Analyzing the Overhead of Security Patching on Containers

Fogo Tunde-Onadele
North Carolina State University
Raleigh, North Carolina
oatundeo@ncsu.edu

Darshan Patel
North Carolina State University
Raleigh, North Carolina
dpatel12@ncsu.edu

1 ABSTRACT

Performing updates to virtual systems may affect resource constraints and rely on human intervention. However, automating this process raises many questions, one of which asks how patching can be strategically executed. We have compiled related works to aid our understanding of the research area. Our focus is to study the impact patches places on container virtual systems in particular. We tackle this by monitoring system metrics (i.e memory usage and CPU utilization). In addition, our approach involves a system that implements patching while maintaining availability of the running service. We learn from our studied vulnerabilities that the patching imposes little to no lasting overhead to system metrics of the host. A maximum increase of 0.65% in average CPU utilization and 0.093% in average Memory usage was observed.

2 INTRODUCTION

In the Internet Security Threat Report of Symantec published in April 2017, a large number of vulnerabilities were listed. For example, in the year 2017, 76% of websites had vulnerabilities and 9% of which were critical [1]. This alone shows that industries don't keep their system updated when vulnerabilities are discovered in the applications. A lot of known attacks still occur in many virtual infrastructures because software are not being updated. There were 229,000 average number of web attacks blocked per day in 2016. The administrators of these systems don't update their systems because of a number of concerns. The businesses can not afford to stop their service and patch the vulnerability and then start the service again. This downtime can cause them a large amount of loss in their business and possibly can result in loss of clients. So, the patching needs to be done while keeping the service available to be used by the clients. Another concern is that patching can cause a change in the resource usage by the system. This can slow down the service. Some extra resources might be required for software to run as it was before patching it. Thus, an overhead of patching is another concern when patching a system. Finally, the difficulty lies in the demand for human attention while updating the software. Updates may cause interruption in service or induce new bugs. Thus, developing an automated update system is a significant endeavor. The companies need to decide whether to patch their system and make it secure or to keep it vulnerable and leverage its current performance. Overall, to build a system for addressing all these problems, we need an automated patching software which analyzes the overhead patching places on a distributed system. This is the problem we aim to address. We seek to investigate this by considering the impact patching has on system metrics (like memory usage, etc) or response time to requests. We consider container systems because they usually denote a singular purpose. Our approach uses Kubernetes, a containerized application management system for automated deployments and scaling.

In summary, we achieve the following in this paper:

- We implement a dynamic shadow patching framework using Kubernetes with no change to the existing virtual infrastructure or interruption to its service.
- We illustrate the overhead of patching using 3 vulnerability case studies.

The system model we have used is described in the next section. Our implementation is detailed in the 4th section. We then present the experimental evaluations and results in the 5th and 6th sections. The 7th section is a discussion about the limitations and issues of our approach. It gives a list of possible things which can be implemented in the future work. The 8th section gives an overview of some related work done previously. The paper ends with a conclusion and acknowledgement.

3 SYSTEM MODEL

The system model of our implementation consists of a Kubernetes cluster, a reverse proxy and the users. Kubernetes is a container management system. A Kubernetes cluster uses a typical Master/Worker architecture. It has multiple containers running on the worker nodes. There are groups of these containers running on the same host which are called pods. A Kubernetes service is an abstraction which defines a logical group of pods and a method to access them. There are multiple types of services which can be made in Kubernetes. We have created nodePort services for our implementation.

By default, all the Kubernetes services are accessible at an internal IP called ClusterIP from inside the Kubernetes cluster only. This allows the applications running inside the pods to access the service. A nodePort service is accessible from outside the cluster. It reserves a port called nodePort on all the worker nodes and proxies that port to the pods selected by the service. This way, the users can access the service from outside the cluster using the master node's IP along with the nodePort of the service. The master node acts as a load balancer by forwarding the incoming traffic to the worker nodes in a round-robin fashion.

Figure 1 shows how our model looks like. We have a single Kubernetes master node and a service running on multiple worker nodes. This is the original service. There is another service called shadow service which is a duplicate of the original service. This is deployed on the new worker nodes which does not have the original service running. This shadow service does not exist initially when the system administrators detect a vulnerability in the application used by the service. The user accesses the service using a reverse proxy. We have used Nginx reverse proxy for our implementation.

The reverse proxy listens to a particular port where the users try to access the service. It forwards the traffic to the master node with the nodePort of the original service. This reverse proxy is also managed by the system administrator. We have used collectd to gather the system metrics of the target system which we want to analyze for the patching overhead. This collectd is installed in the worker node where the target container is running. It collects almost all the system metrics of the host on which it is running. We also install a collectd-web interface. This is used to monitor the live system metrics.

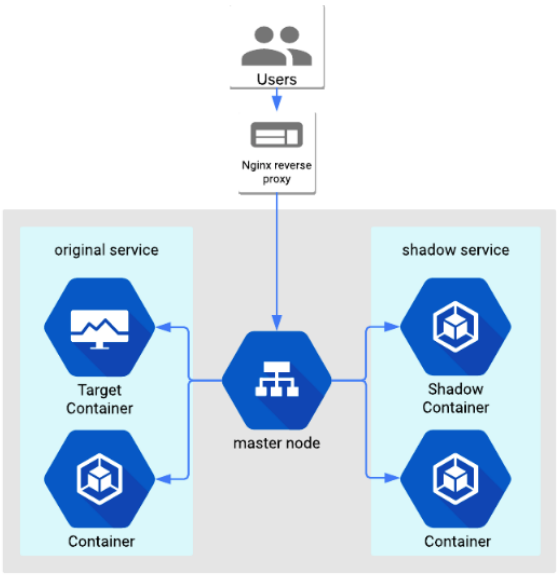


Figure 1: Architecture

4 APPROACH

We first create a distributed system with the help of Kubernetes. This system has a master node from which all the components of this distributed system are controlled. The master node creates a deployment on the worker nodes. It makes a NodePort service for this deployment so that users can access the application externally. We then create an additional node to install a reverse Nginx proxy. This reverse proxy forwards all the requests it gets on port 8080 to the master node with nodePort of the original service. The users of the system can now use the reverse proxy node's IP with port 8080 to access the service. This is the system state we have initially.

Once an administrator detects a vulnerability in an application used by this service, collectd is installed on the host where the vulnerable container is running. This starts collecting the metrics of the vulnerable application. We then deploy a duplicate of this original service on the worker nodes for using it as a shadow service. This step requires us to create extra worker nodes and join them in this Kubernetes cluster. The newly created service will have the containers created from the same image which was used for the original service, since this project limits the application category to stateless applications. So the state of an application running on the containers is not going to change in its lifetime. This allows us to make a shadow service using the same original image of the application. The shadow service will have a nodePort on which it is running. This nodePort is used at the reverse proxy node where the traffic was initially forwarded to the original service. The nodePort of original service is replaced by this new nodePort of the shadow service on the reverse proxy configuration file. After replacing it, the Nginx service is restarted. This is a small duration of time when the user requests will get queued and will not be processed till the Nginx service is back up and running. The point when Nginx gets restarted, the traffic gets forwarded to the shadow service instead of the original service. So, the containers running the original service would not have any users using it. This is when we patch the containers which are using the vulnerable application. The updated applications are then validated. Once the containers are patched, we give some time to stabilize the CPU and memory usage of the host machine. Then the user traffic is directed back to the original service by again changing the nodePort of service on the reverse proxy to the original service nodePort. We then restart the Nginx service. This is another small duration of downtime of the service. Once the service gets restarted, the user traffic is forwarded as it used to be originally to the original service.

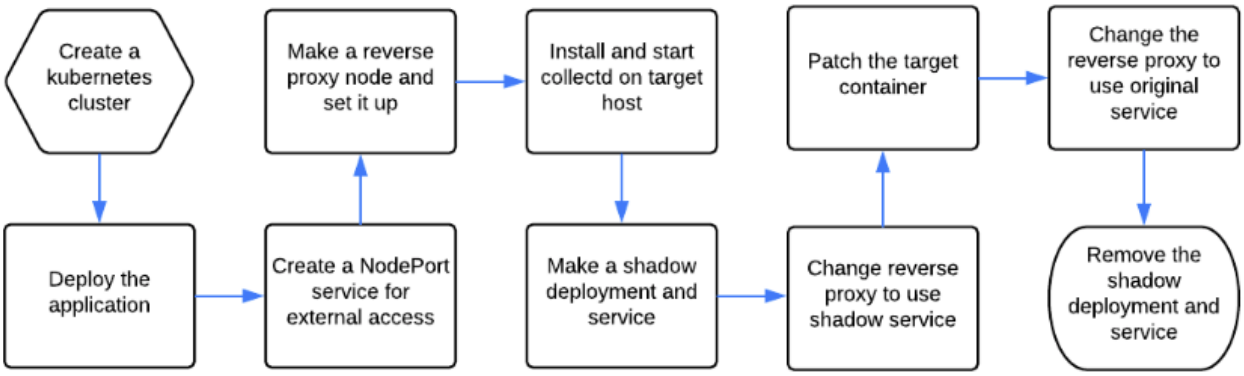


Figure 2: Patching Implementation Overview

The collected system metrics using collectd are then used to analyze the overhead of patching. The shadow deployment and service are removed once the patch found to be stable and successful. The overview of this implementation can be visualize in the flowchart shown in Figure 2.

5 EXPERIMENT

In this section, we discuss the setup of our design and evaluate the overhead in terms of CPU and Memory usage. The overall architecture is discussed in the System Model section with the aid of Figure 1. As in the figure, our experiments involve 1 master node and 2 nodes in each service. All nodes in the system use Ubuntu 16.04.

Experiment Setup

Client: We simulated client requests to the system with HTTP GET requests. Traffic is sent using the requests package of Python. This component sends between 5 to 10 requests per second. Each vulnerable environment runs an Apache server that responds to the client requests.

Reverse proxy: This reverse proxy was useful in sending client traffic to and returning responses from the appropriate Kubernetes service. The service is identified using the master's IP address and the service's assigned nodeport. For normal operation, before updating the specified container, traffic is handled by the original service. Whereas, control is switched to the shadow service while the container is updating.

Monitoring tool: We installed collectd to monitor system metrics and collectd-web to visualize them. In our evaluation we focused on CPU and Memory used metrics data.

Process

We observe system metrics for the target container before, during and after it is patched at 10 minute intervals. Note that during patching we stop requests to the original service. As mentioned in the System Model section, metrics are collected on the node machine where the container is situated. Initially, we observe system metrics prior to represent normal container activity. Here, the reverse proxy is set to give control to the original service in responding to client GET requests. The target container runs the appropriate vulnerable environment according the case studies described below. Next, we gather data during the patch. The reverse proxy is adjusted to the shadow service. Subsequently, the vulnerability is addressed with APT update commands and validated. The Validation is discussed for each vulnerability in the subsection below. Finally, the reverse proxy is tuned back to the original service in which the target container resides. Thus, system metrics are recorded for the period succeeding patching.

We monitored 3 vulnerability case studies to cover various impacts: Heartbleed, ShellShock and PHPMailer. This experiment process outlined above is repeated five times for each exposure to establish stable results. These cases reflect vulnerabilities of various intensities: according to CVSS Scores of 5, 10 and 7.5 respectively [2–4]

Vulnerability case studies

Heartbleed (CVE-2014-0160)

Heartbleed is a buffer overread vulnerability of OpenSSL. It allowed attackers to access sensitive data due to its lack of bound checking when returning a payload to a client in response to a heartbeat request. Thus, in our experiment client requests had to set the verify SSL certificate parameter of the python get function to false to showcase the OpenSSL

vulnerability. The studied vulnerable container image was sourced from [5].

Validation We inspected the openssl versions before at after the update. The version changed from 1.0.1e to 1.0.1t. We also verified the vulnerability had been fixed with an nmap script [6].

Shellshock (CVE-2014-6271)

Shellshock is an execute command type of vulnerability in GNU Bash. It enabled code inserted after environment variable definitions to be still operated on. That way, attackers have a means of running any commands. The susceptible image is provided to the public by [7].

Validation We validated the vulnerability with a sample bash script also contributed by [7]. The exploit takes advantage of the exposure as described above by adding print statements after the definitions that can be easily viewed. After the update, commands appended to environment declarations were ignored as expected. Likewise, We reviewed the Bash version. Although the issue was patched, the fix was released under the same version number.

PHPMailer (CVE-2016-10033)

This exposure of the PHP email transport library, PHPMailer, also lets attackers execute any commands. Malicious users could pass extra inputs to the library's mail command when no sender property is assigned by escape quote commands (\") in the Sender email address.

Validation The vulnerable Docker container image and exploit from [8] aided our exploration of this vulnerability. This case study was interesting because the exploit returns a shell to the user that can be leveraged to run code. We ran the exploit to confirm the initial container was initially exposed and then updated it at the appropriate period of the experiment. Thereafter, we ensured the patch was successful as the shell could no longer be returned.

6 RESULTS

To investigate the overhead of our patching method, we compare the pre-patch and post-patch performance in terms of overhead. We consider the CPU and Memory utilization to explore the extent to which changes occur.

The resulting memory and CPU usage of the target container which is patched is shown for each vulnerability case in the Figures 3-5. The red window outlines the time duration when the container is vulnerable. The yellow window highlights the period where the update patch is applied and container is allowed to for metrics to converge. Finally, patching is considered complete in the green window, representing post-patch events. We use these average values in the red and green windows in our calculation of overhead. These are highlighted in Tables 1-3.

Overall, we find that the update process does not introduce noticeable additional overhead to the machine on which the container runs. The details for each case study are described below.

Heartbleed (CVE-2014-0160)

Figure 3 shows the CPU utilization and Memory usage of the node during this patch process. The OpenSSL update that addresses Heartbleed involves the addition of code to carry out length checks. We expect to see an increase in both measures, on average. Comparing the red and green windows, we see that there is a slight growth in Memory usage. Whereas, CPU usage decreases by a small degree. The average numbers are summarized in Table 1. The update in the yellow window exerts a peak in both CPU and Memory used metrics. Thus, the node to be patched requires significant available CPU and Memory during this period; here, 50.1% of CPU and 220.95 MB of Memory.

Average usage		
	Vulnerable	Patched
CPU (%)	3.503	3.407
Memory (MB)	194.690	194.872

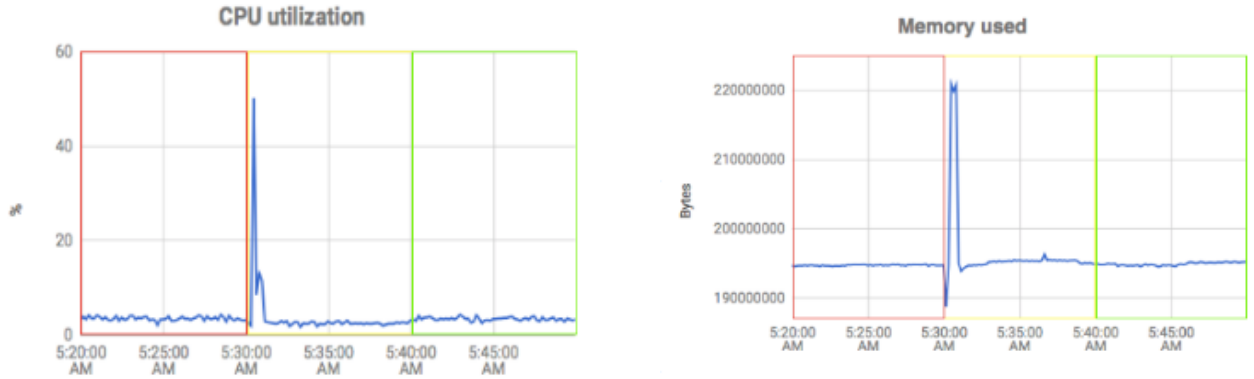
Table 1: Average metric usage of Heartbleed (CVE-2014-0160)

Average usage		
	Vulnerable	Patched
CPU (%)	2.153	2.167
Memory (MB)	188.976	186.868

Table 2: Average metric usage of Shellshock (CVE-2014-6271)

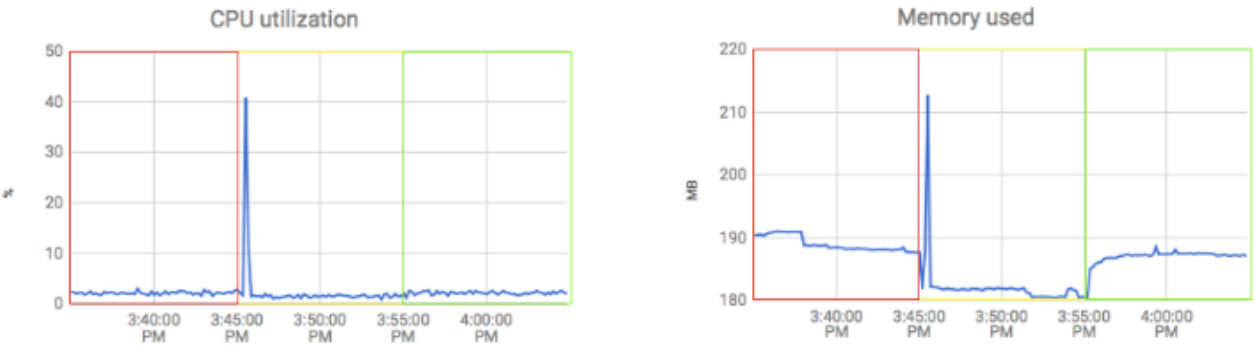
Average usage		
	Vulnerable	Patched
CPU (%)	3.450	3.397
Memory (MB)	188.626	181.332

Table 3: Average metric usage of PHPMailer (CVE-2016-10033)



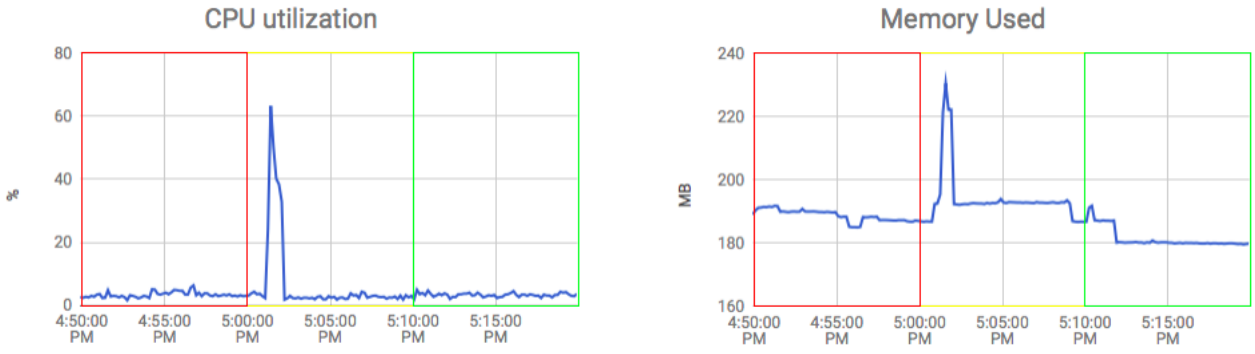
(a) CPU **(b) Memory**

Figure 3: Metrics of the Heartbleed (CVE-2014-0160) patch process



(a) CPU **(b) Memory**

Figure 4: Metrics of the Shellshock (CVE-2014-6271) patch process



(a) CPU **(b) Memory**

Figure 5: Metrics of the PHPMailer (CVE-2016-10033) patch process

Shellshock (CVE-2014-6271)
The metrics for Shellshock case are displayed in Figure 4. To ensure no additional commands are processed, the Shellshock patch introduces flags in parsing the environment

variable string along with some code to work with those flags. As before, we expect an elevated use of the relevant node's metrics. The post patch window exhibits a 0.65% incline in CPU. Whereas, there is a less pronounced reduction of 0.01% in average memory use. Table 2 reflects these comparisons. Furthermore, there is a more emphasized drop in the memory Used metric during the patch period, when not responding to requests, as compared to other periods. This suggests significant memory involved in processing requests.

PHPMailer (CVE-2016-10033)
Figure 5 highlights the PHPMailer results. The patch for the liable code sections involves passing the email sender inputs through a function to cleanup the culpable characters. Likewise, our expectations remained the same as in the prior cases. However, as Table 3 shows, there is a decrease in the average value of both metrics. CPU declines by 1.54%. Memory used drops by a more significant 3.87%.

7 DISCUSSION

Our implementation has some limitations. We will address some of the issues which we had faced while implementing this system. The first and foremost limitation of our approach is that it applies only to the stateless category of applications. We cannot use it with stateful applications. If it is used for stateful applications, the state of original service and shadow service will not remain in sync, which will cause inconsistency problems.

Initially, our plan was to make just one shadow container for replacing the target container when we are patching it. For keeping both target and shadow containers in sync, we were going to duplicate the HTTP requests coming to the target container to go to the shadow container as well and process that request on the shadow container so that there is no load on the target container and we can patch it. We tried updating the iptables of the master node for this purpose, but Kubernetes does not allow updating the iptable rules. If we add a new rule or update the existing ones, then it resets all the rules automatically which should be there for currently running services. We also tried using a proxy on the worker where the target container was running, but it could not listen to the port at which the container was getting requests because the application was already using that port. For dealing with this issue, we had to change the architecture of the system and make a whole new duplicate service to be used as a shadow service instead of making just one single container for replacing the target container.

Currently we have used a non consistent load on the service, which would not give a perfect measure of the resource usage. As a future work, we can use a tool such as Apache JMeter for generating some workload on the service. Also, we can analyze some more metrics like response time of the requests to better gauge overhead.

8 RELATED WORK

The papers listed below are some related work done in the area of dynamic patching. The topics in these papers include patch life-cycle, hot-patching and security patch management. Related work perform patching adapted to different kinds of systems. KARMA [9] addresses Android kernel updates. Others target server systems like Apache in work by Payer et al. [10]. Some systems work on web-based applications to initially handle detection of the vulnerability automatically before making a patch for it [11].

Methods of applying patching can be classified into offline and dynamic techniques. Many patching systems take the dynamic approach in applying their patches [9–15]. Live patching entails applying the updates during runtime. Updating code must be done when it safe to do so. For example, KARMA [9] ensures that no version of old functions are still running. Proteus [16] discusses a calculus approach of determining properties of safe state. These methods tend to be application specific work, however, as they involve detailed knowledge of their target system. Our work can be applied to any application running containers.

[17] is relevant to our work of shadow patching and an example of offline patching. It involves creating a Virtual Machine (VM) copy where tests can be made to determine stability of the patch. Afterwards, the original VM can be more easily updated using state information from the shadow VM. The update requires offline merging but reduces the downtime compared to that without a copy. Rather than on VMs, our work is focused on studying stateless applications on containers. Thus, we do not consolidate states and avoid the downtime it might demand.

9 CONCLUSION

We have presented a shadow patching system architecture and implementation for containers that is dynamic and application agnostic.
We leveraged this system to investigate patching overhead of the process using the apt package update method. Particularly, we explored CPU and Memory-used system metrics of containers with certain concrete vulnerabilities. Our results show that minimal or no overhead introduced after the update. Thus, this method is suitable for patching. Though the shadow mechanism involves duplication of existing service, it maintains availability of the service and avoids the complexity of modifying its infrastructure.

REFERENCES

[1] Internet security threat report (istr) 2017.
[2] Vulnerability details : Cve-2014-0160.
[3] Vulnerability details : Cve-2014-6271.
[4] Vulnerability details : Cve-2016-10033.
[5] Hmlio. hmlio/vaas-cve-2014-0160.
[6] nmap.org: ssl-heartbleed.
[7] opsxcq. opsxcq/exploit-cve-2014-6271, Sep 2017.
[8] opsxcq. opsxcq/exploit-cve-2016-10033.
[9] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security 17)*, 2017.
[10] Mathias Payer and Thomas R Gross. Hot-patching a web server: A case study of asap code repair. In *Privacy, Security and Trust (PST), 2013 Eleventh Annual International Conference on*, pages 143–150. IEEE, 2013.
[11] Hai Huang, Wei-Tek Tsai, and Yinong Chen. Binary analysis and automated hot patching for web-based applications. *Information and Software Technology*, 48(12):1148–1158, 2006.
[12] Suriya Subramanian, Michael Hicks, and Kathryn S McKinley. *Dynamic software updates: a VM-centric approach*, volume 44. ACM, 2009.
[13] Ashwin Ramaswamy, Sergey Bratus, Sean W Smith, and Michael E Locasto. Katana: A hot patching framework for elf executables. In *Availability, Reliability, and Security, 2010. ARES'10 International Conference on*, pages 507–512. IEEE, 2010.
[14] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus: A powerful live updating system. In *Proceedings of the 29th international conference on Software Engineering*, pages 271–281. IEEE Computer Society, 2007.
[15] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. *Practical dynamic software updating for C*, volume 41. ACM, 2006.
[16] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. *ACM SIGPLAN Notices*, 40(1):183–194, 2005.
[17] Duy Le, Jidong Xiao, Hai Huang, and Haining Wang. Shadow patching: Minimizing maintenance windows in a virtualized enterprise environment. In *Network and Service Management (CNSM), 2014 10th International Conference on*, pages 169–174. IEEE, 2014.