UNIVERSITY OF CALIFORNIA SAN DIEGO

Building and Breaking Burst-Parallel Systems

A thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Elizabeth Izhikevich

2018

The thesis of Elizabeth Izhikevich is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California San Diego

2018

DEDICATION

I cannot imagine a person who would *truly* want this thesis dedicated to them...

But, if you are that person, then this is for you.

EPIGRAPH

If you do nothing unexpected, nothing unexpected happens.

And who wants a life like that?

—Sue Black

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thank you to my family, advisors, and friends.

Thank you to my family, Tatyana, Katherine, and Eugene, for showing me that, no matter how crazy the idea, it is always worth trying to see what happens. This mentality is, in part, why my masters degree is being finished early at UCSD.

Thank you to my advisor, Geoff Voelker, for spending the time to teach me, support me and humor me. Your wisdom, calm, wit, and ability to communicate in few words, if any, makes you the ideal mentor and friend.

Thank you to my co-advisor, George Porter, for guiding me in all-things-AWS related. Sprocket turned out to be a perfect project to convince someone to pursue a Ph.D. in Computer Systems and the combination of both you and Geoff makes the ideal co-advising team.

Thank you to Lixiang Ao for being my “Ph.D. student mentor” during our early days working on Sprocket. Your dedication to answering all my questions and being relatively calm throughout our months of coding make you an ideal mentor and partner.

Thank you to my friends who have supported me throughout my days at UCSD and thank you especially to those who spent the time to entertain random (research related) thoughts with me.

Thank you to Office 3142 for collectively mentoring and entertaining me.

Thank you to my committee for your time and input with regards to this thesis.

Chapter 2, in part, is a reprint of the material that will appear in the Proceedings of the ninth ACM Symposium on Cloud Computing, 2018. Ao, Lixiang; Izhikevich, Liz; Voelker, Geoffrey M.; Porter, George. The initial Sprocket framework discussed in Section 2.2 is implemented by Lixiang Ao, Geoffrey M. Voelker, and George Porter.

Chapter 3, in part, is currently being prepared for submission for publication of the material. Izhikevich, Liz; Feteih, Nadah. The thesis author is the primary investigator and author of this material.

ABSTRACT OF THE THESIS

Building and Breaking Burst-Parallel Systems

by

Elizabeth Izhikevich

Master of Science in Computer Science

University of California San Diego, 2018

Professor George M. Porter, Co-Chair
Professor Geoffrey M. Voelker, Co-Chair

This thesis explores, from both a systems and security perspective, a relatively new serverless cloud computing model that runs on stateless container invocations. This thesis demonstrates that the responsiveness, concurrency, and cost of the serverless cloud computing model, on the one hand, enables making design decisions that were not possible before, yet on the other hand, introduces security vulnerabilities that potentially increase the attack surface due to a high probability of achieving co-residence.

This thesis extends a serverless video processing framework for the cloud, Sprocket, that takes advantage of the container computing model in its design in order to provide runtime performance, resource efficiency, and a highly configurable developing environment. Sprocket demonstrates all three qualities when running both a facial recognition and streaming video pipeline. Sprocket can be configured to process a 30-minute video 1000-way parallel in under a minute for less than $1. This thesis also presents an increased attack surface in the container service that Sprocket runs on, AWS Lambda, and shows how to detect co-resident containers on physical machines across victim and adversary accounts. Co-residency is achieved in three unique Amazon data centers, with as many as 30 co-resident containers during one 30-second "attack." This thesis demonstrates that the scalability of the serverless cloud computing model, in particular, inherently increases the probability of co-residency and that existing mitigations for co-residency must be re-evaluated.

Chapter 1

Introduction

Cloud computing has given programmers the ability to scale their software and workloads in an arguably revolutionary way [2]. For instance, using the Amazon Web Service computing platform, EC2 [4], users are able to run tens of powerful servers (virtual machines) concurrently and scale based on the job at hand. Moreover, with the introduction of data-parallel frameworks such as Hadoop and Spark, virtual machines can easily be orchestrated in large clusters to process highly-intensive data processing jobs in an optimized and parallel way. Virtual machines indeed provide a plethora of computing power at the fingertips of the everyday programmer. This thesis, however, demonstrates that a relatively new serverless cloud computing model that runs on stateless container invocations can provide computing power with increased responsiveness, parallelism and cost efficiency for certain tasks.

1.1 A Containers Perspective

Though instantiating a virtual machine is at the immediate request of the user, this thesis presents that it can take up to a minute per machine. A one-minute latency is arguably negligible for running long-lived or non-time-sensitive computations, such as parsing a massive social-media dataset using Spark. For time-sensitive computations, such as pulling data from a database in response to an event, however, the one-minute latency is no longer insignificant. Amazon's container service, AWS Lambda [21], invokes containers consistently in less than a second, thus making it a better candidate for time-sensitive lightweight compute jobs [34]. Invoking one lambda container takes on average 500ms whereas invoking one EC2 VM takes 40 seconds, as will be shown in Section 2.7. Moreover, this thesis demonstrates that, to run an extremely parallel job (e.g., using thousands of threads to apply a filter on frames of a video) performance on a Spark cluster of 30 powerful VMs is worse and more expensive than running 1,000 lightweight containers (AWS EC2's and Lambda's default concurrency limits, respectively).

Container services such as AWS Lambda undeniably have their drawback: short timeouts, no root privileges, limited memory, just to name a few. However, this thesis shows that these restrictions have not stood in the way of developing in this new environment and moreover, are a good match for compelling system applications. It is for these reasons that this serverless cloud computing model has seen a growth in popularity and every major cloud provider (Amazon, Google, Microsoft, IBM) now offers their own version of the same serverless container model [17].

In this thesis we demonstrate a system application, Sprocket, that is designed to run on any serverless container service. Nonetheless, we use the AWS Lambda serverless cloud computing framework as a case study to design and evaluate Sprocket. Introduced in 2014, Amazon's Lambda service provides a model where developers can run code without provisioning or managing servers. Amazon implements Lambda through the use of containers: resource-isolated processes. Lambda restricts the size of a container to be anywhere in the user specified range of 128MB to 3GB. Lambda also restricts the runtime of code, within the container, to be less than 5 minutes. Lambda bills a customer only for the running time of a container on the granularity of 100ms. Developers can either manually invoke up to 1,000 concurrent "lambdas"/ (containers), or they can set up an event driven system where AWS Lambda containers are automatically triggered and invoked.

1.2 A Security Perspective

Cloud computing has also introduced the security concern of having multiple VM tenants using the same physical host [28]. Research examines the ease and likelihood of an adversary being able to place her VM on the same physical host as a victim and break the expected isolation. Co-residency of VMs allows an adversary to share physical resources with a victim, thereby leaking information about the victim to the adversary. For example, by measuring a victim's cache usage, an adversary can steal a victim's password [28]. This thesis analyzes the co-residency threat in container services, as opposed to VMs, and explores how the ephemeral and parallel nature of stateless container invocations contributes to achieving co-residency.

1.3 Thesis Contributions

This thesis explores a serverless cloud computing model that runs on stateless container invocations from two perspectives: designing a system and evaluating security. Concretely, Chapter 2 presents the design of a video processing framework that uses the intrinsic properties of video and the light-weight nature of containers to deliver its performance goals of high parallelism, low latency and low cost. Chapter 3 then focuses on the serverless container framework — including the container scheduler, the CPU scheduler, and shared hardware resources — underlying the video processing system to explore and achieve hardware co-residency among containers in the cloud.

1.3.1 Developing and Optimizing Sprocket: A Serverless Video Processing Framework for the Cloud

Frameworks that are optimized to efficiently process terabytes of numerical and text data have received significant attention, such as MapReduce [9] and Spark [30]. However, frameworks optimized to process video data in particular have largely been absent due to the complexity of video data (e.g., size, compression dependencies, etc.). In Chapter 2 I will first present an efficient video processing framework, Sprocket, that takes advantage of the low latency and high parallelism of container services to achieve its goals. I then describe an extended design of Sprocket which implements a complex application involving external cloud services, such as facial recognition, and discuss the challenges of using such external services. I then focus on the implementation of a "streaming scheduler" that takes advantage of a container's highly responsive invocation and tear-down performance. Lastly, I evaluate the performance of the discussed designs as well as compare the performance to a proposed alternative pipeline implemented in Spark. I show that Sprocket can be configured to process a 30-minute video 1000-way parallel in under a minute for less than $1.

1.3.2 Pushing Lambda to the Limit: Detecting and Evaluating Co-residency in the Serverless Cloud

Due to the lightweight nature of containers, serverless cloud container services are more restrictive in runtime environment, size, and permissions, yet consequently less restrictive in scalability, performance, and cost. In Chapter 3 I use the AWS Lambda container service as a case study to investigate what security vulnerabilities arise when customers are given the freedom to scale their lightweight computing resources in the thousands in a fraction of a second and at a cost of pennies. In particular I investigate vulnerabilities in the container scheduler, the CPU scheduler, and shared hardware resources across containers and host operating systems. I demonstrate vulnerabilities used to achieve co-residency of containers across adversarial and victim accounts in three different AWS data centers, as well as analyze the general threat that cloud providers face when allowing customers to scale their computing resources. Finally, I discuss existing mitigations in the VM and container space against adversarial co-resident behavior, and argue why they may no longer be appropriate and should be re-examined.

1.4 Thesis Scope

This thesis explores a serverless cloud computing model that runs on stateless container invocations. Specifically, we first explore the benefits of the model through the design of a burst-parallel system, Sprocket. We subsequently explore the security risks of the burst-parallel framework by analyzing the ease of achieving co-residency between adversarial and victim accounts within AWS Lambda. Finally, this thesis discusses the opportunity for future work at the intersection of the two projects. Specifically, we discuss going beyond the covert channels we present and being able to identify an uncooperative victim lambda container, with the overall goal of leaking the behavior of an entire burst-parallel system.

Chapter 2

Developing and Optimizing Sprocket: A Serverless Video Processing Framework for the Cloud

2.1 Introduction

Video is one of the most dominant types of data today: 70% of consumer internet traffic is compressed video content [6]. Moreover, new forms of machine learning are enabling the extraction of meaningful information from individual video frames, allowing algorithms to "peer inside" the video content and thus provide crucial insight (e.g., recognizing a criminal in a snapshot of security footage [15]). Video is also one of the most non-trivial formats of data, since video encoding takes advantage of both spatial and temporal similarity for compression. Video thus cannot easily be processed in the same way that numerical or text data can be, as there are costly steps that need to be taken to decode and later encode the video back into its original format. Known frameworks that are optimized to efficiently process terabytes of data, such as MapReduce [9] and Spark [30] are unfortunately also not optimized to process video data, as will

Table 2.1: Details of input videos used in the experiments.

Name	Length	FPS	Content	Resolution
Earth [10]	10 hours	25	Nature	1080p
Synthetic Earth	Variable	25	Uniform 1 second chunks from Earth	1080p
Nature [25]	60 sec	25	Trailer/Contains no faces	720p
Avengers Trailer [3]	35 sec	24	Trailer/Action/Contains faces	720p
Interview [7]	45 sec	24	Interview/Contains faces	720p

be shown in Section 2.7. Due to video's prevalence as a data type however, the need for a system with low latency, low cost, and high scalability for video processing is imperative.

2.2 System background

The following framework in Section 2.2 is implemented by co-authors Lixiang Ao, George Porter and Geoffrey Voelker. Table 2.1 lists the test videos and their properties used in the experiments throughout Chapter 2.

Sprocket is a serverless, scalable, cheap, programmable stage-based pipeline used for video processing. To achieve this, Sprocket takes advantage of the serverless cloud and runs on the AWS Lambda container infrastructure. Sprocket's framework enables containers to be immediately spawned up or torn down in an on-demand fashion, based on both the properties of the video being processed (e.g., video length, video resolution) and the job itself (e.g., simple filter, machine learning). Via a modified version of Mu [12], Sprocket maps different tasks (e.g., decode, filter, encode) in the video pipeline to particular lambda workers. In this fashion, Sprocket is able to start processing video 1000-way parallel within seconds.

2.2.1 A simple application: video filter

To better understand how the Sprocket system is designed, we first describe a relatively simple application.

In the early design stages of Sprocket, the system supported applying simple video filters,

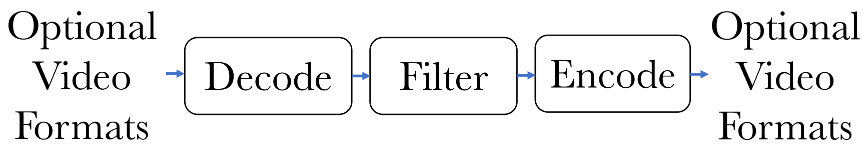

Figure 2.1: Logical overview of the Video Filter pipeline.

such as grayscale, to all frames of an input video in a highly parallel manner (up to 1,000 parallel “threads”). To achieve this, Sprocket first processes an input, which can be in various formats (video link, local mp4, cloud-local file hosted on S3) and sends segments of the input to multiple lambda workers in parallel. Each worker then decodes a chunk of video into a set of sequential frames. The lambda worker then invokes the `FFmpeg` binary to each frame. Finally, each transformed frame is encoded into an output format and stitched together to form the final output video.

This video filter pipeline (Figure 2.1) consists of the following three stages:

Decode: This stage decodes a specified chunk of input video into individual uncompressed frames (in PNG format). Decode receives video metadata as input, along with the timestamp of where in the video to begin decoding, and how many frames to output to the downstream worker. After processing, the decode stage emits the decoded frames to the S3 intermediate storage system.

Filter: The Filter stage applies the `FFmpeg` binary to a chunk of frames. Filter is spawned directly after the Decode stage and receives metadata along with references to the location of the frames stored on intermediate storage. Filter collects the frames from S3 and applies one of its internal filters as specified in the pipeline’s pipespec configuration file, described in Section 2.2.3.

Encode: This stage is responsible for encoding frames. Encode is also implemented with a separate copy of `FFmpeg`, running in a different lambda worker, which receives metadata along with references to the location of the frames in S3. Encode collects the frames from S3, encoding them using the specified encoder format, and finally writes them either in MPEG-DASH format or a single compressed output file generated by ExCamera [12], a highly parallel video encoder that stitches small chunks of video together in parallel. The final result is stored in S3.

2.2.2 Delivery functions

To capture the dependencies between different stages and send correct events from "upstream" stages (e.g. decode) to "downstream" stages (e.g. filter), Sprocket defines a delivery function for every stage which specifies the dependencies of its inputs.

An example of a dependency between Sprocket stages is within a pipeline that takes two input videos, Video A and Video B, and "blends" each sequential frame from Video A and Video B into one frame to create Video C. In this example, a delivery function would need to wait for the *i*th frame of both Video A and Video B to be ready before it can pair together both frames and send the pair to the downstream blend stage. If for some reason Video B's *i*th frame is not ready, the delivery function will hold that frame and not deliver Video A's *i*th frame alone to the downstream blend stage.

2.2.3 Programming Sprocket applications

To program a Sprocket application, a programmer uses a domain-specific pipeline specification language to create a pipeline specification (pipespec) file. In the file, a user specifies the set of stages that make up the pipeline (e.g., decode, filter), the edges between the stages (i.e., upstream vs. downstream stages), and the dependencies between the stages (i.e., through the specification of a delivery function). Figure A.1 illustrates an example of a pipespec file for a facial recognition application, further described in Section 2.3.

2.3 A complex application: facial recognition

To demonstrate Sprocket's ability to implement a complex pipeline, we describe a facial recognition pipeline (Figure 2.2). This application consists of a sophisticated set of operations,

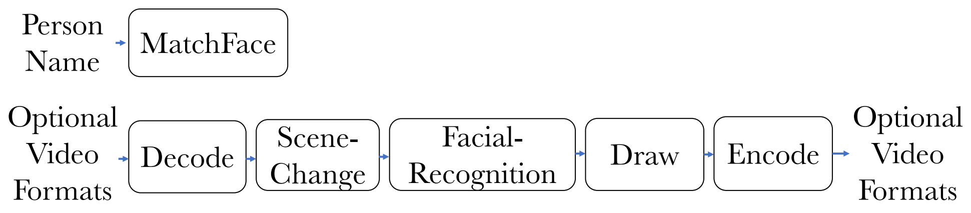

Figure 2.2: Logical overview of Facial Recognition pipeline stages.

including calling out to other cloud services to implement the facial recognition support. To evaluate the capabilities of Sprocket, we implement a pipeline that, given an actor's name and a reference video URL as input, draws a box around the given actor's face in all scenes of the video.

2.3.1 Stage design

The Facial Recognition pipeline consists of six stages: MatchFace, Decode, SceneChange, FacialRecognition, Draw, and Encode (Figure 2.2). At the beginning of the pipeline, an actor's name, provided by the user, will be used to locate candidate images from a Web search for later use in facial recognition. In parallel, the provided input video URL is fetched and decoded into fixed-length chunks of frames, currently one second each. Workers in parallel will then run a scene change algorithm on each chunk of frames to bin them into separate scenes. For each set of chunks grouped in scenes, a worker will then run a facial recognition algorithm to determine if the target face is present in that scene. If a face is identified in the scene, the chunk of frames will have a bounding box drawn on all the frames at the appropriate position returned by the vision algorithm API, which is then sent downstream to the Encode stage. If no face is detected, then the group of chunks will be sent directly to Encode.

Along with the stages described in Section 2.2, the following stages are used in the Facial Recognition pipeline:

MatchFace: The MatchFace stage searches for a target image for the face of a person whose name is specified as a parameter. Sprocket currently uses Amazon's Rekognition API [27], but could also use other service offerings: Microsoft offers a computer vision API as part of its Cognitive Services cloud offering [24], and Google offers a cloud-hosted vision system for labeling and understanding images through its Google Cloud Vision API [14].

MatchFace invokes one of these third-party image search services (in our case Amazon Rekognition) to find the top-*k* images returned given the provided name. The stage then iterates through the returned images and runs a face detection algorithm, via external API call, to make sure that the chosen target image contains a face. The first image to pass the facial detection algorithm becomes the selected target image. MatchFace then stores the selected image in S3 for eventual use by downstream stages. Unlike other stages, MatchFace itself does not emit any data directly to downstream stages.

SceneChange: The SceneChange stage detects scene changes in a set of decoded frames. It is invoked after the Decode stage, and is sent a reference to the decoded frames stored in S3. SceneChange collects the frames from S3 and, after detecting the scene change offsets (using an algorithm internal to `FFmpeg`), emits an event containing a list of these references to frames stored in S3, paired with a boolean value marking which frames serve as the boundaries of the scene change.

FacialRecognition: The FacialRecognition stage detects if a group of frames contains a target face (e.g., of the provided actor). The FacialRecognition stage is spawned once for every group of frames that make up one scene. We chose this design point, rather than running on every frame in the scene, due to the performance limitations of invoking third-party computer vision algorithms at that rate. FacialRecognition downloads the frames from S3 and calls the facial recognition algorithm once on every *n* frames in the scene. The facial recognition algorithm returns whether or not the target image was detected in the frame, and a bounding box of the identified face in the original frame. If at least one frame in the scene is found to contain the target face, all frames in the scene are marked as having the target face. The stage then emits an event containing a list of references to the frame in S3 paired with a bounding box of the identified face. If no target face is identified, FacialRecognition emits a list of frame references paired with empty bounding boxes.

Draw: The Draw stage draws a box at an arbitrary position in the frame, in this case provided as a bounding box around a recognized face. Draw is instantiated from the FacialRecognition stage and only continues if a scene was labeled as containing a recognized face. Otherwise, Draw automatically skips to its final state and sends references to the frames in S3 directly to the downstream stage. Draw only uses the dimensions from one bounded box to draw the same bounding box on all frames. Therefore, Draw assumes that there is little movement of faces in a single scene. We leave as future work interpolating the position of the box based on sampled points throughout the scene. Draw writes the new frames to S3 and emits an event containing a list of frame references.

An alternative version of a Facial Recognition pipeline can also choose to emit scenes that only contain the recognized face. In this case, the Draw stage would be replaced by a SceneKeep stage that only emits references to frames if a face is recognized. Otherwise, the stage will emit an empty list and those particular frames will never be encoded. SceneKeep would be employed to implement a version of a pipeline that edits out all scenes of a theatrical production that do not include a given actor.

Note that there are a variable amount of lambda workers dedicated per stage, largely depending on the properties of the input video. For example, the amount of Decode and SceneChange workers can be arbitrarily determined by the amount of seconds in the video (e.g., one worker per second). However, the amount of scene changes a video contains will determine how many downstream FaceRecognition workers get spawned. Furthermore, the amount of recognized faces in a scene will determine how many Draw workers get spawned.

2.3.2 Calling external cloud services

Calling an external API to run a facial recognition algorithm creates different challenges for Sprocket. The first is that there is always extra latency for calling the external service that is beyond Sprocket's control. For the AWS Facial Recognition service, there is an average latency of 1.43 seconds (SD = 0.22 seconds) per "detect face" call, and 2.28 seconds (SD = 0.08) per "compare face" call, when we attempt to recognize Thor in 14 frames, with and without the presence of Thor, in the Avengers Trailer. We run the same experiment using Microsoft's facial recognition service; the"detect face" and "compare face" operations have an average latency of 0.87 seconds (SD = 0.16) and 0.26 seconds (SD = 0.05), respectively.

Sprocket may also encounter an API call throughput limit that slows down the execution time of stages. This execution latency increases as the parallelism of the pipeline increases, since more concurrent calls create a faster overload of the external API. To address this, Sprocket does two things. First, the facial recognition pipeline includes the scene detection stage, reducing the number of calls to the facial recognition API within one scene. Second, the pipeline has the option to use the streaming scheduler, described in more detail in Section 2.5, which adaptively calculates whether new lambdas need to be invoked to meet a streaming deadline throughout execution. In this way, Sprocket limits the amount of concurrent API calls needed, thus avoiding API call throughput limits.

2.4 Customizing delivery functions

Sprocket defines a delivery function for every stage which specifies the dependencies on its inputs. Moreover, delivery functions can also be used to mitigate challenges and optimize complex pipelines. For example, the Facial Recognition pipeline can incur new scheduling challenges based on the properties of the input data. Stragglers might arise, not just based on performance variation within the cloud platform (e.g. a slow lambda worker), but rather as a data-dependent result of whether or not a given frame contains a face, as has been reported in other contexts [20]. Scenes with recognized faces must eventually go through the Draw stage, thereby unavoidably taking a longer time to complete. Customizing different delivery functions

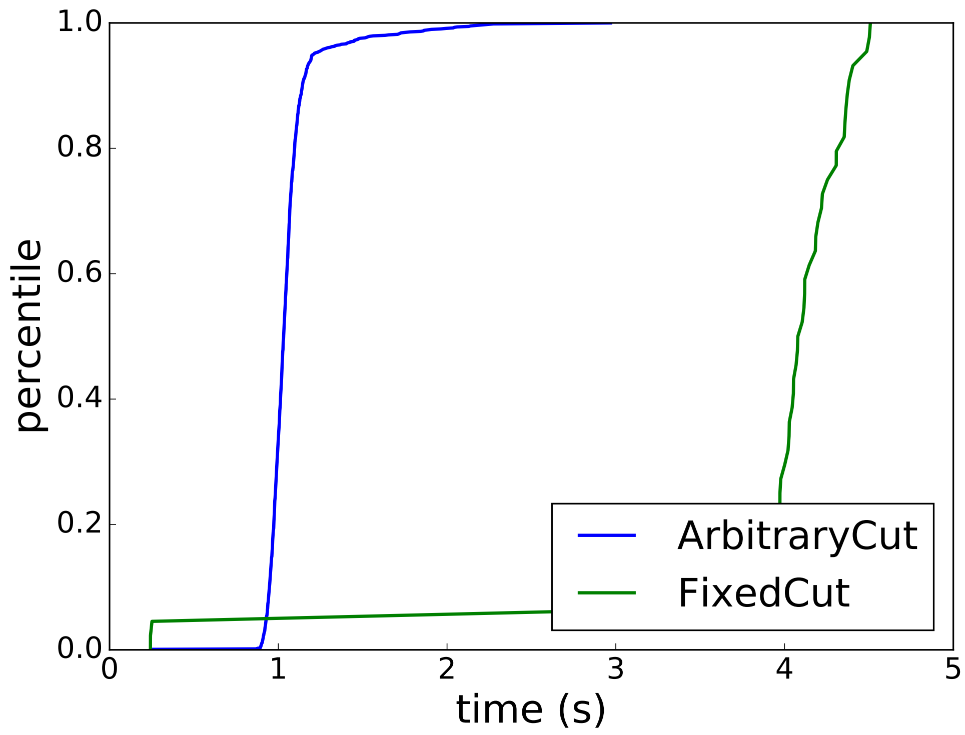

Figure 2.3: The effects of the serialized arbitrary segment delivery function.

can help address the recognized face straggler problem.

We create a serialized arbitrary segment delivery function to address face stragglers. The serialized arbitrary segment delivery function splits up the the delivery of downstream events based on cut-off markers provided by the upstream stage. With this design, the FacialRecognition stage is able to request the next downstream stage, draw, to receive only one frame per worker. By dedicating one lambda worker per frame, the Draw stage completes almost instantly and takes minimal overhead compared to the rest of the pipeline.

Figure 2.4 shows a simplified pseudocode for the serialized arbitrary segment delivery function.

To evaluate the performance gain due to the serialized arbitrary segment delivery function, we run Sprocket on the Interview video, from Table 2.1, where approximately 90% of the frames go through the Draw stage, with and without the use of the serialized arbitrary segment delivery function.

```
def serialized_arbitrary_segment_deliver_func(input_events,deliver_events):
    sort input_events by lineage number
    curr,start = 0
    while input_events exist:
        if input_events[curr]['lineage'] != expected_lineage:
            return
        else if input_events[curr]['cut_here']:
            merged_events = input_events[start:curr]
            add merged_events to deliver_events
            remove merged_events from input_events
            start = curr
        else:
            curr+=1
```

Figure 2.4: Serialized arbitrary segment delivery function pseudocode.

In Figure 2.3 we compare the difference in Draw stage runtimes, across 1,080 frames total, when an upstream delivery function splits the delivery of downstream frames based on runtime markers (arbitrary cut) as opposed to a pre-specified constant number of downstream frames (fixed cut). Without the use of the serialized arbitrary segment delivery function, the Draw stage takes on average three seconds longer to complete.

We also compare the overall runtimes of the pipeline with and without the optimized delivery function and report that, for the Interview video, the use of the optimized delivery function eliminates approximately 1.5 seconds of overall runtime. Running the same experiment with the Avengers video, however, in which only about 7% of the frames go through the Draw stage, the optimization adds no significant performance gain nor overhead.

2.5 Optimizing for streaming

Though Sprocket can be used as a batch system focusing on completion time for the entire video, we design Sprocket to operate equally well as a streaming system. Sprocket's ability to seamlessly allocate and deallocate resources also allows it to change its scheduling behavior in real-time, based on the pattern of the current job. Concretely, Sprocket's streaming scheduler continuously keeps track of the amount of time it is taking to process the current frames to

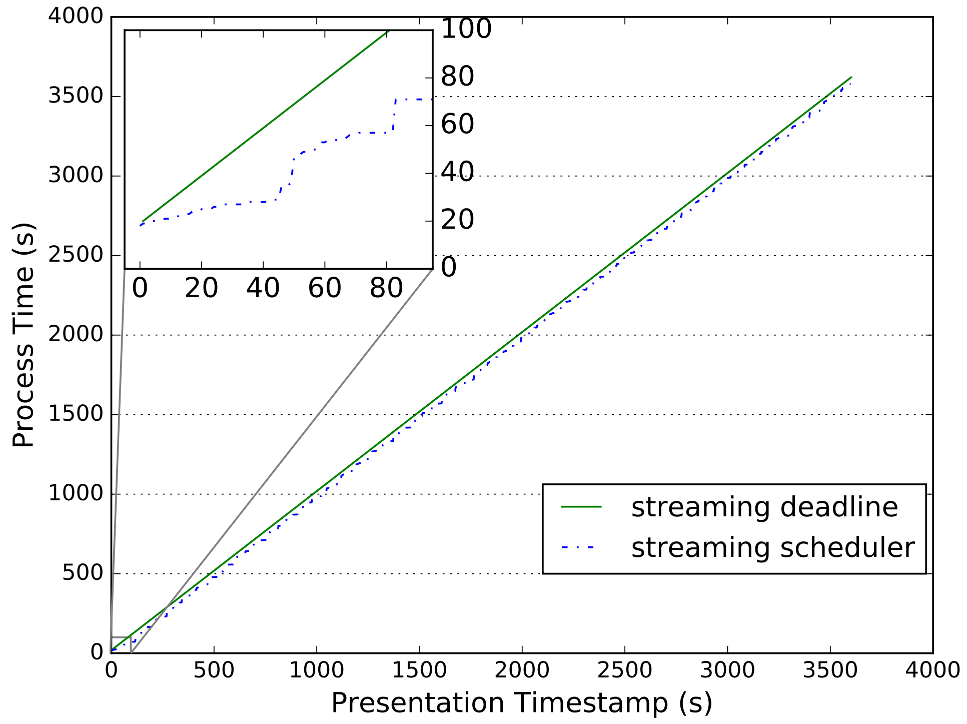

Figure 2.5: Streaming the Earth video through the grayscale pipeline.

determine if it will meet the streaming deadline. If Sprocket is currently ahead of schedule, the streaming scheduler will speculatively put the current pipeline to sleep for the number of seconds nearly equivalent to the difference between the streaming deadline and the current execution time. Putting the executing pipeline to sleep not only optimizes for use of minimum lambda resources, as the number of new lambda invocations drops down to zero for the current pipeline during that time, but also lends itself well to achieving load balancing so a simultaneous different pipeline can be run on Sprocket.

In streaming mode, once Sprocket finishes processing the first result frame, users can start to stream the video. Recall from Section 2.2 that workers encode video chunks into standalone DASH segments. As a result, even if a video is hours in length it does not need to be serialized into a single final video file before viewing. Sprocket is also able to handle streaming input, thus being able to process input video that is most recent. Hence, streaming is seamless as long as Sprocket processes and delivers subsequent video chunks in time.

To demonstrate this behavior, we execute the video filter pipeline on the first hour of the Earth video. Figure 2.5 shows the completion time of each worker and its chunk of video for the variable amount of workers determined by the streaming scheduler. The dashed line corresponds to the deadline that Sprocket needs to meet for seamless streaming. The line starts at the completion time of the first chunk since users have to wait for Sprocket to process it. But once started, Sprocket can easily meet the deadline for the remainder of the video, using minimal resources.

2.6 Evaluating complex pipeline behavior

Sprocket's behavior is highly dependent on the properties of the input video. Straightforward filters or transformations of video chunks perform the same work on each frame. The behavior of a more complex pipeline that recognizes and draws a box around a given actor's face, however, greatly depends on whether the input video contains a face, and whether that face is the one being queried. To demonstrate, we run the FacialRecognition pipeline on the Nature and Avenger videos (we use these short videos due to rate limits of Amazon's Rekognition API, as discussed in Section 2.3.2).

Figure 2.6 illustrates the bimodal execution times of the pipeline's lambda workers in the FacialRecognition stage, depending upon the presence of a face in a given frame. Lambda workers processing the Nature video, which has no faces, take between 1.5 and 4 seconds to complete the FacialRecognition stage. This execution time consists of the time it takes to invoke Amazon Rekognition to detect any faces present. Since no input frames contain faces, the stage passes along the frames immediately. For the Interview video, all the frames have faces, so the lambda workers will experience increased execution times. When Rekognition does detect a face in a frame, it makes a second API call to compare the detected face with the target face. This second call adds another 1.5 to 3.5 seconds of execution time, resulting in the bimodal execution

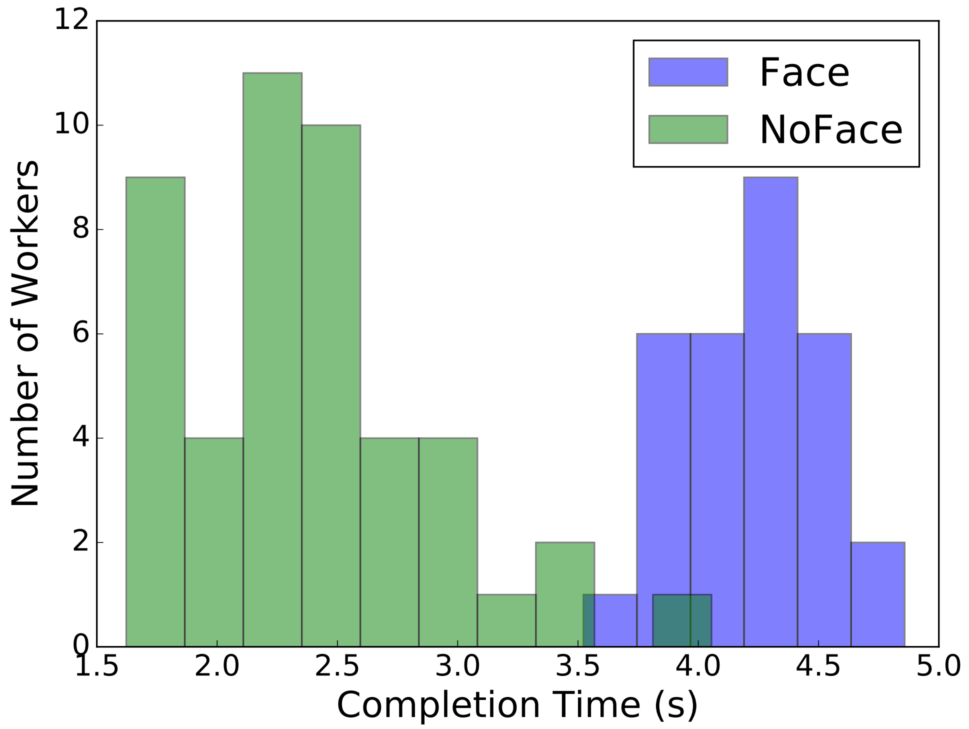

Figure 2.6: FacialRecognition stage behavior with and without faces.

times. In general, the total execution time for the 60-second Nature video with no faces was 32 seconds, and the execution time for the 45-second Interview video with faces was 45 seconds.

Figure 2.7 illustrates similar behavior in the downstream Draw stage for the same experiment as above. If a given face was not recognized, the Draw stage will immediately call the next downstream stage, adding less than a second of execution time. However, if a frame was marked as having a recognized face in the upstream facial recognition stage, then an extra half second will be spent drawing a bounding box around the actors face in the frame. Note that the experiment uses the serialized arbitrary segment delivery function as an optimization for the Draw stage.

2.7 Evaluating Sprocket against alternatives

Cloud providers do provide elastic offerings of more established parallel data processing systems such as Hadoop and Spark, but they are not a good match for Sprocket's goals. In terms of responsiveness, allocating and provisioning clusters potentially takes minutes before the new

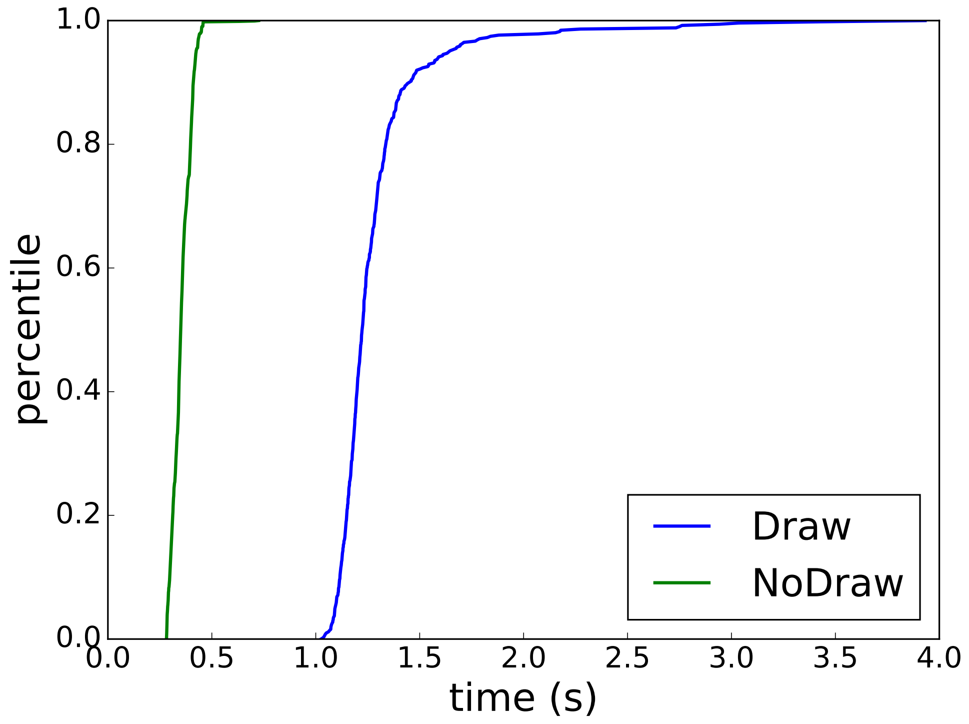

Figure 2.7: Draw stage behavior with and without positive face recognition.

cluster can begin accepting jobs. Processing video with even simple transformations often does not involve any reduction in the data (input and output sizes are similar); efficiencies afforded by reductions in MapReduce-style computations do not apply to a wide range of video processing tasks.

To make this argument more concrete, we perform a couple of experiments to illustrate the performance of a simple video processing application on Amazon's EMR Spark, an EC2 instance, and Sprocket. We use the Earth video as input, segmented into two-second video chunks, and performed a simple grayscale operation using the `FFmpeg` tool in all frameworks. The Spark implementation used an 18-node cluster, with each node processing a partitioned set of video chunks into resulting mp4 output files. The EC2 implementation executed a batch script running 64 `FFmpeg` processes in parallel on an m4.16xlarge instance, which has 64 virtual cores and 256 GiB of memory. Sprocket used a simple 3-stage filter pipeline (Decode, Grayscale, and Encode) executing a variable number of Amazon lambdas, one per chunk, using up to 1,000 instances.

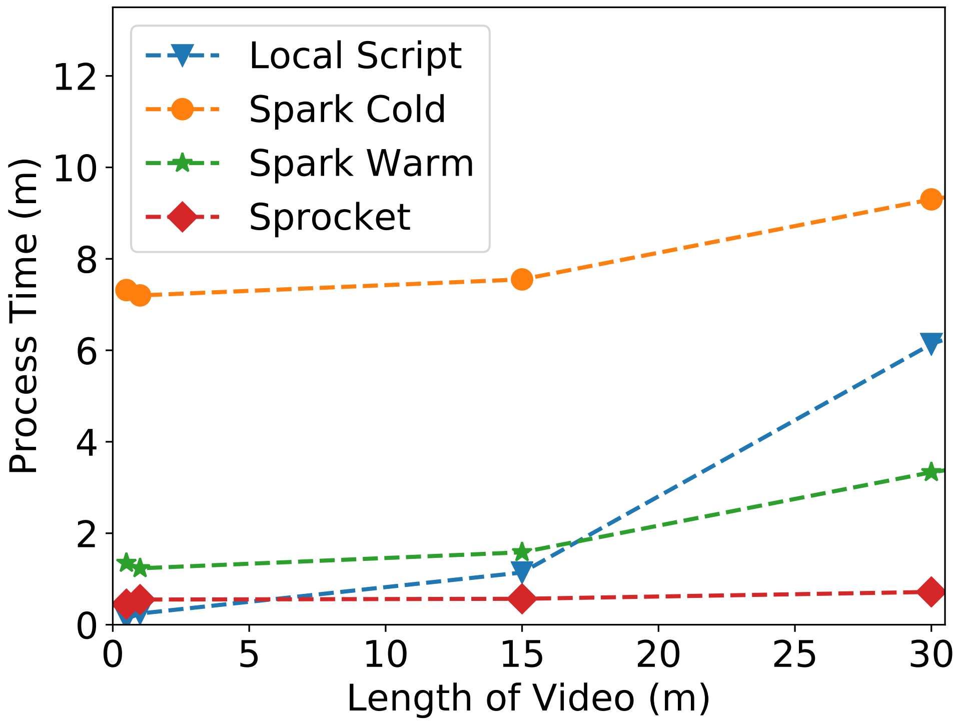

Figure 2.8: Comparing the runtime of 3-stage pipeline implementations.

Intermediate data was stored locally and the final output written to S3.

Figure 2.8 shows the execution time of the application on each platform as a function of video length. For Spark we show two lines, one including the time to provision and bootstrap the resources on AWS and the other with just the application time after the cluster is ready. Our goal in presenting this comparison is not to present this experiment as a "bakeoff" among the most optimized versions possible, but to illustrate the benefits of using serverless infrastructure for a single job. In particular, the startup time of provisioning cluster resources is significant in existing commercial offerings, which Sprocket avoids using the on-demand nature of lambdas.

We further run the same experiment on a more complex 5-stage pipeline (Decode, Grayscale, Rotate, Scale, and Encode) and present the execution times in Figure 2.9. Sprocket and Spark exhibit a relatively similar behavior as in the previous experiment, again showing that Sprocket exhibits less of an overhead when computing.

Figure 2.10 shows the price of our different implementations for the 3-stage filter pipeline

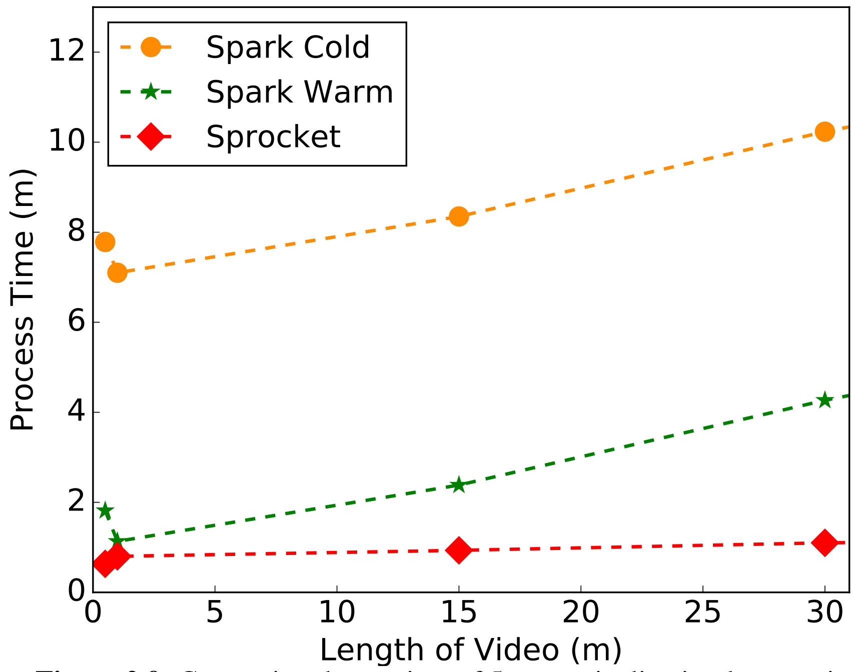

Figure 2.9: Comparing the runtime of 5-stage pipeline implementations.

experiment. Note that the Spark prices do not include the overhead of provisioning and bootstrapping a cluster on AWS (approx. 5 minutes). Across all implementations Sprocket is currently the cheapest, largely because Sprocket's execution time is the shortest.

Figure 2.11 illustrates the time it takes for the first encoded mp4 chunk to appear in AWS S3. This measurement is important for streaming applications in which the earlier the first chunks appear, the earlier the result can be streamed back to the client. Thus, the rest of the chunks can finish being computed upon as the beginning of the video is already streaming. Sprocket's time to first byte time is less than 1/4th of the best performing Spark cluster, not including provisioning and bootstrapping overhead, with regards to overall runtime. Clearly there is a scheduling overhead that Spark consistently encounters which inhibits it to efficiently finish an arbitrary first job.

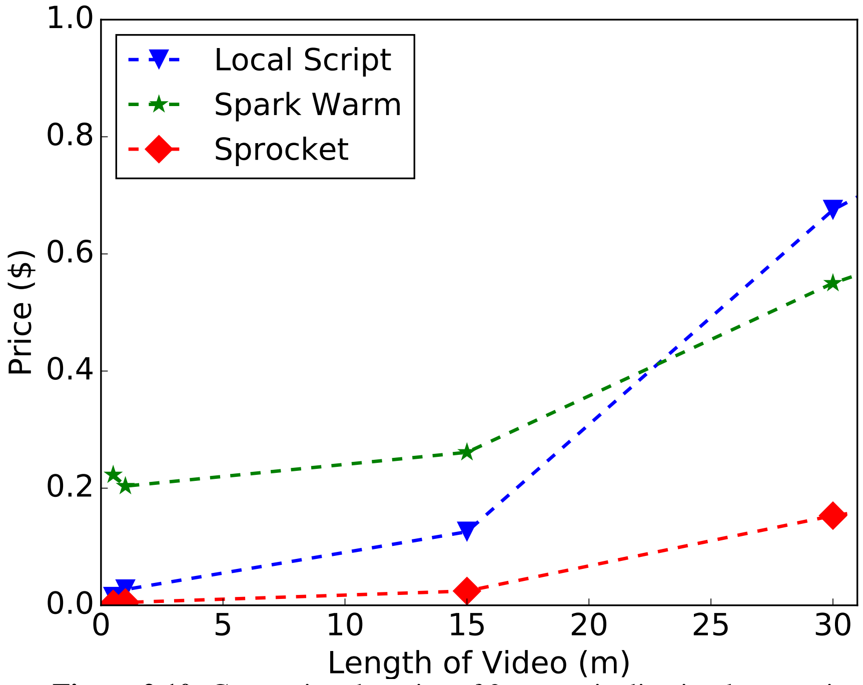

Figure 2.10: Comparing the price of 3-stage pipeline implementations.

2.8 Related Work

AWS recently released a framework that allows one to build applications that run on scalable lambdas [31]. To build a scalable lambda application an AWS Step Function is used to create a state machine that assigns lambdas to different stages of the state machine. AWS Step thus allows for a stage-based system to be built that can also run thousands of lambdas concurrently.

Sprocket differs from AWS Step in primarily two ways. First, AWS Step cannot implement a pipeline. A parallel state in AWS Step that executes many lambdas at once will wait for all lambdas to terminate before processing the next state. A work-around that the developer community has found has been to make multiple copies of each state machine so that they can run independently. However, such a solution does not lead to an obvious approach to make AWS Step dynamically re-size the amount of lambdas devoted to parallel intermediate stages, as the amount of parallel stages must be determined from the beginning of execution. AWS Step can

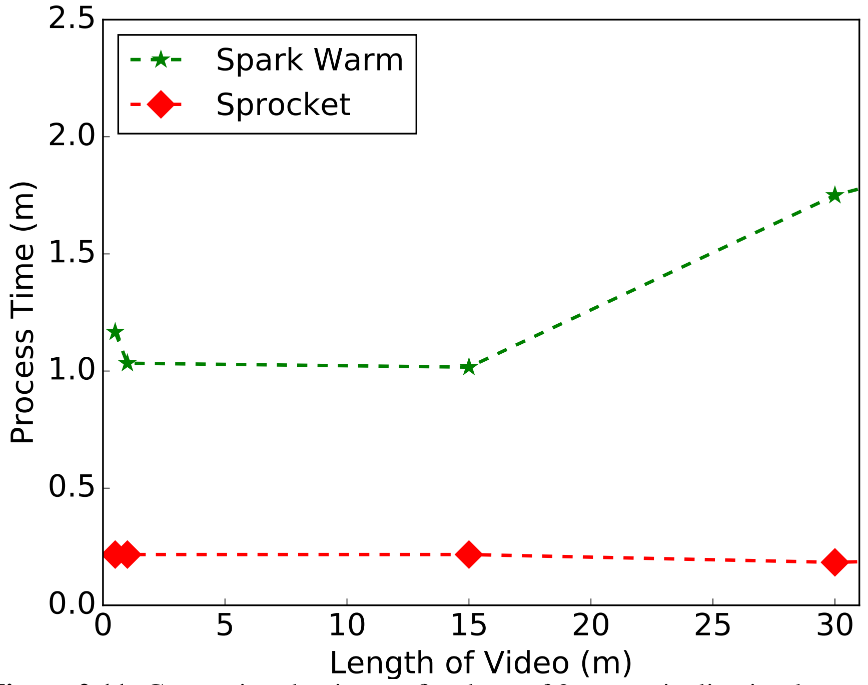

Figure 2.11: Comparing the time to first byte of 3-stage pipeline implementations.

scale the amount of lambdas based on an overall increased workload but it is not evident how to scale based on the dynamic request of a previous stage. Consequently, an intermediate Sprocket stage such as FacialRecognition, for example, would not be able to be implemented using AWS Step as it dynamically decides how many downstream Draw stages to spawn based on the amount of faces it has recognized in the previous stage.

2.9 Summary

Sprocket achieves easy configurability and low latency by taking advantage of the scalable and responsive serverless container framework that Amazon Lambda provides. We demonstrate Sprocket's configurability by first introducing a simple video processing pipeline that applies a filter to individual frames. We then extend the frameworks of that pipeline to allow for a more complicated facial recognition task. We also demonstrate Sprocket's scheduling configurability by describing a streaming scheduler which adapts the amount of container resources being used to achieve a streaming deadline. Finally, to demonstrate Sprocket's low latency, we benchmark Sprocket against a similar video processing implementation in Spark and show how Sprocket is faster, more scalable, and cheaper.

2.10 Open Problems and Directions

Though Sprocket is an already sophisticated system, there are ways in which Sprocket can be further developed on its own, as well as influence the development of more general processing pipelines.

Sprocket's streaming scheduler currently assumes a fairly symmetric workload across stages and therefore assigns a constant amount of lambda workers whenever the pipeline is not sleeping. Future work will involve assigning a variable amount of lambda workers throughout all of the data processing stages. Sprocket will need to behave speculatively for this to be efficient and will need to take advantage of straggler mitigation not from the perspective of recovering from abnormal *expected* runtimes due to platform behavior (e.g., the lambda worker is being slower than expected), but rather from abnormal *unexpected* runtimes due to input data behavior (e.g., the data has an unpredictable number of faces and thus requires more facial recognition calls).

As a general system, Sprocket has a unique "burst-parallel" behavior. Within a second granularity, Sprocket is able to invoke up to 1,000 containers to process a stage. It would therefore be interesting to investigate the concept of burst-parallel systems in general and what type of tasks can be efficiently completed in a highly parallel yet extremely short period of time (e.g., 1,000 containers in one second).

2.11 Acknowledgments

Chapter 2, in part, is a reprint of the material that will appear in the Proceedings of the ninth ACM Symposium on Cloud Computing, 2018. Ao, Lixiang; Izhikevich, Liz; Voelker, Geoffrey M.; Porter, George. The initial Sprocket framework discussed in Section 2.2 is implemented by Lixiang Ao, Geoffrey M. Voelker, and George Porter.

Chapter 3

Pushing Lambda to the Limit: Detecting and Evaluating Co-residency in the Serverless Cloud

3.1 Introduction

The utilization of third-party infrastructures for compute power has become increasingly popular because of its flexibility and ease of use. With the rise of providing customers the ability to rent out virtual machines on public clouds (Amazon EC2, Microsoft Azure, Google Compute Engine), customers are able to rent out tens of powerful VMs at once, without the need to worry about infrastructure set up or scheduling (the current default concurrency limit for EC2 is 30 VMs). The security of having multiple VM tenants using the same physical host in the public cloud is also of great interest. Research continues to be done to examine the ease and likelihood, within the constraints of default account limits, of an adversary being able to place her VM on the same physical host as a victim and break the expected isolation. Being co-resident allows an adversary to share physical resources with a victim, thereby leaking information about the victim to the adversary. For example, by measuring a victim's cache usage, an adversary can steal a victim's password [28].

Moreover, within the past couple of years, the same public cloud services have also begun to offer containers on demand (AWS Lambda, Microsoft Azure Containers, Google Cloud Functions). Customers now no longer need to worry about directly provisioning compute resources in advance and can use an event driven environment to scale these containers in the thousands while being billed less than a penny at a millisecond granularity. However, the new container services are also more restrictive to users with regards to image size (less than 3GB), runtime environment (pre-set libraries), and execution time (less than 5 minutes), in order to be able to provide the fast and inexpensive service. Nonetheless, cloud container services are especially useful for event-driven business models such as Netflix, where containers can be scaled in response to variable traffic [26]. It is therefore no surprise that over 1,000 companies are using Amazon's Lambda service [8].

With these new freedoms and restrictions that public cloud container services offer, it highlights the need to re-evaluate the same original fears that public cloud VMs brought about [28] [33] [35]. Concretely, we investigate the ease and likelihood, within the constraints of default account limits, of an adversary being able to place her ephemeral container on the same physical host as a victim container by using the AWS Lambda container service as a case study. First, we investigate weaknesses in the container placement scheduler and leverage the weaknesses to increase an attack surface. Second, we examine the container CPU scheduler and identify a covert channel wherein an adversary is able to determine whether or not a neighbor is running a similar expensive job. Third, we take advantage of a shared hardware resource and see how it can be used to break container isolation. Fourth, we combine the previously explored weaknesses to achieve co-residency of containers across adversarial and victim accounts in three different AWS data centers, as well as analyze the general threat that cloud providers face when allowing customers to scale their computing resources. Finally, we explore existing mitigations in the VM and container space against adversarial co-resident behavior, and argue why they may no longer be appropriate and should be re-examined.

3.2 Related Work

Isolation of virtual machines and containers is defined and compared in Soltesz et al. [29], where they define two major categories of isolation: resource and security. They show that general purpose containers are more efficient than VMs, yet still attain a similar level of isolation with regards to resource and security.

Co-residency of virtual machines in the public cloud was first explored by Ristenpart et al. [28]. The research focuses on exploiting VM placement policies and developing a series of co-residency checks, both using the network and hardware, to ensure placement on the same machine. The scale of available adversarial resources was restricted to less than 100 virtual machines. The smaller footprint and scalability of AWS Lambda, however, allows us to explore co-residency at a much wider, faster, and cheaper scale than has been done before.

Subsequent work with VMs has focused on finding faster and cheaper ways to detect and exploit co-residency [33] [35]. Furthermore, focus has shifted towards physical resource contention, as opposed to network cartography, as most public clouds have adopted VPC technology that makes earlier networking co-residency strategies ineffective.

Co-residency of public cloud containers, to our knowledge, has only been explored in Zhang et al. [37], but only on a scale of running up to 30x30 containers. They work with long-lived container frameworks, such as AWS Elastic Beanstalk, and find that the highest probability of co-residency happens approximately one hour after the victim has been invoked. Their work deviates from the ephemeral nature of AWS Lambda, which allows containers to be created and destroyed at a second granularity, and thus leads to a different style of threat heavily influenced by burst scheduling.

Serverless container infrastructures in general are explored in Wang et al. [34], where scalability in the thousands, cold-start latency, and resource efficiency is addressed. They detect severe contention between AWS Lambda functions and a bin-packing-like strategy for lambda scheduling. However, co-residency of containers across customer accounts and security in general is not discussed.

Container vulnerabilities have also been explored with regards to leakage channels [13] [32]. However, the research has mostly worked under the assumption that containers share the same host operating system. We show that sharing a host OS with other customers in the AWS Lambda infrastructure does not seem possible, and thus we focus on hardware leakage channels to identify containers that may not share the same operating system but are still placed on the same physical machine.

To our knowledge, there has been no prior work done on assessing the inherent risk that different sized data centers might face from brute force attacks when allowing users to scale computing resources to a much larger scale than before (tens to thousands).

3.3 AWS Lambda

Amazon implements Lambda through the use of containers: resource-isolated processes. Lambda restricts the size of a container to be anywhere in the user specified range of 128MB to 3GB. Lambda also restricts the runtime of code, within the container, to be less than 5 minutes. Lambda bills a customer only for the running time of a container on the granularity of 100ms. Developers can manually invoke up to 1,000 concurrent lambdas.

Multiple lambda containers can get scheduled on one host operating system and multiple host operating systems can get scheduled on the same hypervisor and physical machine. To identify whether two lambda containers share the same host operating system, we employ a couple of existing OS co-residency checks. By checking static identifiers such as the host id, internal IP address or public IP address, we can identify which containers correspond to the same host OS [13]. Furthermore, we are able to prove OS co-residence by draining the OS supplied /dev/random pseudorandom number generator in one container and checking the available entropy across any container that is sitting on the same host OS [32].

We report that a host operating system, on average, schedules up to 3GB worth of containers on one host OS. We also report that, after running our OS co-residency checks with 1,000 victims on account A and 1,000 adversaries on account B in over 20 trials across two data centers, we have not been able to identify a shared host OS across accounts. For this reason, evaluating container security across accounts in the AWS Lambda environment poses a different challenge than the common state of evaluating container security where a shared host OS is often assumed [13] [32]. Thus, we will choose to focus on hardware vulnerabilities that exist across containers, as well as weaknesses in the lambda placement and CPU scheduler.

3.4 Lambda Placement Scheduler

The lambda placement scheduler schedules lambda containers on particular host operating systems. We found that AWS Lambda, on average, schedules up to 3GB of containers per host OS. Therefore, with every 3-GB lambda that is invoked, a new host OS and potentially a new physical machine will be assigned. On the other hand, for every 128-MB lambda container that is invoked, it is likely to be placed on the same host OS as its concurrently invoked sibling, until the 3-GB limit is reached.

Another property of the container scheduler is the distinction between “warm” and “cold” containers [22]. To optimize for container instantiation time, AWS Lambda will try to reuse previous instantiations of containers. In this way, if the same lambda function is invoked sequentially, the container will be “cold” the first invocation, but “warm” during the second invocation because it will use the same original container [34]. If, however, a lambda function’s

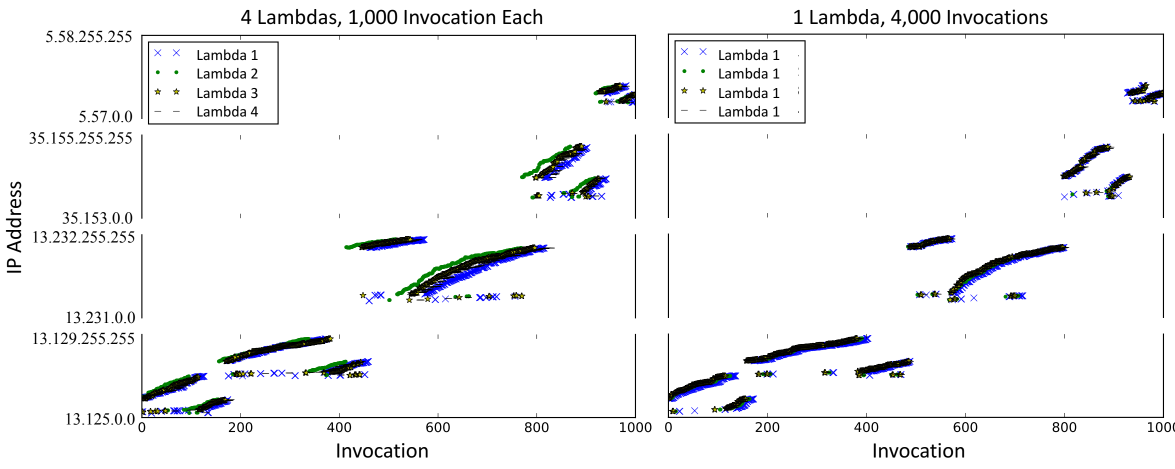

Figure 3.1: Sorted IP Address overlap between container invocations.

behavior is changed (e.g., new package dependency, different uploaded code, long inactivity period), the container will be torn down and any following invocation will be a fresh “cold” start. We report that “cold” containers can experience an average invocation latency of up to 5 seconds, depending upon the number of concurrent invocations.

3.4.1 Manipulating the Lambda Scheduler

To increase the surface of an attack, it is better for an attacker to spawn 1,000 3-GB lambdas as opposed to 1,000 128-MB lambdas, thus increasing the attack surface by a factor of 24. By spawning 3-GB lambdas the adversary should never be co-resident with another lambda on the same OS, yet will still have the ability to be co-resident with a victim physical machine, as a physical machine can host multiple operating systems. Choosing the 1,000 3-GB attack strategy has a cost more expensive than invoking 1,000 128-MB lambdas, but notably by a factor smaller than 24: \$0.13 vs \$1.45 (for a 30-second attack).

Another method to increase the surface of an attack is to invoke more “cold” lambdas. Invoking a “cold” lambda increases the probability of being scheduled on a new host OS, and thus potentially a new physical machine. We test the probability of being scheduled on a new host OS by comparing the amount of unique IP addresses generated as a result of invoking 1,000 instances of four independent lambda functions (each being cold) vs. invoking 1,000 instances of the same lambda function four times (the last three times being warm). Each unique IP address identifies a unique host OS, as discussed in Section 3.3. Figure 3.1 shows that invoking four lambda functions creates less IP address overlap, where each lambda invocation number is assigned to more unique IP addresses across the four independent functions. Concretely, invoking four different lambda functions results in 1,728 unique IP addresses and consistently invoking the same lambda function results in 487 unique IP addresses.

By being assigned to more unique IP addresses, an adversary is therefore able to reach new host operating systems, thereby increasing the attack coverage. We repeat the experiment over a dozen of times in different data centers and find that similar trends are always seen and even more prominent in larger data centers where there are more IP addresses available (e.g., us-east-1 and us-east-2). In some cases, we have seen near 4,000 unique IP addresses across 4 independent lambda functions.

To our knowledge there are currently two techniques for increasing the probability of hitting a "cold" lambda/new host OS within a short period of time. (1) Constantly create fresh lambda functions between sequential invocations. (2) Change the uploaded code/ lambda function behavior. We discuss both strategies further in Section 3.7.1 and how they interact with potential mitigations.

3.5 CPU Scheduler

AWS Lambda uses a variant of the Linux Completely Fair Scheduler (CFS) to partition the use of CPU cores across multiple lambda containers [5]. The amount of CPU time a lambda container is given is proportional to the size of a lambda container. For example, a 1024-MB lambda container should receive 8 times more CPU usage time than a 128-MB lambda container. To test this expected behavior, we run one 128-MB lambda and one 1024-MB lambda concurrently

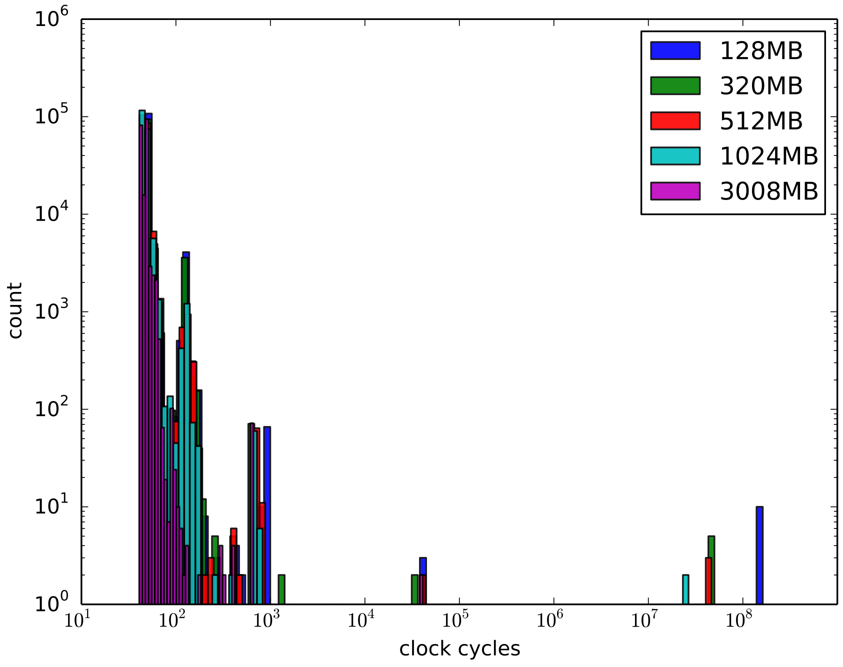

Figure 3.2: Sampling instruction latency across different container sizes.

and report that the 128-MB lambda uses roughly 25% total CPU across all running processes whereas the 1024-MB lambda uses roughly 200% total CPU.

We further test this behavior by sampling the execution time of the `rdrand` instruction and show that, as the size of the lambda increases, the amount of execution time outliers begin to decrease, and completely disappear with the 3-GB lambda. Figure 3.2 shows that the largest execution time a 3-GB lambda experiences is below 10^5 clock cycles, whereas the smaller lambda sizes experience execution times above 10^7 clock cycles. We believe these execution time outliers are demonstrative of context switches and imply that a container with more dedicated CPU time (e.g., 3-GB) experiences context switches for a shorter amount of clock cycles.

Another property of the Linux CFS is the scheduler allocation of extra CPU resources to waiting jobs when the CPU is idle. In the context of AWS Lambda, if there are no other lambda containers on the same host as one 128-MB lambda, then the 128-MB lambda will see an increase in performance. We test the behavior of the CPU scheduler by comparing the performance of

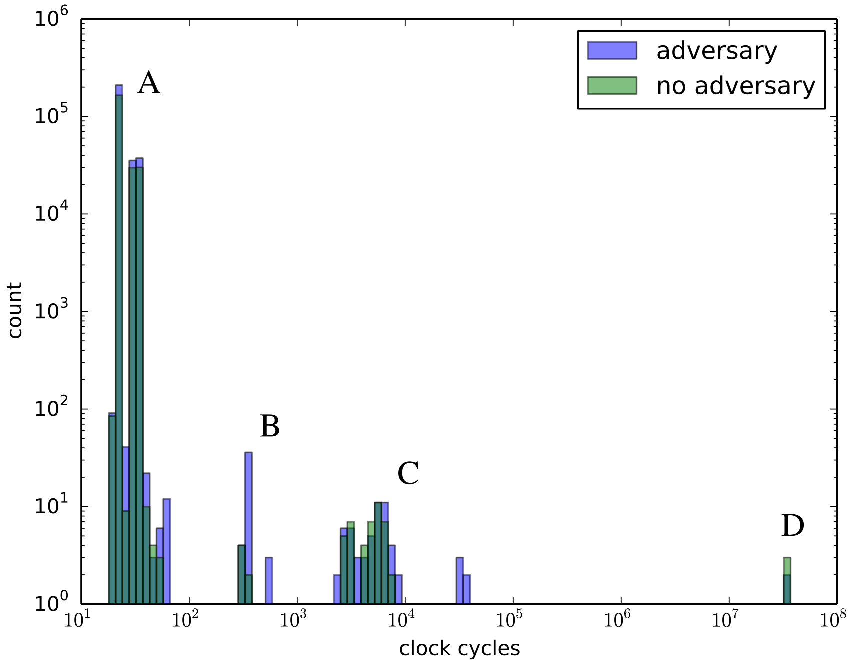

Figure 3.3: The effects of a 1024-MB adversarial container on a 320-MB container.

different size containers with and without the presence of another container.

In Figure 3.3 we show the number of clock cycles an `rdrand` query takes, across 200,000 trials, with and without the presence of one 1024-MB "adversarial" lambda that is constantly querying the `rdrand` instruction. We report that the majority of `rdrand` queries take between 10^1 and 10^2 clock cycles (group A). All measurements above 10^3 clock cycles (groups C and D) are consistent with context switch measurements as depicted in Figure 3.2. We believe that the increased latencies between 10^2 and 10^3 clock cycles (group B) are due to contention over `rdrand`, as those latencies dramatically increase only when the adversarial lambda is running. For simplicity in measurement, we will thus be using the average time across *n* instruction latency measurements to determine whether an adversary is increasing contention of the `rdrand` instruction.

It is important to note though, as seen in Figure 3.2, that for a 3-GB lambda the context switch times overlap in the 10^2 to 10^3 clock cycle range. Consequently, it is hard to tell whether

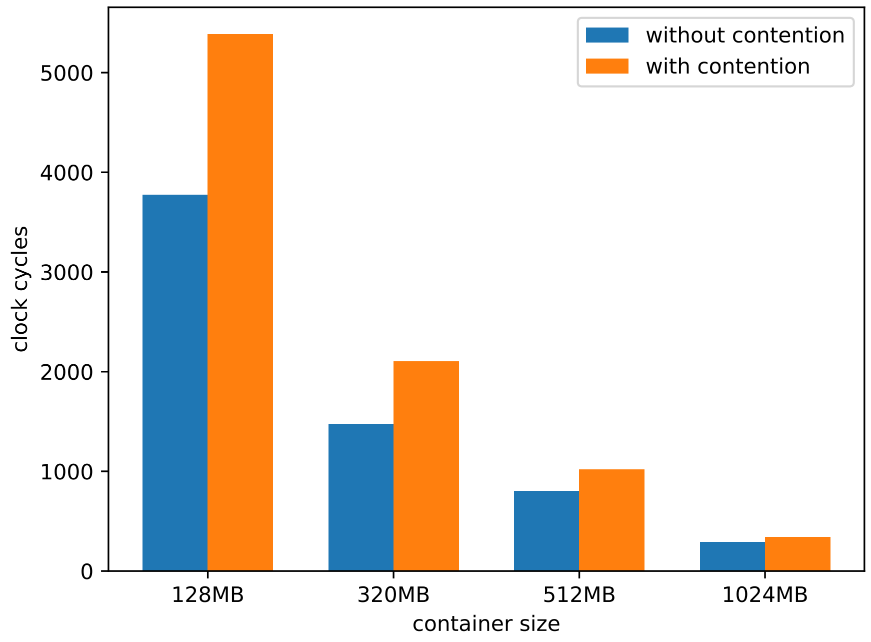

Figure 3.4: The effects of a 1024-MB adversarial container across container sizes.

those samples are due to context switches or `rdrand` contention and it thus prompts us to use the smaller, 320-MB lambda to measure contention in our subsequent experiments.

To further explore the contention effect across lambda sizes, we execute the same experiment as described previously across different lambda sizes with and without the presence of one 1024-MB lambda that is increasing contention over the `rdrand` instruction. Figure 3.4 shows an increase in the average amount of clock cycles to a successful `rdrand` call when there is contention present. Moreover, the decrease in performance is relatively stronger in the smaller lambdas, where the 128-MB lambda sees an average decrease in performance of 42% with the adversary present . We take advantage of the amplification in contention when running smaller containers in experiments described below.

3.6 Determining Co-Residence

We take advantage of the container scheduler, the CPU scheduler, and a shared hardware resource (`rdrand`) to show hardware co-residency of lambda containers across accounts, even when they do not share the same underlying OS.

3.6.1 Attack strategy

Intel's hardware random number generator, invoked by the `rdrand` instruction, is shared among all physical cores of a machine and is therefore an ideal candidate for being a contested hardware resource. Previous work has established a covert channel across cores using the hardware random number generator, yet we are the first, to our knowledge, to use `rdrand` [11] as a method of co-residency checking across containers.

Thus, the generalized attack strategy is as follows:

1. Schedule one small container (e.g., 128MB, 320MB): record the average latency for an operation (preferably an operation using a resource that faces contention over cores (e.g., `rdrand`).

2. Sequentially schedule one large container (e.g., 1024MB+) on the same hardware host: create contention over the chosen operation (e.g., create an infinite loop calling `rdrand`).

3. Concurrently schedule the original one small container (e.g., 128MB, 320MB): record the average latency for the chosen operation. It will noticeably go up because it can no longer use as many of the free pool of CPU resources on the physical host, as the adversarial container is now using those resources.

4. To run the same attack strategy again: invoke cold lambdas, as discussed in Section 3.4, so as to not get penalized by the CPU scheduler, as discussed in Section 3.7.1.

To achieve a greater probability of co-resident placement across accounts, we design an experiment with a large attack surface within the default AWS account limits. Concretely, on the victim account, we invoke 1,000 concurrent 320-MB victim containers each of which measure the average runtime of an `rdrand` call across 25,000,000 trials, go to sleep for 50 seconds, and again record the average runtime of an `rdrand` call across 25,000,000 trials. It takes roughly 30 seconds to determine the average runtime of an `rdrand` call across 25,000,000 trials. On the adversary account, we invoke 1,000 adversarial 3-GB containers which infinitely loop the `rdrand` instruction for only 30 seconds after the first batch of victim containers have run, thereby only creating a contention of resources for the second reported average from the victim containers. We then analyze the victim container averages with and without the adversary containers running, and see if we find a significant performance gain without the presence of adversarial containers.

3.6.2 Evidence of co-residency

Figure 3.5 shows the behavior of 10 of the 1,000 victim account containers when running 1,000 adversarial account containers. The co-resident behavior is similar to the expected co-resident behavior seen in Figure 3.4, where there is nearly a 20% decrease in performance during the second measurement when the adversary is running. We plot the behavior of a "safe" container group during the same experiment that does not exhibit any significant changes between measurements, thus implying that no adversarial containers were scheduled on the same physical host. We also include the behavior of a "control" container experiment when no adversarial containers were run. The control experiment highlights the fact that even with the inherent noise present from executing the experiment in a public cloud, the co-resident results are still significant.

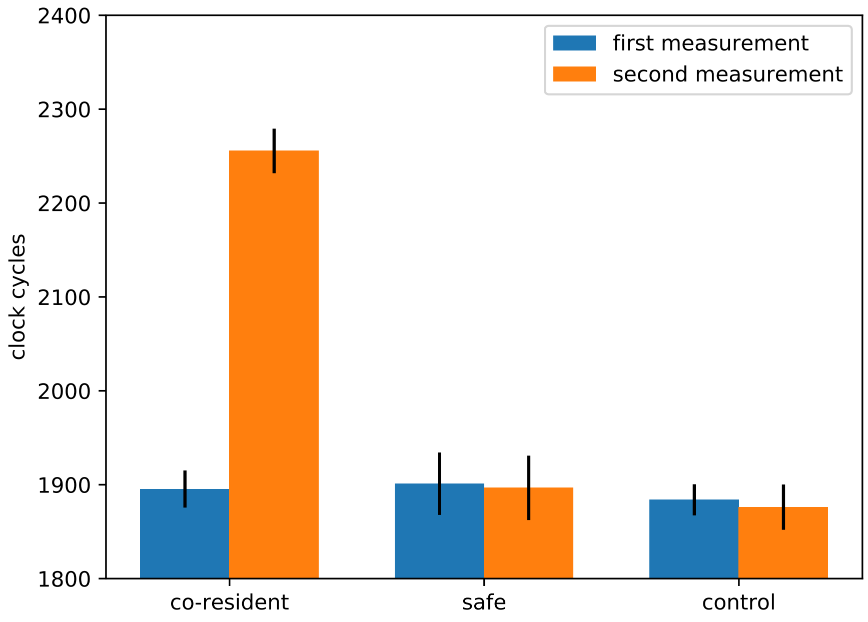

Figure 3.5: Measuring the effects of co-residency.

3.6.3 Evaluation of Co-Residency

We run our attack strategy with both victim and adversary containers and report our results in Table 3.1. We run our experiments across different regions, different days of the week, and different times of the day. We report the minimum number of victim containers needed to be invoked in order to begin seeing the co-residency effects, similar to Figure 3.4, in a select number of victim containers. As the number of available IP prefixes/size of the data centers increase, the number of victims needed to achieve co-residency also increases. We also report the total number of co-resident victim containers after spawning 1,000 victim containers. Note that the number of co-resident containers is often a multiple of 10 as the lambda scheduler schedules 320-MB containers in batches of 10 on the same host OS, and thus the entire host OS becomes victim to the adversary container.

The price of the co-residency attack sums up to less than $1.45 [23], the total cost of spawning 1,000 3GB lambda containers for 30 seconds.

Table 3.1: Results of running co-residency attacks with 1,000 adversary containers.

Region	**Available IP prefixes**	**Min # victims to achieve co-residency**	**# Co-resident victims**	**# Co-resident host OS**
ap-south-1	44	406	10	1
ap-south-1	44	545	20	2
ap-south-1	44	210	30	3
us-east-2	60	997	10	1
us-east-1	226	934	2	1

3.6.4 General threat of co-residency in data centers

When cloud providers allow customers to scale computing resources in the thousands, it is important to evaluate the potential threat that customers could face. To asses the general threat, we create a probabilistic model based on the following formula:

$$chance\ of\ co-residency = 1 - ((hosts - adversaries)/hosts)^{victims}$$

where *hosts* is the number of physical machines in a data center that a victim or adversary container could run on, *adversaries* is the number of adversary containers invoked, and *victims* is the number of victim containers invoked.

To apply this model to AWS Lambda, we choose to evaluate the general threat of the ap-south-1 region. To assess the size of the ap-south-1 region, we count the number of IP addresses derived from the ip-range prefixes that AWS assigns to AWS services [16]. By doing so, we estimate that, at maximum, there are 531,720 unique IP addresses in the ap-south-1 region. We therefore set the number of hosts to be 531,720 as an absolute maximum bound on the number of physical machines a lambda container could be scheduled on in the ap-south-1 region. Though we know that IP addresses get assigned to unique host operating systems and a physical host can run multiple host operating systems, we still choose to keep one physical machine = unique IP Address as the maximum number of hosts, as a conservative example. We also set adversaries to a constant 1,000 containers, the default maximum for an account. Since the attack strategy previously described involves invoking 3-GB adversary containers, each container will be assigned to its own host OS and thus its own IP address. In this way, 1,000 containers is equivalent to 1,000 IP addresses. We vary the number of victims needed to asses the general threat. It is important to note that our model assumes that there is a uniform scheduling distribution (i.e., it is not taking into account that an existing third party load might be on a victim's physical host and thus the adversary is less likely to hit it).

Nonetheless, it can be seen that the data in Table 3.1 roughly matches with our model. Figure 3.6 plots the model and the general threat that the AWS ap-south-1 data center faces with the AWS Lambda service. We report that if ap-south-1 does in fact have 531,720 physical hosts dedicated to AWS Lambda (which we hypothesize is a gross overestimate), then it only takes roughly 400 victims for an adversary to have over a 50% chance of achieving co-residence, a number that is quite reasonable for a target company such as Netflix to be running. Furthermore, one could interact and overload an event-driven service so as to have it to inadvertently invoke more containers to thus increase the attack surface and probability of co-residency [1].

The attack strategy presented in this model and overall work is a brute-force strategy and has no way of targeting a particular victim. However, since the total cost of a 30-second attack is $1.45, as mentioned previously, cost is not a drawback to executing an attack through a brute-force strategy. Furthermore, attacks that are exploited through the use of a shared hardware resource, such as Spectre [19], would be a good candidate to deploy at this brute-force scale, as an adversary could just attempt to read all possible out of bound memory across 1,000 containers, and then just parse the results.

The general threat also illuminates the fact that smaller data centers have an increased likelihood of a chance of collision and thus are more vulnerable to co-residency. In this way, it is most likely safer for a customer to use a larger data center.

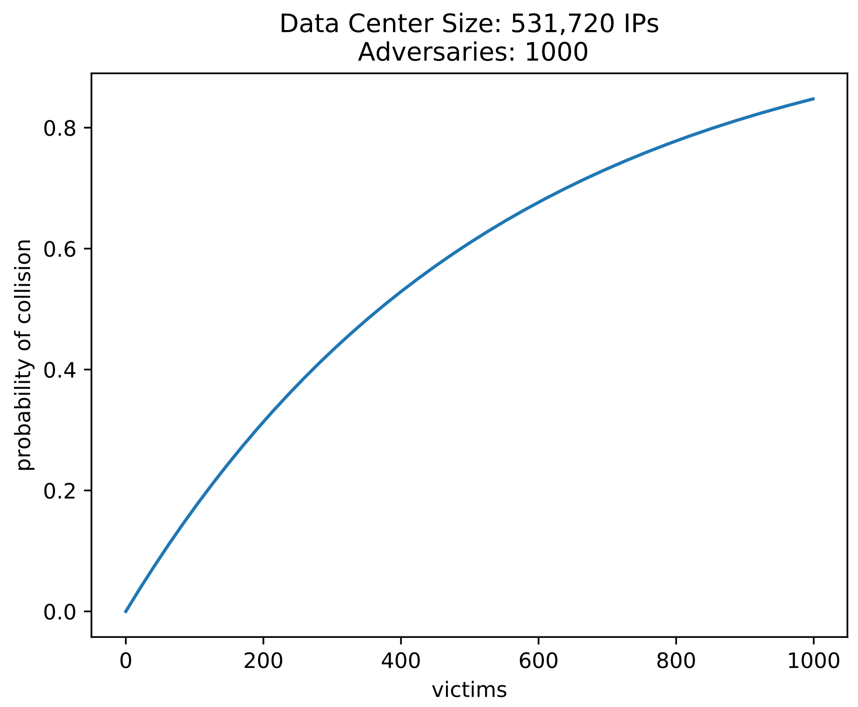

Figure 3.6: Amount of victim containers needed for a more probable collision.

3.7 Potential mitigations and their limitations

We will look at different aspects of the vulnerabilities introduced throughout this work and asses the current available mitigations, their effectiveness, and potential future directions for mitigations.

3.7.1 Tricking Amazon's CPU scheduler mitigation

When running our experiments we notice that AWS Lambda mitigates the container's abuse of CPU resources by having the Completely Fair Scheduler occasionally limit the amount of available CPU resources a lambda container uses during subsequent runs. CFS can thus hinder our co-residency detection strategy, if a victim container is no longer seeing a significant enough change in runtime. However, this penalty can be avoided in two ways:

1. Invoke an identical, yet brand new, lambda function with no prior history of average

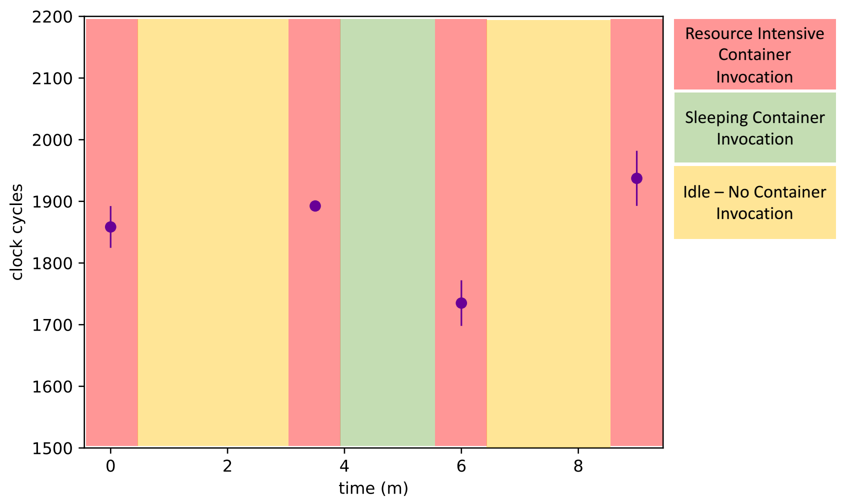

Figure 3.7: Tricking the Completely Fair Scheduler with different container behavior.

resource usage.

2. Invoke the same lambda function to run a program that sleeps, thereby lowering the history of average resource usage.

The first proposed method is by far the fastest and cheapest. However, it is also, as we have discovered, detectable by AWS if done too often and thus not favorable.

The second proposed method requires careful execution. We conduct an experiment where we invoke a container with a resource intensive job (querying `rdrand` 20,000,000 times), and then put the job to sleep. We measure the execution time of the resource intensive job before and after the container is put to sleep. Figure 3.7 shows the average latency of the resource intensive job decreases by over 100 clock cycles immediately after the container is put to sleep. However, the latency of the resource intensive job increases again if the container is not immediately put back to sleep. We run the same experiment across 4 trials and consistently see the same behavior.

Beyond the current mitigation introduced by AWS, a way to mitigate an attack using the CFS is to enforce the scheduler not to give up available resources to lambdas which are not initially entitled to them (i.e., smaller lambdas). However, this mitigation would be unfavorable to a cloud provider, like Amazon, as it is not work conserving and leads to an idle CPU.

3.7.2 Malicious behavior mitigations

Previous work has monitored and mitigated malicious tenant behavior on the cloud. However, none of the mitigations have been explored at the speed and scale that a brute force attack on AWS Lambda could achieve. For example, Zhang et al. propose a malicious behavior mitigation that collects samples at 30-second intervals. However, this mitigation would need to be re-evaluated for a finer-grained sampling rate, for services such as AWS Lambda, as containers are often spawned and shut-down in smaller time periods [36]. Also, collecting samples for history can become irrelevant once new containers are created and old ones are torn down. The Sherlock system also involves profiling a container, but at the granularity of minutes; again, minutes are too long for ephemeral containers [18].

3.8 Open Problems and Directions

In this work we have presented a covert channel where both the “victim” and “adversary” accounts are working together to detect if they are co-resident by creating contention over `rdrand`. Future work will involve demonstrating a side channel where the adversary account can profile the victim without him being aware. Profiling an uncooperative victim presents its own challenge in that the current nature of serverless computations are often ephemeral, thereby restricting attacks to complete within seconds. Furthermore, the ephemeral nature prompts two research questions: what valuable information can be extracted from a victim container that is running at millisecond granularity and to what extent does the ephemeral nature of containers protect the victim from attackers?

3.9 Summary

The AWS Lambda service introduces an easy way to achieve co-residency of containers across accounts using a brute force strategy. Giving customers the ability to scale compute instances in the thousands instantly presents cloud providers a challenge to scheduling and isolating abnormal behavior. As cloud providers give customers more flexibility and freedom to scale computing resources, it is important to asses the inherent risk that results. In this chapter, we demonstrate that co-residency can easily be achieved at scale in data centers all around world, all while the adversary stays within the default limit her account provides. Moreover, multiple attack accounts only amplify the threat of achieving co-residency.

3.10 Acknowledgments

Chapter 3, in part, is currently being prepared for submission for publication of the material. Izhikevich, Liz; Feteih, Nadah. The thesis author is the primary investigator and author of this material.

Chapter 4

Conclusion and Future Directions

This thesis explores a serverless cloud computing model that runs on stateless container invocations. We show that the responsiveness, concurrency, and cost of the serverless cloud computing model, on the one hand, enables building systems with new designs and tradeoffs, yet on the other hand, introduces potential security risks due to an increased probability of achieving co-residence.

This thesis first describes Sprocket, a serverless video processing framework for the cloud, and shows its modular structure by demonstrating how a simple filter application can be turned into a complex facial recognition application. Sprocket can be configured to process a 30-minute video 1000-way parallel in under a minute for less than \$1. We further show how Sprocket takes advantage of the scalable nature of container invocations by designing a streaming scheduler that seamlessly invokes a variable amount of containers based on the current content delivery deadline.

While the serverless cloud computing model enables burst-parallel systems like Sprocket to be built, there are also certain risks that the serverless cloud computing model introduces. Concretely, we present a covert channel in the AWS Lambda framework using the `rdrand` instruction to demonstrate evidence of co-residency of containers across accounts in three different data centers. We find as many as 30 co-resident containers during one 30-second “attack.” We further define a model which illustrates that, if cloud providers allow customers to scale their compute resources in the thousands, then co-residency becomes probabilistically achievable.

The contributions of this thesis present the first steps of designing a sophisticated serverless video processing system and demonstrates the potential security vulnerabilities of the serverless computing model. More work, however, can be done at the intersection of the two projects. Specifically, we demonstrate our ability to detect context switches and contention of resources in the AWS Lambda computing environment. Since our burst-parallel system, Sprocket, runs on AWS Lambda in the public cloud and uses a large amount of concurrent container invocations, it therefore appears possible that Sprocket’s system behavior should be detectable by an adversary in the same data center.

The techniques we describe to detect contention of resources currently do not have the ability to fingerprint where the contention is coming from (e.g., our targeted victim or another resource consuming container). However, with the ability to identify a victim, it seems that one should be able to leak information about a running burst-parallel system’s behavior. By demonstrating a side channel on the behavior of an entire system like Sprocket, we would begin to uncover security implications of burst-parallel systems in general and perhaps begin to think about how to obfuscate such systems against adversaries in the same data center.

The cost effectiveness and scalability of the serverless computational model is ideal for highly parallel and time-sensitive systems, and it will undoubtedly further gain popularity in upcoming years. It is also for these reasons that it is imperative that we focus on the security of such frameworks to ensure that the integrity and privacy of burst parallel serverless systems are not compromised.

Appendix A

Facial Recognition Pipespec

This appendix chapter provides an example “pipespec” for the Facial Recognition pipeline discussed in Section 2.3. The pipespec demonstrates the stage dependencies for the facial recognition pipeline as well as the delivery functions used to organize and send data from the upstream to the downstream stage. Note that input_0, input_1, and output_0 are bound to runtime parameters.

```
{
  ["nodes":
    {
      "name": "parallelize_link",
      "stage": "parallelize_link",
      "config": {
      }
    },
    {
      "name": "matchFace",
      "stage": "matchFace",
      "config": {
      }
    },
    {
      "name": "decode",
      "stage": "decode",
      "config": {
      }
    },
    {
      "name": "scenechange",
      "stage": "scenechange",
      "config": {
      }
    },
    {
      "name": "rek",
      "stage": "rek",
      "delivery_function": "serialized_arbitrary_segment_delivery_func",
      "config": {
      }
    },
    {
      "name": "draw",
      "stage": "draw",
      "delivery_function": "serialized_arbitrary_segment_delivery_func",
      "config": {
      }
    },
    {
      "name": "encode",
      "stage": "encode_frame_list",
      "delivery_function": "serialized_frame_delivery_func",
      "config": {
      }
    }
  ],}
```

Figure A.1: The Facial Recognition pipespec example.

```

 "streams":
 [
   {
     "src": "input_0:video_link",
     "dst": "parallelize_link:video_link"
   },
   {
     "src": "input_1:person",
     "dst": "matchFace:person"
   },
   {
     "src": "parallelize_link:chunked_link",
     "dst": "decode:chunked_link"
   },
   {
     "src": "decode:frames",
     "dst": "scenechange:frames"
   },
   {
     "src": "scenechange:scene_list",
     "dst": "rek:scene_list"
   },
   {
     "src": "rek:frame",
     "dst": "draw:frame"
   },
   {
     "src": "draw:frame",
     "dst": "encode:frame_list"
   },
   {
     "src": "encode:chunks",
     "dst": "output_0:chunks"
   }
 ]

}
```

Figure A.1: The Facial Recognition pipespec example, continued.

Bibliography

[1] Intrusion and Exfiltration in Server-less Architectures. https://media.ccc.de/v/33c3-7865-gone_in_60_milliseconds.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010.

[3] Avengers Trailer. https://www.youtube.com/watch?v=eMobkagZu64.

[4] AWS EC2. https://aws.amazon.com/ec2/.

[5] How Does Proportional CPU Allocation work with AWS Lambda. https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-with-aws-lambda-41cd44da3cac.

[6] Cisco Visual Networking Index: Forecast and Methodology, 2016-2021. https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.html.

[7] Colbert Interview. https://www.youtube.com/watch?v=Y6XXMGUb5kU.

[8] Companies Using AWS Lambda. https://stackshare.io/aws-lambda/in-stacks.

[9] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[10] Earth. https://www.youtube.com/watch?v=wnhvanMdx4s.

[11] Dmitry Evtyushkin and Dmitry Ponomarev. Covert Channels Through Random Number Generator: Mechanisms, Capacity Estimation and Mitigations. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 843–857, New York, NY, USA, 2016. ACM.

[12] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads. In *Proceedings of the 14th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, March 2017.

[13] Xing Gao, Zhongshu Gu, Mehmet Kayaalp, Dimitrios Pendarakis, and Haining Wang. ContainerLeaks: Emerging Security Threats of Information Leakages in Container Clouds. *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 237–248, 2017.

[14] Google Cloud Vision API. https://cloud.google.com/vision/.

[15] Thomas Huang, Ziyou Xiong, and Zhenqiu Zhang. *Face Recognition Applications*, pages 371–390. Springer New York, New York, NY, 2005.

[16] AWS IP Ranges. https://ip-ranges.amazonaws.com/ip-ranges.json.

[17] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the Cloud: Distributed Computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 445–451. ACM, 2017.

[18] K. Joshi, A. Raj, and D. Janakiram. Sherlock: Lightweight Detection of Performance Interference in Containerized Cloud Services. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 522–530, Dec 2017.

[19] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *ArXiv e-prints*, January 2018.

[20] Yongchul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. A Study of Skew in MapReduce Applications. The 5th Open Cirrus Summit, 2011.

[21] AWS Lambda. https://aws.amazon.com/lambda/.

[22] Understanding Container Reuse in AWS Lambda. https://aws.amazon.com/blogs/compute/container-reuse-in-lambda/.

[23] AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[24] Microsoft Computer Vision and Cognitive Services API. https://azure.microsoft.com/en-us/services/cognitive-services/computer-vision/.

[25] Nature. https://www.youtube.com/watch?v=eMobkagZu64.

[26] Netflix and AWS Lambda. https://aws.amazon.com/solutions/case-studies/netflix-and-aws-lambda/.

[27] AWS Rekognition. https://aws.amazon.com/rekognition/.

[28] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[29] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM.

[30] Apache Spark. http://spark.apache.org/.

[31] AWS Step Functions. https://aws.amazon.com/step-functions/.

[32] Understanding and Hardening Linux Containers. https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf.

[33] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-tenant Public Clouds. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 913–928, Berkeley, CA, USA, 2015. USENIX Association.

[34] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association.

[35] Si Yu, Gui Xiaolin, Lin Jiancai, Zhang Xuejun, and Wang Junfei. Detecting VMs co-residency in cloud: Using cache-based side channel attacks. *Elektronika ir Elektrotechnika*, 19(5):73–78, 2013.

[36] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. Memory DoS Attacks in Multi-tenant Clouds: Severity and Mitigation. *CoRR*, abs/1603.03404, 2016.

[37] Weijuan Zhang, Xiaoqi Jia, Chang Wang, Shengzhi Zhang, Qingjia Huang, Mingsheng Wang, and Peng Liu. A Comprehensive Study of Co-residence Threat in Multi-tenant Public PaaS Clouds. In Kwok-Yan Lam, Chi-Hung Chi, and Sihan Qing, editors, *Information and Communications Security*, pages 361–375, Cham, 2016. Springer International Publishing.