Dynalize: Dynamic Analysis of Mobile Apps in a Platform-as-a-Service Cloud

Pablo Graubner, Lars Baumgärtner, Patrick Heckmann, Marcel Müller, Bernd Freisleben
Department of Mathematics and Computer Science, University of Marburg
Hans-Meerwein-Str. 6, D-35032 Marburg, Germany

Email: {graubner,baumgaertner,heckmann,muellerm,freisleb}@informatik.uni-marburg.de

Abstract—Ensuring the software quality of mobile applications with respect to performance, robustness, energy consumption, security and privacy is an important problem for a growing researcher and developer community. In this paper, we present Dynalize, a Platform-as-a-Service cloud for the dynamic analysis of mobile applications. It allows researchers and developers to investigate mobile applications at runtime in a virtual device cloud and to publish the performed analyses as web services. In contrast to existing approaches, it makes use of container virtualization on top of Infrastructure-as-a-Service instances, enabling dynamic provisioning and fast deployment of dynamic analyses. A custom container layout and a novel storage solution on the virtual server layer ensures cost- and runtime-efficient large-scale analyses of thousands of apps. The applicability of Dynalize is demonstrated by a security analysis of about 6,000 Android applications. Experiments on container startup, virtual device to container throughput and different storage backends show the feasibility of the proposed approach.

I. INTRODUCTION

Ensuring the software quality of mobile applications (i.e., mobile apps) with respect to performance, robustness, energy consumption, security and privacy is a challenging problem for developers within a competitive business. This problem fostered research on static and dynamic analysis of mobile apps. While static analysis is typically used to scan the code of a mobile app for programming-related issues affecting efficiency and reliability, dynamic analysis is applied to observe runtime properties of a mobile app that is executed on a physical or virtual device. Nowadays, there is a need to analyze a potentially large number of mobile apps developed for a variety of mobile devices and environments.

There are several tools that examine the runtime properties of mobile apps in order to evaluate bugs [23], performance issues [24] or to detect malware [22]. Furthermore, there are generic frameworks such as PUMA [11] that provide support for dynamic app analysis, but not with respect to scalability and dynamic resource provisioning. On the other hand, Device-as-a-Service (Daas) providers such as Genymotion [9], Manymo [18] and Testdroid [5] assume that a human tester controls devices remotely. These approaches are aimed at supporting the development and evaluation of single apps on a small number of virtual or physical devices in parallel.

In this paper, we propose Dynalize, a novel Platform-as-a-Service (PaaS) cloud for dynamic analysis of mobile apps. Dynalize provides a scalable platform for dynamic app analysis, allows developers to integrate existing tools with only slight modifications and can be used to publish analysis results as Software-as-a-Service (SaaS) offerings. Therefore, Dynalize provides (i) a container management/job processing engine that serves as a layer between Infrastructure-as-a-Service (IaaS) and PaaS, and (ii) a service platform for the dynamic analysis of mobile apps.

Dynalize has the following features:

1) It runs virtual devices on top of IaaS instances, which provides scalability and dynamic resource provisioning.
2) It makes use of containerization and container virtualization, which allows developers to execute existing tools with heterogeneous dependencies regarding libraries and programming languages.
3) It provides a storage architecture for mobile apps, which can be used to distribute and process large amounts of data for thousands of apps in an efficient and coordinated manner.
4) It offers users (i) one-time and (ii) long-term provisioning, which enables users to either (i) start and terminate IaaS instances on demand or (ii) keep IaaS instances running in order to make use of local caches.
5) It offers a web service frontend to users, which allows users to publish their results as a SaaS service.

The paper is organized as follows. The Dynalize platform design and implementation are described in Section II. Experimental results are presented in Section III. Section IV discusses related work. Section V concludes the paper and outlines areas of future work.

II. DYNALIZE

Dynalize consists of dynamically provisioned IaaS instances, a central Scheduler Node and a Service Node (see Figure 1). The components of Dynalize are described below.

A. IaaS Instances

Figure 2 shows the layout of an IaaS instance provisioned by Dynalize to analyze mobile apps. On the left side, the relation between data provider, local cache daemon and the container is shown. On the right side, the relation between global scheduler, local scheduler and local cache daemon is illustrated. Basically, the global scheduler partitions the total set of apps and sends a list of tasks to the local
Schedulers residing on the IaaS instances (see Section II-B). An IaaS instance usually hosts two or more containers that are created and coordinated by the local scheduler. When a list of tasks is received, the local scheduler triggers the local cache daemon to asynchronously download these apps from the data provider. In the meantime, the containers are created and the virtual devices are started.

**Container Layout:** In general, container virtualization makes use of operating system level virtualization: The host kernel is shared with several isolated process groups (in the remainder of the paper referred to as containers or guests) that appear as separate physical systems to guest processes. In contrast to full- and para-virtualization solutions, container virtualization is lightweight; it enables isolated environments for user processes without additional virtualization overhead. The file system within a container is composed of several layers during creation time: Each layer represents a separate folder on the host’s file system. During startup, these folders are overlaid transparently and form a single coherent file system using a storage backend.

During the development of Dynalize, we evaluated three different storage backends: Advanced Union File System (AUFS) [3], LVM2 DevMapper [15] and VFS [6]. AUFS is a union file system that provides a file-level Copy-On-Write (COW) mechanism, while DevMapper is the kernel part of the LVM2 logical volume system, implementing a block-level COW. VFS does not provide COW support, instead a container’s file system is formed by making a deep copy of its layers. The main advantages of composing a file system consisting of layers are deployment time and reusability of the base layer. Since AUFS gives the best results with respect to container startup times and throughput (see Section III-C), Dynalize makes use of AUFS.

A Dynalize container consists of the following layers: (i) The virtual device layer, (ii) the environment layer and (iii) the task layer. The term *task* refers to the analysis of a single app, while the term *job* refers to a list of apps to be analyzed. The virtual device layer contains the emulation program for the virtual device and its virtual storage, i.e., its system image and a virtual SD card. This layer may contain emulators such as the Android Emulator, but also software such as the open source QEMU extension PANDA [7]. Dynalize virtual devices rely on full system emulation, since they are executed on the application layer, and hardware-acceleration is usually not supported in an IaaS cloud. On the other hand, full system emulation supports the execution of both native ARM or x86 binaries, which is not the case with hardware-accelerated x86-emulators. Furthermore, Dynalize offers different virtual device types, reflecting different amounts of resources reserved for each virtual device. The environment layer consists of programming languages, libraries and other dependencies for the execution environment of a dynamic analysis. It contains dependencies that are consistent for a long period and not changed with every minor update of an analysis. Finally, the task layer contains the program that performs the dynamic analysis.

This layered design ensures that Dynalize can use the snapshot capability provided by several IaaS providers. Even if a task changes slightly, only the task layer has to be retransmitted to the IaaS instances. Since the virtual device layer and the environment layer can grow very large (several GBs for system images, SD cards etc.), this container layout avoids the retransmission of large files within the virtual device and the environment layer. In contrast, if a new version of an analysis is deployed in Dynalize, it is retransmitted to and cached within each IaaS instance.

**Local Cache Daemon:** In contrast to cluster solutions such as Andlantis [4], data transfer from and to the data

---

**Figure 1.** Dynalize platform architecture.

**Figure 2.** Layout of an IaaS instance used to analyze apps with Dynalize.
provider is completely separated from the containers. By design, a data provider can be any storage service that supports downloads via HTTP/HTTPS, such as Amazon S3\(^1\) or GitHub\(^2\). All containers share a common read-only directory residing on the host, in which prefetched apps are stored by the local cache daemon. Prefetching allows a significant reduction of startup time of an analysis, since data transmission can be started before the container is created. Also, the incoming data cache allows the reuse of a previously fetched app. On the other hand, the outgoing data cache is a writable directory residing on the host, which is created during the container startup process. Each container has its own outgoing data cache directory, and when a task within a container signals the end of an analysis, the data within the cache is asynchronously transferred to the data provider while the container is restarted for the next analysis.

B. Scheduler Node

The Scheduler Node provides a frontend to the user, enable her to deploy her analysis and to submit jobs (see Figure 3). The user can choose between (i) one-time and (ii) long-term jobs, enabling her to either (i) start and terminate IaaS instances on demand or (ii) keep IaaS instances running to make use of local caches. Furthermore, a job configuration specifies a data provider as well as the type and number of virtual devices used to process this job.

Device Scheduler: IaaS instances are started and terminated by the device scheduler that maps virtual devices onto IaaS instances. It is based on a multi-objective scheduling algorithm: Since IaaS instances are billed in a pay-as-you-go manner, the objective function of the scheduling algorithm includes job execution time and execution costs. The search space includes different IaaS providers, each allowing their users to choose from a broad variety of instance types with different prices per hour. Primarily, instance types specify resources like the number of CPUs and the amount of RAM for a virtual server. Secondly, instance types also reflect different kinds of hardware, which are optimized, for example, for I/O-intensive or CPU-intensive workloads. Therefore, the multi-objective scheduler models execution time as a function of performance parameters like average virtual device startup time and throughput. The device scheduler is implemented as a variant of the Strength Pareto Evolutionary Algorithm (SPEA2) \cite{28}, has good convergence properties and is adaptable to different application areas.

Global Scheduler: The global scheduler is responsible for distributing tasks to the IaaS instances. First, the tasks are partitioned among the virtual devices. The task list for each virtual device is sent to the corresponding local scheduler that initiates the app download and the creation of the containers (see Section II-A). After a task is finished, the local scheduler notifies the global scheduler, which in turn keeps track of the on-going and finished tasks. In case of mixed instance types, the global scheduler reschedules tasks between the instances. In case of one-time jobs, instances

\(^1\)http://aws.amazon.com/s3
\(^2\)https://github.com
can be terminated if no tasks can be rescheduled.

C. Service Node

As Figure 1 shows, both Scheduler Node and Service Node provide a frontend to the Dynalize user. While the Scheduler Node frontend is used to configure the IaaS provider interface, to submit jobs and to deploy analyses, the Service Node allows Dynalize users to make their work publicly available. For example, when an analysis has been successfully performed on a large number of apps, the developer can automatically generate both a job submission template for the Service Handler and a web page template for the Presenter. Both templates can be modified to ensure a cost-efficient execution or a custom web page layout.

Since the main work is done by the Scheduler Node, a service request is processed in a straightforward manner. The Presenter runs a web server, on which a service user can upload apps to analyze. After an app is uploaded, the Service Handler submits a job to the Scheduler Node, which is terminated after all apps are analyzed. When the Scheduler Node signals job completion, results are uploaded to the cloud storage and made available to the user on the web page of the Service Node. The service can be tested using our demo service³.

III. Evaluation

In the following sections, a use case and a performance evaluation of Dynalize are described.

A. Use Case

As a use case for Dynalize, we investigate which Android apps potentially use malware obfuscation techniques. Rastogi et al. [22] have shown that typical anti-malware solutions heavily depend on static signatures and therefore are ineffective against simple program transformations (transformation attacks). In particular, programs that make use of reflection (e.g., using the Java Reflection API) or code encryption can make parts of an app binary unavailable for static analysis. On the other hand, dynamic analysis can give hints on which apps use obfuscation techniques. For example, dynamic analysis can easily detect whether native libraries are linked within a binary or not. Afterwards, these apps can be further investigated with other tools.

For Dynalize, this is an interesting use case: (i) The execution of native binaries requires full system emulation, which Dynalize provides. (ii) The analysis needs to be performed on a representative set of applications, i.e., thousands of apps. In our use case, we have analyzed about 6,000 free Android apps from the Google Play store, with an average size of 17.7 MB per app and a total size of 10 GB, for signs of obfuscation. While a sequential dynamic analysis of these 6,000 apps took about 148 hours (nearly one week), Dynalize can offer a significant speedup due to parallelization. (iii) An appropriate de-obfuscation technique for arbitrary apps requires a variety of tasks, such as scanning the application for native library use, dynamic library linking, use of the Java Reflection API, system calls and anomalous network traffic. These can be implemented using different libraries within the execution environment of Dynalize. (iv) The use case can give evidence about the significance of dynamic mobile app analysis; to make appropriate assumptions about real-world mobile apps, a representative sample of apps needs to be analyzed.

The dynamic analysis in our use case has been implemented using Python 2.7, the Android Developer Tools, the Android Emulator, and the Android Debug Bridge. Most of the data has been collected with the Linux strace tool that traces system calls and signals. The Android Debug Bridge is used to install, start and trace an app on the virtual device. Each output is stored in a single file on the outgoing data cache on the IaaS instance.

Dynalize has been used with 40 virtual devices in parallel on 20 c4.large Amazon EC2 instances, two virtual devices each. The data was previously uploaded to Amazon S3. The additional startup time introduced by the instance boot time is negligible (less than a minute). However, due to resource sharing between virtual devices, the analysis of an individual app is approximately 10% slower than with a single virtual device. In summary, dynamic app analysis for the 6,000 apps was executed within 4.4 hours (3% of the time required for sequential analysis), with costs of 11.88 $ (90 instance hours). The results show that 906 apps (15.1%) use native libraries and 192 apps (3%) dynamically load other binaries. There is a high usage of the Java reflection API, since it is used by 4,238 apps (71%). Another 3,483 apps (58%) make use of the Java cryptography API.

B. Virtual Device Performance

IaaS providers allow users to choose between several instance types. Scheduling virtual devices on IaaS instances requires us to consider different IaaS instance types. During the sequential execution of our use case, we observed that the virtual device startup took 22-47% of the total execution time. Furthermore, the throughput between virtual device and container is important. Therefore, both virtual device startup time and throughput were evaluated for different IaaS instance types. They were measured with the Android Emulator with 512 MB RAM and the Android Debug Bridge, running on top of an Ubuntu 14.04 Linux distribution.

The virtual device startup time for different Amazon EC2 instance types in region eu-west-1 is shown in Figure 4(a), while the costs for the used instance types is listed in Table I. All virtual devices were started simultaneously, the values in brackets reflect the number of vCPUs. Note that t2.large only provides 1 vCPU and 2 GB RAM, while m3.medium provides 1 vCPU and 3.75 GB RAM. The t2 and m3 instance types are generic instances, while
c4 is optimized for data processing, r3 for RAM accesses and t2 for storage. Furthermore, t2 instances are burstable. According to Amazon\(^4\), CPU performance of t2 instances is based on CPU credits. Initially, a t2 instance earns credits for 30 minutes of 100% utilization, and every hour credits for another 12 minutes are added. If an instance runs out of credits, it is reduced to 20% (t2.small) and 40% (t2.medium) of CPU usage.

Since CPU performance is significant for full system emulation, Figure 4(a) shows the best results with 1+ vCPU per virtual device. Nevertheless, the values are not linearly scaled due to concurrent access on the device images (I/O). An interesting fact is that the best execution times were measured with both the low-end t2 and the high-end c4 instance types. This is due to t2’s ability to burst CPU performance for a short period of time. Therefore, this is a good choice for analyses of a small number of apps. On the other hand, c4 instance types use a high-frequency Intel Xeon processor with SSD-backed instance storage.


Table I

<table>
<thead>
<tr>
<th>Instance Type</th>
<th>vCPUs</th>
<th>RAM (GB)</th>
<th>$/h</th>
</tr>
</thead>
<tbody>
<tr>
<td>t2.small</td>
<td>1</td>
<td>2</td>
<td>0.028</td>
</tr>
<tr>
<td>t2.medium</td>
<td>2</td>
<td>4</td>
<td>0.056</td>
</tr>
<tr>
<td>m3.medium</td>
<td>1</td>
<td>3.75</td>
<td>0.077</td>
</tr>
<tr>
<td>m3.large</td>
<td>2</td>
<td>7.5</td>
<td>0.154</td>
</tr>
<tr>
<td>c4.large</td>
<td>2</td>
<td>3.75</td>
<td>0.132</td>
</tr>
<tr>
<td>r3.large</td>
<td>2</td>
<td>15</td>
<td>0.195</td>
</tr>
<tr>
<td>i2.xlarge</td>
<td>4</td>
<td>30.5</td>
<td>0.938</td>
</tr>
</tbody>
</table>

Therefore, both one and two virtual devices can be started within a short period of time.

Figure 4(b) depicts the throughput between the container and the virtual device for different Amazon EC2 instance types. It was measured by transferring a 133 MB app via the Android Debug Bridge from the analysis task to the emulated 2GB SD-card storage residing in the virtual device. The data is first passed through the ADB process running within the container and afterwards sent to a ADB server process within the virtual device via a QEMU-specific channel. Hence, the throughput of the Android Device Bridge makes heavy use of the CPU and involves several processes.

As Figure 4(b) shows, the best results were achieved with high-end c4 and l4 instance types. Due to the burst ability of t2, this low-end instance type also gives good results. The outlier for m3 indicates that the ADB throughput does not increase with a larger amount of RAM.

C. Container Storage Backends

As mentioned in Section II-A, the container file system consists of multiple layers. This can be realized with three storage backends: AUFS, LVM2 DevMapper and VFS.

Figure 5 shows different backend throughputs measured with the bonnie++\(^5\) file system benchmark suite. It tests sequential input and sequential output on the block level. Although the measurements with different Amazon EC2 instance types show a high variance, they indicate the following: (i) In general, read throughput per instance type is better than write throughput. (ii) AUFS is better for instance types with more CPU efficiency and more RAM, while DevMapper performs better for I/O optimized instance types. (iii) The best results are performed with VFS, which comes with nearly zero overhead.

Afterwards, the startup time for each container was evaluated. In this benchmark, a Linux ARM system image was booted with QEMU, measuring the interval between the container was created and the guest kernel was booted. Figure 6 shows that the AUFS storage backend performed best with an average startup time of 8.09 seconds. The DevMapper backend needed 10.55 seconds on the average, whereas VFS performed worst with an average of 131.68 seconds. The long startup time is due to the deep copy mechanism used.

by VFS: As Figure 7(a) depicts, Devmapper needed only slightly more (3.5%) disk space compared to AUFS, but VFS needed 1934% additional storage (32.12 GB in total).

The latter is also confirmed by Figure 7(b) that shows the average disk throughput of both sequential input and output. Since VFS cannot be used for its slow deployment times, an interesting result is that AUFS is slightly better than DevMapper.

D. Discussion

The results of the experiments can be summarized as follows: (i) In contrast to a sequential mobile app analysis, Dynalize offers parallel dynamic mobile app analysis to process mobile apps on a large scale in short time. Furthermore, Dynalize can be used as an economical and flexible alternative to a dedicated cluster. (ii) Our measurements with different Amazon EC2 instance types indicate that the ability of t2 instance types to burst CPU performance for a short period of time is useful for analyses of a small number of apps, while high-end instance types like c4 can perform analysis efficiently with a high number of virtual devices in parallel. (iii) While throughput and virtual device startup measurements show high variance for different instance types, the AUFS storage backend proved to be the best of the three alternatives AUFS, DevMapper, and VFS.
IV. RELATED WORK

Dynamic analysis has been used to analyze mobile apps with respect to privacy [8], security [22], [27], performance [24], energy consumption [10], [19], [20] and bugs [1], [12], [14], [16], [23]. Several frameworks for large-scale dynamic analysis for mobile apps [4], [11] and automatic test frameworks running in the cloud [17], [25] have emerged. Furthermore, on-demand physical and virtual devices for testing individual apps with different operating system releases and configurations [5], [9], [18] and clouds of devices for automatic tests [2], [13], [21] have been developed.

The problem of performing dynamic analysis of mobile apps has been solved differently in the past. Approaches such as AppDoctor [12] and AndroidRipper [1] analyze mobile apps sequentially and accelerate virtual device startups with snapshots. While this is practicable for a small number of apps, without parallelization, dynamic analysis on a large-scale can take months or even years. For example, a measurement study of the Google Play store by Viennot et al. [26] has covered roughly 1 million apps. Assuming an execution time of one minute per app, a sequential analysis of these apps would take about two years.

Another approach is to perform dynamic analysis in parallel on a cluster of physical [11] or virtual devices [4]. For example, Andlantis [4] can analyze 3,000 Android apps per hour for malware on a cluster of 200 servers. Nevertheless, it relies on dedicated resources and only supports malware analysis. In contrast, Dynalize provides a generic solution with dynamically provisioned resources, enabling the user to analyze mobile apps with respect to other targets than malware, such as performance, bugs or energy consumption.

Furthermore, cloud-based approaches have been proposed for large-scale dynamic analysis. For example, Mahmood et al. [17] have deployed the Android emulator on IaaS instances to perform a large number of analyses in parallel. Furthermore, Ravindranath et al. [23] have developed VanarSena, allowing app developers to upload a Windows Phone binary and obtain a bug report within a short amount of time. In contrast, Dynalize supports multiple platforms and custom emulators such as the open source QEMU extension PANDA [7] that adds the ability to record and replay executions. More importantly, Dynalize provides caching and scheduling methods to enable the user to consider costs/performance trade-offs and to decide between one-time or long-term analyses. Furthermore, in contrast to previous cloud-based solutions, a layered file system has fast deployment times even if the program that performs the dynamic analysis changes or makes use of other libraries.

More recently, container virtualization has attracted the interest of researchers (e.g., REMnux6). Containers can be deployed faster, compared to IaaS instances, and can be arbitrarily assembled from multiple storage layers. Container virtualization can also be beneficial for dynamic analysis: (i) Different virtual device emulators, programming languages and libraries can be easily composed to an execution environment for an analysis. (ii) Virtual devices can be recreated fast in order to provide a clean execution environment for an analyzed app. Therefore, Dynalize adapts container virtualization for its platform solution.

Several cloud providers already offer container virtualization services based on the docker container engine [6], such as the Amazon EC2 Container Service (ECS)7 and the Google Container Engine8. A user can acquire a cluster of containers that can be composed arbitrarily out of existing libraries, execution environments and – of course – virtual devices. For example, a developer performing dynamic app analysis can start a cluster of IaaS instances and make use of a container cluster manager such as Kubernetes9 to parallelize her dynamic analysis. But from a technical perspective, this approach is limited. Since these approaches leave the virtual server hosting the virtual container unmodified, the local storage cannot be used as additional cache, which is not efficient if large numbers of applications should be analyzed. For example, the study of Viennot et al. [26] is based on a data volume of 5.3 TB. Even smaller data sets with a volume of a few GBs require that the data is distributed and processed in an efficient and coordinated manner. Therefore, in contrast to pure container services, Dynalize provides a novel storage solution: It employs app prefetching and caching on the IaaS instance executing the Dynalize container, using a local cache daemon.

V. CONCLUSION

In this paper, Dynalize has been presented, a Platform-as-a-Service (PaaS) cloud for the dynamic analysis of mobile applications. It relies on container virtualization on top of IaaS instances, enabling dynamic provisioning and fast deployment of dynamic analyses. A platform architecture as well as a custom container layout and a novel storage solution on the virtual server layer have been presented. A dynamic security analysis of about 6,000 Android applications has shown the cost- and runtime-efficiency of a large-scale analysis for thousands of apps. Experiments focusing on container startup, virtual device to container throughput and different storage backends showed the feasibility of our approach.

In future work, we will perform a comparison between Dynalize and container virtualization engines like Kubernetes. We will also address other storage backends, like btrfs10 and overlayfs11. Furthermore, efforts will be

---

6https://remnux.org/docs/containers/malware-analysis/
7https://aws.amazon.com/ecs
8https://cloud.google.com/container-engine
9https://github.com/GoogleCloudPlatform/kubernetes
10https://btrfs.wiki.kernel.org/index.php/Main_Page
made to improve the cost- and runtime-efficiency of Dynalize. For example, QEMU checkpoint/restore mechanisms will be evaluated as an alternative to full virtual device termination/restarts. Finally, we will evaluate how Dynalize can be automatically customized to support new device emulators and the corresponding environment libraries.

ACKNOWLEDGMENTS

The work presented in this paper is supported by the LOEWE initiative (Hessen, Germany) within the NICER project and by Amazon Web Services (AWS) via an Education Grant Award.

REFERENCES