Insight from a Docker Container Introspection

Thomas Watts
Computer Science
School of Computing
University of South Alabama
thw1321@jagmail.southalabama.edu

Ryan G. Benton
Computer Science
School of Computing
University of South Alabama
rbenton@southalabama.edu

William Bradley Glisson
Cyber Forensics Intelligence Center
Department of Computer Science
Sam Houston State University
glisson@shsu.edu

Jordan Shropshire
Information Systems and Technology
School of Computing
University of South Alabama
jshropshire@southalabama.edu

Abstract

Large-scale adoption of virtual containers has stimulated concerns by practitioners and academics about the viability of data acquisition and reliability due to the decreasing window to gather relevant data points. These concerns prompted the idea that introspection tools, which are able to acquire data from a system as it is running, can be utilized as both an early warning system to protect that system and as a data capture system that collects data that would be valuable from a digital forensic perspective.

An exploratory case study was conducted utilizing a Docker engine and Prometheus as the introspection tool. The research contribution of this research is two-fold. First, it provides empirical support for the idea that introspection tools can be utilized to ascertain differences between pristine and infected containers. Second, it provides the ground work for future research conducting an analysis of large-scale containerized applications in a virtual cloud.

1. Introduction

The proliferation of cloud computing is rapidly expanding into all aspects of society. Investment in the public cloud space has gone from 58.6 billion dollars in 2009 to 219.6 billion in 2016 according to Statista [1]. Gartner's projections as of October 2017 have growth set to exceed 411 billion dollars by 2020 [2]. In 2017, SAP CEO Bill McDermott stated that "cloud revenue is expected to overtake license revenue for the first time" and that cloud revenue would more than double by 2020 to roughly 10 billion dollars [3]. In support of this prediction, Amazon Web Services [4] experienced a 43 percent jump in sales from 12 billion dollars in 2016 to slightly under 17.5 billion dollars in 2017 [5]. Additionally, Rightscale's 2018 State of the Cloud survey reported that the adoption of a public cloud rose 3 percent to 92 percent [6].

As investments expand and cloud services are integrated into all aspects of life, concerns surrounding the detection of security problems arise from both practitioners and academicians [7-11]. Emphasizing these concerns, Alert Logic, a big data security-as-a-service company, published a 2015 report that stated "businesses using cloud environments are largely considered a 'fruit-bearing jackpot' for hackers" [12]. Recent cases surrounding breaches Tesla [13] and FedEx [14] have underscored Alert Logic's stance. There was also a container specific problem reported by Ars Technica wherein a corrupted container was introduced that mined cryptocurrency. The malicious container was brought to light by a user on the popular programming code sharing website GitHub in August 2017 [15].

Even in light of recent security events, the use of containers is expanding. Containers share smaller operating system kernels, allowing faster and more efficient use of the hardware than hypervisors, which virtualize an entire machine [16]. RightScale indicated Docker [17] usage expanded from 35 to 49 percent [6]. Hortonworks, which publishes its Hortonworks Data Platform based on a distribution of Apache Hadoop [18, 19], has adopted Docker containers as part of Apache Hadoop YARN 3.1 in order to "enable new use cases and improve existing capabilities" within their platform [20]. The efficiency of a container comes at a cost, which can primarily be seen in terms of accessibility. Much like virtual machines, when containers are destroyed,

URI: https://hdl.handle.net/10125/60156
ISBN: 978-0-9981331-2-6
(CC BY-NC-ND 4.0)
HICSS

those system resources are put back into a resource pool. Worse, from a data collection point of view, a container's lifetime, from creation and destruction, can be a matter of seconds; this duration is sufficient for many to perform their function [21]. Hence, the data about containers must be accessed/collected while they are executing. In order to access this data, application programming interfaces (APIs) have been created and leveraged to create introspection tools [22, 23]. These introspection tools have the ability to obtain data from a running container environment regardless of running time.

The evolving atmosphere in cloud computing encourages organizations to consider cloud environments from a security perspective along with ways to improve incident response situations [24-27]. Trends in adoption of containers and increasing security prompts the hypothesis that introspection tools can be used as a data collection tool for an early warning system, as well as a forensic analysis tool, within a containerized system. Subsidiary questions identified as part of this research are as follows:

1. What data does an introspection tool have access to in a containerized environment?
2. How does it and how often does it log information?
3. How persistent is the log information?

The research contribution of this paper is an initial analysis of the viability of introspection tools for performing a security analysis of containerized software. The paper is structured as follows: Section 2 discusses the research surrounding cloud computing and the challenges presented by the cloud for detecting security problems. Section 3 presents the experimental methodology and design. Section 4 examines the results of a series of experiments designed to determine if an introspection tool can capture data for forensics analysis and early warning from a running containerized system. Section 5 draws conclusions and presents future work.

2. Related work

The structure of a cloud presents unique challenges to practitioners. While primarily focused on forensics in a cloud, O'Shaughnessy and Keane [28] expanded on several concepts that underpin problems in collecting data within a cloud environment, particularly with regard to what data is accessible to the parties operating a cloud. Two highlights of this point are chain of custody and multi-jurisdictional-legislation. Chain of custody is key in legal proceedings because it proves what parties had access to data that will be used as evidence in that proceeding [28]. A cloud system complicates a chain of custody since evidentiary data can be located in a different geographical locations and be collocated with other client data on a rack of servers. Chain of custody becomes even more complicated if the geographical location crosses jurisdictions, such as national borders. Depending on the jurisdictional change, it could be very difficult to even gather data for detecting anomalies either in a streaming environment, which is needed for early warning systems, or for post hoc analysis of a compromised system, which is needed for forensics. In response to these realities, recent research focuses on (a) building toolkits to circumvent some of the issues presented by O'Shaughnessy and Keane or (b) analyzing a cloud from the underlying hardware side.

Dykstra and Sherman [29] built a series of analysis tools for OpenStack, a cloud operating system [30] that accounts for 24% of private cloud adoption in 2017 [6]. Their tools, FROST, allow for a user to retrieve an image of virtual disks associated with that user's virtual machines and check both API requests and OpenStack firewall logs. These tools were built on-top of OpenStack and integrated into the Horizon dashboard which serves as the web-based user interface for OpenStack. The authors ran a pair of evaluations on FROST. The first evaluation involved 100 fictitious users with five virtual machines each; FROST utilized requested logs from a subset of those users. The second evaluation was based on a twelve-user test with a large private government cloud. The second evaluation was successful enough that the organization wanted to deploy FROST upon this cloud in mid-2013.

Saibharah and Greethaukumari [31] also used OpenStack but eschewed the idea of building tools directly into cloud platform; rather, they sought to use existing tools built into the platform. They built a framework based off of snapshots of both random-access memory and disk images, as well as working through logging systems native to OpenStack. Finally, the researchers extended their framework to incorporate network forensics. The authors tested this framework utilizing Wireshark [32] to gather network data and a purpose-built cloud process for the framework simulation. The evaluations showed that evidence could be obtained for several different types of attacks on a cloud environment.

In contrast to building toolkits, Graziano et al. [33] used a physical memory dump of a given system to identify if a hypervisor is present as well as identifying the type of hypervisor. Hypervisors virtualizing memory changes how that memory is allocated and accessed by the virtual machines that the hypervisor serves. The authors assert the only

way to gain access to that virtual memory is through analyzing the specific hypervisor, and then translating the rest of the memory over based on how that hypervisor handles the virtualization. The team developed a tool called Actaeon that extended the open source memory forensics framework Volatility [34]. The authors tested their plugin on a variety of hypervisors including Xen [35], KVM [36] and VMWare [37] and correctly identify all of the hypervisors in under a minute.

Whereas previous approaches dealt with hypervisors and/or underlying systems in cloud infrastructure, Casalicchio and Percibali [38] focused specifically on containers. In particular, they sought to determine if tools collected the same information. The researchers tested a battery of traditional Linux metrics including iostat and mpstat as well as utilizing the container specific cAdvisor [39] and the platform specific docker stats command to pipe metrics into both Prometheus [22] and Grafana [40] for collection. Tests centered upon CPU and Disk I/O intensive workloads. They determined different tools present similar but not completely equal results.

Previous approaches assumed there was no deliberate tampering, internal contradictions or inconsistent entries in their work; however, Thrope et al. [41] chose to focus their efforts on these areas. In order to test these issues, the group built a virtual machine profiler model and a log auditor to fill in potential gaps within the logs themselves. The authors tested their model and audit software with a series of four experiments that focused on execution timelines by a virtual machine on the hypervisor. Each timeline was subtly different in order to test system times as well as log manipulation. In one timeline, a Microsoft Word document's author was misattributed. The variance between execution timelines showed that temporal inconsistencies within a system, brought on issues such as differing system times, as well as log deletions, which could be detected by the forensic platform.

Shropshire [42] approached the problem of detecting anomalous behavior within a compromised cloud system from a hardware prospective. He developed PowerCheck, an application that identifies discrepancies by comparing the system state parameters reported by software running on a system with those parameters which application estimates based upon server energy consumption. The application predicted results of energy consumption on CPU load, memory consumed, and disk reads/writes among others. Once those predictions were made, PowerCheck analyzes data from a running system and does a comparison to determine if a cloud system has been compromised. The application was tested on VMWare ESXi [37] with power measurements made by a Watt's Up Pro Power meter, and performance reports provided to vSphere Client. The tests demonstrated the efficacy of PowerCheck and validated the idea of secondary system measures as legitimate integrity monitors.

One drawback with the pervious approaches seeking evidence of compromise is they assumed the operating system was not compromised. This was the motivation behind the next study. Zhang et al. [43] utilized BIOS level analysis to gain access to, and export out through a PCI card, memory contents and CPU registers that underpin a Xen hypervisor. Their application, HyperCheck, was different in that it does not rely on any software running on the target machine beyond a trusted BIOS. HyperCheck was tested against Xen through four attacks: modifying the interrupt descriptor table, the hypercall table, the exception table, and the Xen code itself. It was able to identify all of those modifications.

The previous studies concentrated on gathering behavior and performance evidence of various types; however, the scalability of the approaches was not directly assessed. Stelly et al. [44] dealt with this issue via the containerization of the digital forensics process with their SCARF toolkit. They focused on scalability across large platforms using Docker Swarm, and attempted to prove that while the data needed for forensic analysis continue to expand as cloud adoption increased, their platform could extend just as easily by showing high throughput. The group ran tests on both a legacy cluster, and a cluster with cutting edge hardware and found that several of the components of the SCARF system, such as Yahoo's OpenNSFW network [45], had large throughput gains comparing the two systems.

Cloud computing research on early warning systems and forensic data gathering tools currently focuses on designing tools that either extend the capabilities of the cloud control software such as FROST or taking advantage of certain virtualization properties to analyze specific virtual components. These systems struggle with containerized systems since many were focused on hypervisors which virtualize at a different level of the cloud, or heavily modify an underlying system. There has been minimal investigative research into how container introspection tools, whose systems only require an open port to function, can contribute to both early warning systems and forensic data investigation by polling and storing data

3. Methodology

This research investigates the viability of gathering behavioral (performance) data from containers using an introspection tool to detect problems within the system. This data can be used to set up an early warning system or store potential forensic evidence depending on the needs of an end user. The research according to Oates is classified as an exploratory study due to its attempt to understand the overall research problem [46].

3.1 Experimental testing environment

The experiment is conducted using Ubuntu 16.04 [47], Docker [17], and Prometheus [22]. The system utilizes an Intel Xeon E5 CPU with 16 gigabytes of RAM. Docker is open source software that runs on top of a host operating system, and as such, has an engine associated with it. The engine handles communication with repositories to pull container images, as well as provides administrative oversight to the containers under its control. Figure 1 illustrates where Docker, the container engine, is in a virtual environment.

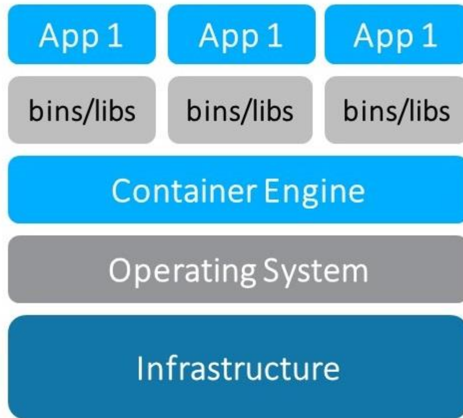

Figure 1. Docker location [48]

Prometheus is an open source introspection tool that provides the ability to check multiple nodes in a containerized architecture through console readouts or graphs from several hundred metrics on both the Prometheus server, as well as targets that are connected to that server. Prometheus utilizes the Docker API, which allows Prometheus to access data via a well-defined data pipe; this also mitigates the amount of stress on the system. In order to use the service, ports have to be opened within the Docker environment, as well as configured within Prometheus to listen and scrape information from the Docker environment. The default time between information scrapes by Prometheus is fifteen seconds, but that can be customized via the modification of Prometheus configuration files. Prometheus provides a query language to create specific metrics about the Docker environment, which can be manually reviewed; the same metrics can also be presented via a series of graphic functionalities for easier infrastructure visualization. Figure 2 shows a list of http request metrics that Prometheus gathers when running the default configuration; the system will, in addition to metrics, print some guiding comments as seen in the first two lines of Figure 2. By default, Prometheus saves these metrics every two hours in data chunks. The data chunks can be stored in a local file or into a database server, which can be on another server; these options can be configured. Hence, in case of an unexpected shutdown, the data that would be lost, at most, is the current set of data in memory. One final advantage Prometheus has is that it can execute on a server other than Docker.

```
# HELP http_request_size_bytes The HTTP request sizes in bytes.
# TYPE http_request_size_bytes summary
http_request_size_bytes{handler="alertmanagers",quantile="0.5"} NaN
http_request_size_bytes{handler="alertmanagers",quantile="0.9"} NaN
http_request_size_bytes{handler="alertmanagers",quantile="0.99"} NaN
http_request_size_bytes_sum{handler="alertmanagers"} 0
http_request_size_bytes_count{handler="alertmanagers"} 0
http_request_size_bytes{handler="alerts",quantile="0.5"} NaN
http_request_size_bytes{handler="alerts",quantile="0.9"} NaN
http_request_size_bytes{handler="alerts",quantile="0.99"} NaN
http_request_size_bytes_sum{handler="alerts"} 0
http_request_size_bytes_count{handler="alerts"} 0
http_request_size_bytes{handler="clean_tombstones",quantile="0.5"} NaN
http_request_size_bytes{handler="clean_tombstones",quantile="0.9"} NaN
http_request_size_bytes{handler="clean_tombstones",quantile="0.99"} NaN
http_request_size_bytes_sum{handler="clean_tombstones"} 0
http_request_size_bytes_count{handler="clean_tombstones"} 0
http_request_size_bytes{handler="config",quantile="0.5"} NaN
http_request_size_bytes{handler="config",quantile="0.9"} NaN
http_request_size_bytes{handler="config",quantile="0.99"} NaN
```

Figure 2. Example prometheus metrics

3.2 Experimental methodology

The experiment has three phases: (a) the initial setup required to run Prometheus and Docker, (b) acquiring data for the baseline, and (c) executing and acquiring data when the system is in an infected state. A fourth phase is also presented, which describes what data will be analyzed.

3.2.1 Initial Setup. To create the data collection environment, Ubuntu, Prometheus and Docker have to be installed on a machine. The steps for the initial setup are as follows. Download the latest Ubuntu image from Canonical's website [47]. This experiment utilized Ubuntu image version 16.04.3. Load the downloaded image onto a USB Flash drive and leave the USB drive plugged into the PC.

Restart the machine, and, at the BIOS screen, press the necessary key to enter the BIOS settings. Navigate through BIOS to the boot order screen, and make sure that USB drives are checked in the boot order ahead of the primary hard drive of the

experimental platform. Exit BIOS which will restart the machine, and boot into the Ubuntu installation suite. At the first screen of the installation process, select "Install Ubuntu". Allow Ubuntu to download updates while installing on the next screen.

The following screen sets installation type, and it is at this point where the disk will be formatted. Select "Erase disk and install Ubuntu" and then click continue in the pop-up window. At this point, the installation will shift to configuration details, such as time zone, on the next screen. The experiment was built in Mobile, Alabama at the University of South Alabama, which is in the Central time zone, so Chicago is appropriate. Select language and keyboard type on the following screen. The platform used English (US) as the language, and QWERTY as its keyboard.

Next, set a default username and password that will automatically have administrator privileges, and Ubuntu will install over the course of several minutes, and then restart itself to complete the installation. Once Ubuntu has restarted, the first step to install Docker from a repository [17] is to update apt package index. Enter the command "sudo apt update". Once apt is updated, configure it to be used over HTTPS using "sudo apt install apt-transport-https ca-certificates curl software-properties-common". Add Docker's Gnu Privacy Guard (GPG) key using: *"curl –fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add-"* Build the repository for Docker by inputting: *"sudo add-apt-repository" deb [arch= amd64] https://download.docker.com/linux/ubuntu $(ls b_release -cs) stable"*. Given these new packages, update apt again using "sudo apt update" and then install Docker with "sudo apt install docker-ce". This will complete the Docker installation.

In order to allow Prometheus to connect to Docker, a JSON file has to be added in the /etc/docker/ file. Since this is a new installation of Docker, this has to be created through a text editor such as nano. Open nano using "nano daemon.json" and then add "{"metrics-addr" : "127.0.0.1:9323", "experimental" : true }" to this new file. Save this file in the /etc/docker/ directory.

Prometheus provides precompiled binaries on their website [22]. These binaries are tarballs, so extract it using "tar xvfz prometheus-*.tar.gz" and then change into the newly extracted files with "cd prometheus-*". Modify the prometheus.yml to scrape Docker by adding "job_name: 'docker' and static_configs: -targets: ['localhost:9323] in the appropriate sections of the yml file. Alternatively, Docker provides an updated yml file within their Prometheus documentation [17].

Next a containerized Apache2 server is obtained and deployed on the experimental platform. Apache2 is obtained from the Docker Hub [45]. The command to start Redis is "docker run –dit –name apache2testbed –p 8080:80 –v "$PWD":/usr/loca/apache2/htdocs/ httpd:2.4."

Once Apache2 has been installed through Docker, navigate a browser to localhost:9323/metrics to get to the Prometheus dashboard. Prometheus provides two hundred thirty metrics from installation and can be configured with additional rules by a user. This methodology focused on metrics found at installation. In order to simulate traffic on a web server, ApacheBench [49], a stress test tool, was used to simulate hits on the server. ApacheBench is part of the apache2-utils package and is installed with the "sudo apt install apache2-utils" command.

3.2.2 Gathering data for the benign scenario. Now that Apache2 web server has been installed, Prometheus can gather metrics based on user defined scrape times. The collected data is kept in memory until written to file every two hours. In order to avoid any potential issues, a pristine Apache2 web server was run for two hours, and ApacheBench is used to push one hundred thousand hits on the server every twenty minutes to provide traffic. The ab command is "ab –k –c 100 –n 100000 <web server IP>." The web server IP can be found through ifconfig. After two hours of collecting metrics, the data is saved for comparison against the malware infected web server.

3.2.3 Gathering data for the infected scenario. This scenario is similar to the benign scenario with the exception that the Apache2 web server is infected with malware. The steps in subsection 3.2.1 are repeated to reset the testbed, and a Chapro binary, a malicious Apache webserver module, is downloaded from a malware repository and run on the system. The two-hour test is repeated with attendant ApacheBench hits every twenty minutes. Metrics are pulled to provide comparison between the two platforms. This ends the gathering of the data.

3.2.4 Data analysis from the two scenarios. Once the three phases are done, an analysis of the two data sets using the metrics provided by Prometheus needs to be conducted. As previously stated, Prometheus provides two hundred and thirty metrics in its initial configuration. The analysis of these metrics is a simple comparison between the values given by the two testbeds, which is presented in the next section.

4. Results and discussion

The results presented below are from multiple runs; there is a total of twenty hours of data. The tables within this section capture the results of each run and the averages of the five separate runs, as well as the percentage differences between the clean and contaminated versions of the Apache2 web server.

After an initial examination, nine metrics were identified that have utility in detecting the presence of malware. These nine metrics are grouped into three categories. The first provides information about the underlying engine, the second provides information about the memory usage, and the third provides information about the process and HTTP requests.

4.1. Daemon engine

Prometheus is able to gather data about the underlying Docker engine. The first metric that has been identified, engine_daemon_engine_info, provides specific information on Docker itself and demonstrates that the two experiments are performed on the same Docker engine. This would be useful for a forensic investigation because it would document the Docker system information and also aid in ensuring any simulations would be performed on the correct Docker container engine.

The other two metrics focused on providing information about the Docker engine were identified as potentially being useful. Shown in Table 1 is engine_daemon_engine_memory_bytes; this metric captures how much memory the Docker container has allocated from the host upon which the Docker engine sits. As can be seen, the metric indicates that the infected containers have been assigned more resources from the resource pool. This makes sense, as malware would take additional memory above and beyond that needed by the base service.

Table 1. Engine_daemon_engine_memory_bytes

Experiment	Clean	Infected	% Difference
1	2985674229	3239857152	9%
2	3200538134	3511309584	10%
3	3405294587	3549809237	4%
4	3009478856	3353498206	11%
5	3159152434	3419008911	8%
Average	3152027648	3414696618	8%

The last metric, Engine_daemon_events_total, shown in Table 2, describes the number of events that the engine handled throughout a test. As can be seen, there is a stark difference between the two testbeds since it specifically references the number of events that are logged by the Docker engine. The extra program running within the Apache2 web server on the infected container that was deployed had several more interactions with the Docker Engine throughout the battery of tests.

Table 2. Engine_daemon_events_total

Experiment	Clean	Infected	% Difference
1	32	39	21%
2	33	39	18%
3	37	45	21%
4	35	37	5%
5	33	40	21%
Average	34	40	17%

4.2. Go memory statistics

Prometheus is written in Go [50] , an open source programming language, and one of the modules that the Prometheus provides focuses on memory allocation. Where the Docker engine metrics shown in section 4.1 focus on the engine and how it handles events, the three Go metrics focus on how memory is allocated within the environment. The first memory metric is go_memstats_alloc_bytes_total, which is the total count of bytes allocated to a server. As in Table 3, the infected containers use more memory than the clean. This is not surprising as the malware is conducting additional work, requiring more memory. The total difference varies between experiments; with experiment 5 having the lowest. The average difference overall is fairly high; however, the metric may not be sufficient by itself.

Table 3. Go_memstats_alloc_bytes_total

Experiment	Clean	Infected	% Difference
1	1314183517	1432935864	9%
2	1714271435	1944971723	13%
3	1809472402	2053409271	13%
4	1455934827	1699468245	16%
5	1593068259	1633482952	2%
Average	1577586088	1752873431	11%

The second metric is go_memstats_frees_total, which shows the amount of free requests that are performed. As seen in Table 4, the memory freed is generally greater for the infected container; however, similar to the previous metric, the differences vary greatly between experiments. It is noteworthy that the difference observed are lower than that found in the

previous memory metric, go_memstats_alloc_bytes_total. Hence, this metric is unlikely to be sufficient by itself to create early warnings. It would be a good indicator in a post forensic analysis for detecting when infections began changing the system.

Table 4. Go_memstats_frees_total

Experiment	Clean	Infected	% Difference
1	163795705	189523901	15%
2	206526775	215692761	4%
3	214580257	224596002	4%
4	169934501	192180500	13%
5	164373404	167924960	2%
Average	183842128	197983614	8%

Table 5 shows the third metric, go_memstats_heap_released, which shows how much memory is released to the host operating system by the Docker processes. Of the three, this shows a clear performance difference, with the lowest difference being 12%. A simple threshold, assuming a good baseline exists, would be sufficient to trigger an alarm. However, this assumes no major variations are expected; otherwise, this and the other two memory statistics would be inputs into a more sophisticated analysis system.

Table 5. Go_memstats_released_bytes_total

Experiment	Clean	Infected	% Difference
1	3706530	4199240	13%
2	3852390	4723000	22%
3	4059500	4855900	19%
4	3523560	4024010	14%
5	3290020	3701850	12%
Average	3686400	4300800	16%

4.3. Process & http requests

The last set of metrics gather data about the system running underneath Docker. The first metric, process_max_fds, captures the maximum number of open file descriptors. As seen in Table 6, there is a general trend that the infected system has more files open; however, this can vary greatly, and, in some experiments, the difference is negligible. It is unclear, at this time, why there is a wide range of differences. Given the difference, this cannot be used by itself as an early warning system; however, it could be useful when combined with other metrics. A forensics analyst could use it as an indicator to detect when changes occurred assuming a normal baseline exists.

The second metric, process_virtual_memory_bytes, refers to how much virtual memory is used by the overall Docker containers. As seen in Table 7, the underlying system must allocate more memory when infected containers are executed. This indicates that the malware is requiring a fair amount of memory to be available for use. What is interesting is that experiment 5 shows a high virtual memory usage, even when some of the more individual memory usage in the Docker containers were lower.

Table 6. Process_max_fds

Experiment	Clean	Infected	% Difference
1	1091797	1105923	1%
2	972579	1117239	14%
3	1095761	1192469	8%
4	1089851	1180307	8%
5	992892	1152971	16%
Average	1048576	1149782	9%

Table 7. Process_virtual_memory_bytes

Experiment	Clean	Infected	% Difference
1	560294865	623246092	11%
2	593261704	683256017	15%
3	642359760	691269132	7%
4	602874102	671296123	11%
5	531242209	616019356	15%
Average	586006528	657017344	12%

The last metric is the number of HTTP requests handled by the system, which is represented by http_requests_total. As seen in Table 8, the number of HTTP requests are greater for the infected containers. This is expected as the malware injects an iframe into the content provided by Apache2. Given the general consistency in the numbers, this could be a good indicator for an early warning system and would be telling for a forensic investigation.

Table 8. Http_requests_total

Experiment	Clean	Infected	% Difference
1	731039	799125	9%
2	779683	852301	9%
3	802359	882559	10%
4	650048	772471	18%
5	752941	827619	9%
Average	743214	826815	8%

4.4. Other Prometheus metrics

The nine metrics in this study looked primarily at memory and communication between multiple planes of a virtual system. However, Prometheus gathers other metrics by default that describe CPU activity, IO activity and operating events. Additionally, Prometheus is able to obtain metrics from other APIs besides those provided by Docker. It can poll metrics from entire cloud systems such as Amazon's Amazon Web Services [4] or Microsoft's Azure [51]. Furthermore, system administrators can build custom rules for their individual systems. This allows tools such as Prometheus to be used to detect multiple types of problems in a many various environments. Hence, more elaborate data gathering could be conducted, as well as the capturing of data relevant for detection of other problems and situations.

5. Conclusions and future work

This research proposed three subsidiary questions to determine if an introspection tool could serve as an early warning system and forensics tool in a virtual environment. The first investigates what data an introspection tool can access. Prometheus, the chosen introspection tool for this study, can capture performance information about the containers, the Docker Engine, and the host OS. This includes I/O, memory statistics, and operating events. The Prometheus documentation indicates that histogram information can be generated instead of single counters and/or measurements. This information is useful for monitoring and early warning systems. For forensic investigations, it can confirm when malware was running, and when it began; this, would aid in determining when the malware was installed.

The second subsidiary question is how often is the data collected? In Prometheus, collecting data is a configurable option. By default, it is 15 seconds; this can be tuned to ensure that the collection is not burdensome upon the system. It collects the data by pulling information from the Docker Engine through its API, which requires a port for pulling to be opened for communication. While opening ports does lead to potential risks, this can be mitigated by controlling what servers are allowed to access the Docker Engine via the known port.

The final subsidiary question dealt with the persistence of the log information. As discussed in the methodology, Prometheus can store data as local files or in a database, either of which can be on a separate server. This does come with a caveat; Prometheus does allow for the user to specify how long it should run before writing the data to file or to a database. Data held solely in memory is subject to being lost, if Prometheus is shut down prematurely.

The answers to the subsidiary questions provide support for the hypothesis that introspection tools can be used as a data collection tool for an early warning system, as well as a forensic analysis tool, within a containerized system. Of the default metrics, nine metrics were identified as being immediately useful, as they provided either human distinguishable differences or they provided useful forensics information. engine_daemon_engine_info is a metric that would be of interest to forensics investigators as it would help confirm the exact version of Docker engine being executed. The other eight metrics permitted the operator to determine if Apache2 was corrupted via simple percentage differences. While individual metrics may not be sufficient for an early alert system, in combination, they tend to compensate for when other measures are weak.

More interesting, the nine metrics captured different types of performance: the engine itself, the memory usage of the containers, and information of the underlying host. Thus, any malware seeking to corrupt the Docker Engine to hide its existence would need to determine the typical usage patterns; this, in turn, leads to other opportunities for the monitoring system to detect the malware presence.

Future work can be split into three major efforts. The first effort would be conducting more complex studies using Prometheus to detect different types of malware and/or system compromises. It is unlikely all malware and compromises will be detectable with the same nine metrics. Thus, future work will require determining what metrics are useful for detecting different types of attacks. It will also expand the underlying test bed system to utilize larger numbers of containers and interactions between containers, such as how Apache Hadoop YARN [20] has done to streamline some data processing. Determining what time intervals would be best for Prometheus to collect information would be part of this work.

The second effort focuses on taking the data from Prometheus and creating an alerting system. At present, Prometheus can create simple alerting rules, which activate when certain thresholds are violated. This is often insufficient when determining if a system has been compromised. Hence, the data would need to be feed into online systems [52], where anomaly detection methods could be used to detected unexpected behavior. This can be done with using methods such as simple learning systems, such as the Hierarchical Temporal Memory method [53], DBScan [54], or generalized linear models [55].

The third effort will develop forensic tools to find suspicious behavior within the introspection logs and correlate them with other factors. For instance, Rightscale's yearly State of the Cloud surveys [6] underscore how necessary it is to provide different analytical mechanisms.

6. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1726069.

7. References

[1] Statista. *Public cloud services: market size 2009-2020 | Statistic*. 2018; https://www.statista.com/statistics/273818/global-revenue-generated-with-cloud-computing-since-2009/.

[2] van der Meulen, R. and C. Pettey, *Gartner Forecasts Worldwide Public Cloud Services Revenue to Reach $260 Billion in 2017*. 2018.

[3] Evans, B. *Inside SAP: As Cloud Surpasses License Revenue In 2018, 10 Strategic Insights*. 2018.

[4] *Amazon Web Services (AWS) - Cloud Computing Services*. 2018; https://aws.amazon.com/.

[5] Peterson, B. *Amazon Web Services is now a $17.5 billion business*. 2018.

[6] Weins, K., *Cloud Computing Trends: 2018 State of the Cloud Survey*. 2018.

[7] Ab Rahman, N.H. and K.-K.R. Choo, *A survey of information security incident handling in the cloud.* Computers & Security, 2015. **49**: p. 45-69.

[8] Agrawal, B., T. Wiktorski, and C. Rong, *Adaptive real-time anomaly detection in cloud infrastructures.* Concurrency and Computation: Practice and Experience, 2017. **29**(24).

[9] Osanaiye, O., K.-K.R. Choo, and M. Dlodlo, *Distributed denial of service (DDoS) resilience in cloud: review and conceptual cloud DDoS mitigation framework.* Journal of Network and Computer Applications, 2016. **67**: p. 147-165.

[10] Cahyani, N.D.W., N.H.A. Rahman, W.B. Glisson, and K.-K.R. Choo, *The Role of Mobile Forensics in Terrorism Investigations Involving the Use of Cloud Storage Service and Communication Apps.* Mobile Networks and Applications, 2017. **22**(2): p. 240-254.

[11] Grispos, G., W.B. Glisson, and T. Storer, *Chapter 16 - Recovering residual forensic data from smartphone interactions with cloud storage providers*, in *The Cloud Security Ecosystem*, R.K.-K.R. Choo, Editor. 2015, Syngress: Boston. p. 347-382.

[12] Palmer, D. *Hackers see cloud as 'a fruit-bearing jackpot' for cyber attacks | Computing*. 2015.

[13] Team, R.C.S.I., *Lessons from the Cryptojacking Attack at Tesla*. 2018, @Redlockio.

[14] Diachenko, B. *FedEx Customer Records Exposed*. 2018.

[15] Goodin, D. *Backdoored images downloaded 5 million times finally removed from Docker Hub*. 2018; https://arstechnica.com/information-technology/2018/06/backdoored-images-downloaded-5-million-times-finally-removed-from-docker-hub/.

[16] Chae, M., H. Lee, and K. Lee, *A performance comparison of linux containers and virtual machines using Docker and KVM.* Cluster Computing, 2017: p. 1-11.

[17] *Docker*. 2018.

[18] Backaitis, V., *Big Data Crushers Have Peaked For Now, Regardless Of Hortonworks' Upcoming Earnings.* 2018.

[19] *Welcome to Apache Hadoop!* 2018; http://hadoop.apache.org/.

[20] Shane Kumpf, V.K.V., Saumitra Buragohain, *Trying out Containerized Applications on Apache Hadoop YARN 3.1 - Hortonworks*. 2018.

[21] Vaughan-Nichols, S.J. *What is Docker and why is it so darn popular?* 2018; https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/.

[22] *Prometheus - Monitoring system & time series database*. 2018.

[23] Datadog, *Infrastructure & Application Monitoring as a Service | Datadog.* 2015.

[24] Grispos, G., W.B. Glisson, and T. Storer, *Cloud security challenges: Investigating policies, standards, and guidelines in a fortune 500 organization.* arXiv preprint arXiv:1306.2477, 2013.

[25] Grispos, G., W.B. Glisson, and T. Storer, *Security incident response criteria: A practitioner's perspective.* arXiv preprint arXiv:1508.02526, 2015.

[26] Grispos, G., W.B. Glisson, and T. Storer, *Enhancing security incident response follow-up efforts with lightweight agile retrospectives.* Digital Investigation, 2017. **22**: p. 62-73.

[27] Grispos, G., T. Storer, and W.B. Glisson, *Calm before the storm: the challenges of cloud.* Emerging digital forensics applications for crime detection, prevention, and security, 2013. **4**(1): p. 28-48.

[28] O'shaughnessy, S. and A. Keane. *Impact of cloud computing on digital forensic investigations*. in *IFIP International Conference on Digital Forensics*. 2013. Springer.

[29] Dykstra, J. and A.T. Sherman, *Design and implementation of FROST: Digital forensic tools for the OpenStack cloud computing platform.* Digital Investigation, 2013. **10**: p. S87-S95.

[30] *What is OpenStack?* 2018; https://www.openstack.org/software/.

[31] Saibharath, S. and G. Geethakumari. *Design and Implementation of a forensic framework for Cloud in OpenStack cloud platform*. in *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*. 2014. IEEE.

[32] *Live View*. 2018; http://liveview.sourceforge.net/.

[33] Graziano, M., A. Lanzi, and D. Balzarotti. *Hypervisor memory forensics*. in *International Workshop on Recent Advances in Intrusion Detection*. 2013. Springer.

[34] *The Volatility Foundation - Open Source Memory Forensics*. 2018; http://www.volatilityfoundation.org.

[35] *VS16: Video Spotlight with Xen Project's Lars Kurth*. 2018; https://www.xenproject.org/.

[36] *KVM*. 2018; https://www.linux-kvm.org/page/Main_Page.

[37] *VMware – Official Site*. 2018; https://www.vmware.com.

[38] Casalicchio, E. and V. Perciballi. *Measuring docker performance: What a mess!!!* in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. 2017. ACM.

[39] *google/cadvisor - Docker Hub*. 2018; https://hub.docker.com/r/google/cadvisor/.

[40] *Grafana - The open platform for analytics and monitoring*. 2018; https://grafana.com/.

[41] Thorpe, S., I. Ray, T. Grandison, A. Barbir, and R. France. *Hypervisor event logs as a source of consistent virtual machine evidence for forensic cloud investigations*. in *IFIP Annual Conference on Data and Applications Security and Privacy*. 2013. Springer.

[42] Shropshire, J. *Securing cloud infrastructure: unobtrusive techniques for detecting hypervisor compromise*. in *ICCSM2015-3rd International Conference on Cloud Security and Management: ICCSM2015*. 2015. Academic Conferences and publishing limited.

[43] Zhang, F., J. Wang, K. Sun, and A. Stavrou, *Hypercheck: A hardware-assistedintegrity monitor.* IEEE Transactions on Dependable and Secure Computing, 2014. **11**(4): p. 332-344.

[44] Stelly, C. and V. Roussev, *SCARF: A container-based approach to cloud-scale digital forensic processing.* Digital Investigation, 2017. **22**: p. S39-S47.

[45] Docker. *Explore - Docker Hub*. 2018; https://hub.docker.com/explore/.

[46] Oates, B.J., *Researching information systems and computing*. 2005: Sage.

[47] *Ubuntu 16.04.3 LTS*. 2018.

[48] Labs, R., *Playing Catch-up with Docker and Containers*. 2017, @Rancher_Labs.

[49] *ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4*. 2018; https://httpd.apache.org/docs/2.4/programs/ab.html.

[50] Donovan, A.A. and B.W. Kernighan, *The Go programming language*. 2015: Addison-Wesley Professional.

[51] *Microsoft Azure Cloud Computing Platform & Services*. 2018; https://azure.microsoft.com/en-us/.

[52] Abbady, S., C.-Y. Ke, J. Lavergne, J. Chen, V. Raghavan, and R. Benton. *Online mining for association rules and collective anomalies in data streams*. in *Big Data (Big Data), 2017 IEEE International Conference on*. 2017. IEEE.

[53] Ahmad, S., A. Lavin, S. Purdy, and Z. Agha, *Unsupervised real-time anomaly detection for streaming data.* Neurocomputing, 2017. **262**: p. 134-147.

[54] Thang, T.M. and J. Kim. *The anomaly detection by using dbscan clustering with multiple parameters*. in *Information Science and Applications (ICISA), 2011 International Conference on*. 2011. IEEE.

[55] Behzadi, S., K. Hlavácková-Schindler, and C. Plant. *Dependency anomaly detection for heterogeneous time series: A Granger-Lasso approach*. in *Data Mining Workshops (ICDMW), 2017 IEEE International Conference on*. 2017. IEEE.