing applications.

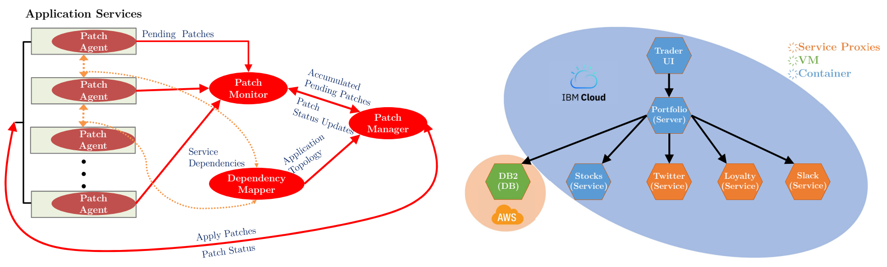
To the best of our knowledge, this is the first work on automated patch management for enterprise applications, deployed across containers and VMs in a hybrid cloud, that considers the length of the maintenance window and dependencies across application tiers.

2 Automated Patch Management

HCPM automates patch management for applications deployed in a hybrid cloud and ensures that critical patches are applied as early as possible while complying with application dependencies. HCPM's architecture is shown in Figure 1(a). HCPM consists of a *DependencyMapper*, a *PatchMonitor*, a few *PatchAgents* and a *PatchManager*. The *PatchMonitor* and *DependencyMapper* provide inputs to *PatchManager* to enable automated patching, whenever feasible.

The *PatchAgent* (per-cloud or per-tier) monitors the state of the application and OS patches for each node, and maintains a list of pending patches along with their importance. Whenever there is a new pending patch, the *PatchAgent* communicates this list to the *PatchMonitor*. The *PatchMonitor* aggregates pending patches, along with their importance scores, across all *PatchAgents*. When required, *PatchMonitor* communicates the pending patches to *PatchManager*.

The *DependencyMapper* constructs the dependency graph of the application using tools for automatic discovery as well as direct input from the application.



(a) HCPM's architecture (b) Topology

Fig. 1. Stock-trader application deployment in our hybrid cloud environment. The direction of the edges represents the dependence relationship between each component of the application: $trader \rightarrow portfolio \rightarrow Db2$; $Stocks$; $Twitter$; $Slack$; $Loyalty$.

DependencyMapper keeps updating the topology in real time as it discovers more information about the application. The *PatchManager* communicates with *DependencyMapper* and *PatchMonitor* to perform the actual patching.

For selection of offline patches, *PatchManager* employs our PatchSelect algorithm. For a given maintenance window of size, say, W minutes, PatchSelect finds optimal subset of patches for each tier separately as multiple tiers can be patched simultaneously in the offline window. For a given tier, let the estimated time to reboot its node be R and that to reboot all its dependent tiers be R'; then, we have $W_{actual} = W - R - R'$ minutes left to apply patches to the tier. PatchSelect partitions all patches according to their importance level and then sorts the patches, for each level, in ascending order of their patching time. Starting from the most important level (level $i = 1$), PatchSelect greedily selects as many patches as possible, in sorted order, such that the sum of their patching time is less than W_{actual}. If the time to apply patches of level $i = 1$ is $W_1 < W_{actual}$, and either all patches of level $i = 1$ have been applied or no more patches of level $i = 1$ can be applied without exceeding W_{actual}, then the algorithm proceeds similarly to the next priority levels, in order, with remaining time $W_{actual} = W_{actual} - W_1$. Given an application with S tiers and N number of pending patches, the time complexity of PatchSelect is $\mathcal{O}(S^2 + N\ log(N))$.

To minimize the time for applying patching, *PatchManager* is implemented as a multi-process agent which brings down tiers in order of dependencies and starts applying patches to offline tiers simultaneously. Once the patching of a subset of patches is successful, *PatchManager* informs *PatchMonitor*, which in turn removes the applied patches from the pending list. If some of the patches fail, *PatchManager* logs the specific cause of failure and reverts the patches so that the application stays healthy.

3 Experimental Evaluation

This section evaluates the performance of HCPM for stock-trader [5], a multi-tier microservice-based application deployed in our hybrid cloud environment as shown in Figure 1(b). Our hybrid cloud environment is composed of a 2-core, 8GB memory VM on AWS public cloud, and 4 4-core, 16GB memory VMs on our private cloud. All VMs on our private cloud are connected as a cluster using the open-source ICP — a Kubernetes-based private cloud. To simulate real workload of multiple users (create new portfolios, buying stocks, etc.), we use jmeter [4].

HCPM Deployment: We implement HCPM as follows: the *PatchAgent* module is implemented using the publicly available Vulnerability Advisor (VA) [3] and BigFix [1]. We use REST APIs of VA and BigFix to communicate with *PatchManager*. The *PatchMonitor* and *PatchManager* are implemented in Python. The *DependencyMapper* module is implemented using WeaveScope [7].

Evaluation: In our deployment of HCPM, the *PatchAgent* module periodically scans for services that are missing critical security patches. HCPM extracts the dependency (see Figure 1(b)) among the components of the stock-trader using its *DependencyMapper* module. Then, it uses the PatchSelect algorithm to find and apply the optimal subset of patches from the list of pending patches (see Table 1) for the given maintenance window, which is set to 2 minutes, and restarts the containers within the maintenance window. While restarting, HCPM makes sure that dependency constraints are not violated. Given a short maintenance window of 2 minutes, HCPM identifies that the patches for the Db2 VM cannot be applied, and these are thus omitted by PatchSelect; the remaining patches are applied in 54 seconds.

Tier	**Pending vulnerable packages**
Portfolio	libgcrypt20, procps, gpgv, libssl1.0.0, gnupg, libprocps4
Trader	gnupg, libgcrypt20, libssl1.0.0, libprocps4, gpgv, procps
Db2	java (RHSA-2018:0349-01), wget (RHSA-2018:3052)

Table 1. Pending packages for the images of stock-trader.

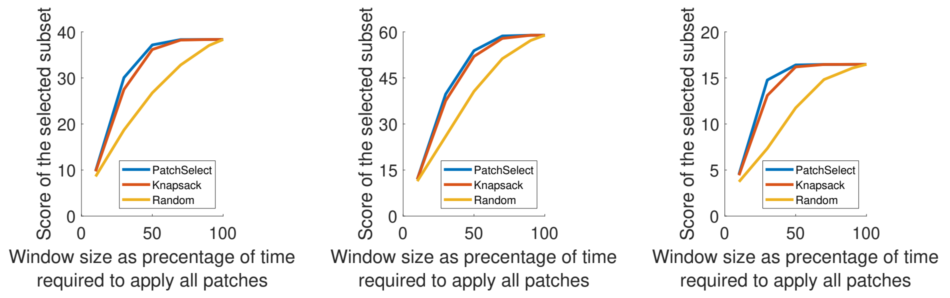
4 Simulation Results

For simulations (written in Python), we consider a large, multi-tier data streaming application, as in VScope [9], which has 11 tiers. We use details of pending patches based on the statistics of OS patches from last year [3] . To assign importance scores to patches, we consider three different scenarios using scores of 1 (high), 2 (med), and 3 (low) as shown in Figure 2.

Figure 2 compares the performance of PatchSelect with two other strategies i.e. *Knapsack*, which models patch selection as a knapsack problem and uses a dynamic programming algorithm and *Random*, which is a simple strategy of randomly selecting patches until the maintenance window is exhausted. In Figure 2, we use geometric scores when computing the accumulated score for a subset because typically, it is more important to apply a *single* high priority patch as compared to numerous low priority patches. Let N be the total number of pending patches. Consider a subset s with m_i number of patches of importance i, for $i \in \mathbb{Z}^+$. We define the accumulated geometric score for a subset s as:

$$score(s) = \sum_{i \geq 1} \sum_{j=1}^{m_i} \frac{1}{N^i} = \sum_{i \geq 1} \frac{m_i}{N^i} \quad (1)$$

In all three scenarios in Figure 2, PatchSelect improve the performance over random by about 18-29%, on average, with a maximum improvement of 53-101%. Against knapsack, we improve performance by about 2-3%, on average,



(a) Uniformly distributed patch importance levels i.e. 33% hi, 33% med., 33% low. (b) Skewed distribution of patch importance levels i.e. 50% hi, 30% med., 20% low. (c) Empirically distributed patch importance levels i.e. 12% hi, 48% med., 40% low.

Fig. 2. Simulation results showing the performance of PatchSelect, knapsack, and random patch selection under different patch importance levels.

with a maximum improvement of 6-13%. While the knapsack score is often close to that of PatchSelect, we find that the running time of knapsack is about 100× that of PatchSelect in almost all cases.

5 Conclusion

The emergence of hybrid cloud computing has made it easier for businesses to leverage the elasticity of economical public clouds while safeguarding sensitive data in their private clusters. However, this distributed deployment makes it difficult to patch hybrid-cloud–deployed applications, especially when the application has to be taken down to apply critical patches. Our solution, HCPM, automatically patches application components across clouds within the allotted offline time period while respecting tier dependencies. Importantly, HCPM does so while providing optimality guarantees and bounds on running time.

References

1. BigFix. ”https://www.ibm.com/security/endpoint-security/bigfix”
2. How to shut the window of (unpatched) opportunity. https://www.welivesecurity.com/2018/04/19/patching-shut-window-unpatched
3. IBM Vulnerability Advisor. ”https://github.com/IBM-Bluemix-Docs/va”
4. Jmeter. https://jmeter.apache.org
5. Stock-trader application. https://github.com/IBMStockTrader
6. Time's up for the Ticker? Facebook appears to axe feed for tracking your friends' activity. ”https://techcrunch.com/2017/12/10/times-up-for-facebook-ticker/”
7. Weavescope. https://github.com/weaveworks/scope
8. A. Hopmann et al.: High availability of machines during patching
9. C. Wang et al.: VScope: Middleware for Troubleshooting Time-sensitive Data Center Applications. In: Middleware 2012. pp. 121–141. Montreal, Canada
10. Dake, S.C.: Containerized upgrade in operating system level virtualization
11. K. Kloeckner et al.: Building a cognitive platform for the managed it services lifecycle. IBM Journal of Research and Development **62**(1), 8–11 (jan 2018)
12. Plummer, S., Warden, D.: Puppet: Introduction, implementation & the inevitable refactoring. In: Proceedings of the 2016 ACM SIGUCCS Annual Conference